Red Hat build of Debezium 1.9.7

Debezium User Guide

For use with Debezium 1.9.7

Last Updated: 2022-11-17

Red Hat build of Debezium 1.9.7 Debezium User Guide

For use with Debezium 1.9.7

Legal Notice

Copyright © 2022 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide describes how to use the connectors provided with Debezium.

Table of Contents

PREFACE 8
MAKING OPEN SOURCE MORE INCLUSIVE 8

CHAPTER 1. HIGH LEVEL OVERVIEW OF DEBEZIUM 9
1.1. DEBEZIUM FEATURES 9
1.2. DESCRIPTION OF DEBEZIUM ARCHITECTURE 9

CHAPTER 2. REQUIRED CUSTOM RESOURCE UPGRADES 11

CHAPTER 3. DEBEZIUM CONNECTOR FOR DB2 12
3.1. OVERVIEW OF DEBEZIUM DB2 CONNECTOR 12
3.2. HOW DEBEZIUM DB2 CONNECTORS WORK 13
3.2.1. How Debezium Db2 connectors perform database snapshots 14
3.2.1.1. Ad hoc snapshots 14
3.2.1.2. Incremental snapshots 15
3.2.2. How Debezium Db2 connectors read change-data tables 19
3.2.3. Default names of Kafka topics that receive Debezium Db2 change event records 20
3.2.4. About the Debezium Db2 connector schema change topic 20
3.2.5. Debezium Db2 connector-generated events that represent transaction boundaries 24
3.3. DESCRIPTIONS OF DEBEZIUM DB2 CONNECTOR DATA CHANGE EVENTS 26
3.3.1. About keys in Debezium db2 change events 28
3.3.2. About values in Debezium Db2 change events 29
3.4. HOW DEBEZIUM DB2 CONNECTORS MAP DATA TYPES 38
3.5. SETTING UP DB2 TO RUN A DEBEZIUM CONNECTOR 43
3.5.1. Configuring Db2 tables for change data capture 44
3.5.2. Effect of Db2 capture agent configuration on server load and latency 46
3.5.3. Db2 capture agent configuration parameters 46
3.6. DEPLOYMENT OF DEBEZIUM DB2 CONNECTORS 47
3.6.1. Obtaining the Db2 JDBC driver 47
3.6.2. Db2 connector deployment using AMQ Streams 47
3.6.3. Using AMQ Streams to deploy a Debezium Db2 connector 48
3.6.4. Deploying a Debezium Db2 connector by building a custom Kafka Connect container image from a Dockerfile 53
3.6.5. Verifying that the Debezium Db2 connector is running 57
3.6.6. Descriptions of Debezium Db2 connector configuration properties 61
3.7. MONITORING DEBEZIUM DB2 CONNECTOR PERFORMANCE 80
3.7.1. Monitoring Debezium during snapshots of Db2 databases 81
3.7.2. Monitoring Debezium Db2 connector record streaming 83
3.7.3. Monitoring Debezium Db2 connector schema history 85
3.8. MANAGING DEBEZIUM DB2 CONNECTORS 86
3.9. UPDATING SCHEMAS FOR DB2 TABLES IN CAPTURE MODE FOR DEBEZIUM CONNECTORS 87
3.9.1. Performing offline schema updates for Debezium Db2 connectors 87
3.9.2. Performing online schema updates for Debezium Db2 connectors 88

CHAPTER 4. DEBEZIUM CONNECTOR FOR MONGODB 90
4.1. OVERVIEW OF DEBEZIUM MONGODB CONNECTOR 90
4.2. HOW DEBEZIUM MONGODB CONNECTORS WORK 92
4.2.1. MongoDB topologies supported by Debezium connectors 93
4.2.2. How Debezium MongoDB connectors use logical names for replica sets and sharded clusters 94
4.2.3. How Debezium MongoDB connectors perform snapshots 94
4.2.3.1. Ad hoc snapshots 94
4.2.3.2. Incremental snapshots 96

4.2.4. How the Debezium MongoDB connector streams change event records 100
4.2.5. Default names of Kafka topics that receive Debezium MongoDB change event records 101
4.2.6. How event keys control topic partitioning for the Debezium MongoDB connector 101
4.2.7. Debezium MongoDB connector-generated events that represent transaction boundaries 101
4.3. DESCRIPTIONS OF DEBEZIUM MONGODB CONNECTOR DATA CHANGE EVENTS 103
4.3.1. About keys in Debezium MongoDB change events 104
4.3.2. About values in Debezium MongoDB change events 107
4.4. SETTING UP MONGODB TO WORK WITH A DEBEZIUM CONNECTOR 118
4.5. DEPLOYMENT OF DEBEZIUM MONGODB CONNECTORS 118
4.5.1. MongoDB connector deployment using AMQ Streams 118
4.5.2. Using AMQ Streams to deploy a Debezium MongoDB connector 119
4.5.3. Deploying a Debezium MongoDB connector by building a custom Kafka Connect container image from a Dockerfile 124
4.5.4. Verifying that the Debezium MongoDB connector is running 127
4.5.5. Descriptions of Debezium MongoDB connector configuration properties 131
4.6. MONITORING DEBEZIUM MONGODB CONNECTOR PERFORMANCE 142
4.6.1. Monitoring Debezium during MongoDB snapshots 142
4.6.2. Monitoring Debezium MongoDB connector record streaming 144
4.7. HOW DEBEZIUM MONGODB CONNECTORS HANDLE FAULTS AND PROBLEMS 146

CHAPTER 5. DEBEZIUM CONNECTOR FOR MYSQL 151
5.1. HOW DEBEZIUM MYSQL CONNECTORS WORK 151
5.1.1. MySQL topologies supported by Debezium connectors 151
5.1.2. How Debezium MySQL connectors handle database schema changes 152
5.1.3. How Debezium MySQL connectors expose database schema changes 153
5.1.4. How Debezium MySQL connectors perform database snapshots 158
5.1.4.1. Ad hoc snapshots 159
5.1.4.2. Incremental snapshots 160
5.1.5. Default names of Kafka topics that receive Debezium MySQL change event records 164
5.2. DESCRIPTIONS OF DEBEZIUM MYSQL CONNECTOR DATA CHANGE EVENTS 166
5.2.1. About keys in Debezium MySQL change events 168
5.2.2. About values in Debezium MySQL change events 169
5.3. HOW DEBEZIUM MYSQL CONNECTORS MAP DATA TYPES 180
5.4. SETTING UP MYSQL TO RUN A DEBEZIUM CONNECTOR 186
5.4.1. Creating a MySQL user for a Debezium connector 186
5.4.2. Enabling the MySQL binlog for Debezium 188
5.4.3. Enabling MySQL Global Transaction Identifiers for Debezium 189
5.4.4. Configuring MySQL session timesouts for Debezium 190
5.4.5. Enabling query log events for Debezium MySQL connectors 191
5.5. DEPLOYMENT OF DEBEZIUM MYSQL CONNECTORS 191
5.5.1. MySQL connector deployment using AMQ Streams 192
5.5.2. Using AMQ Streams to deploy a Debezium MySQL connector 192
5.5.3. Deploying Debezium MySQL connectors by building a custom Kafka Connect container image from a Dockerfile 197
5.5.4. Verifying that the Debezium MySQL connector is running 201
5.5.5. Descriptions of Debezium MySQL connector configuration properties 205
5.6. MONITORING DEBEZIUM MYSQL CONNECTOR PERFORMANCE 229
5.6.1. Monitoring Debezium during snapshots of MySQL databases 229
5.6.2. Monitoring Debezium MySQL connector record streaming 232
5.6.3. Monitoring Debezium MySQL connector schema history 235
5.7. HOW DEBEZIUM MYSQL CONNECTORS HANDLE FAULTS AND PROBLEMS 236

CHAPTER 6. DEBEZIUM CONNECTOR FOR ORACLE 238

6.1. HOW DEBEZIUM ORACLE CONNECTORS WORK 238
6.1.1. How Debezium Oracle connectors perform database snapshots 238
6.1.1.1. Ad hoc snapshots 240
6.1.1.2. Incremental snapshots 241
6.1.2. Default names of Kafka topics that receive Debezium Oracle change event records 245
6.1.3. How Debezium Oracle connectors expose database schema changes 245
6.1.4. Debezium Oracle connector-generated events that represent transaction boundaries 250
6.1.4.1. How the Debezium Oracle connector enriches change event messages with transaction metadata 251
6.1.5. How the Debezium Oracle connector uses event buffering 251
6.1.6. How the Debezium Oracle connector detects gaps in SCN values 252
6.1.7. How Debezium manages offsets in databases that change infrequently 253
6.2. DESCRIPTIONS OF DEBEZIUM ORACLE CONNECTOR DATA CHANGE EVENTS 253
6.2.1. About keys in Debezium Oracle connector change events 254
6.2.2. About values in Debezium Oracle connector change events 255
6.3. HOW DEBEZIUM ORACLE CONNECTORS MAP DATA TYPES 263
6.4. SETTING UP ORACLE TO WORK WITH DEBEZIUM 272
6.4.1. Compatibility of the Debezium Oracle connector with Oracle installation types 273
6.4.2. Schemas that the Debezium Oracle connector excludes when capturing change events 273
6.4.3. Preparing Oracle databases for use with Debezium 274
6.4.4. Resizing Oracle redo logs to accommodate the data dictionary 274
6.4.5. Creating an Oracle user for the Debezium Oracle connector 274
6.4.6. Support for Oracle standby databases 277
6.5. DEPLOYMENT OF DEBEZIUM ORACLE CONNECTORS 277
6.5.1. Obtaining the Oracle JDBC driver 278
6.5.2. Debezium Oracle connector deployment using AMQ Streams 278
6.5.3. Using AMQ Streams to deploy a Debezium Oracle connector 279
6.5.4. Deploying a Debezium Oracle connector by building a custom Kafka Connect container image from a Dockerfile 284
6.5.5. Configuration of container databases and non-container-databases 288
6.5.6. Verifying that the Debezium Oracle connector is running 288
6.6. DESCRIPTIONS OF DEBEZIUM ORACLE CONNECTOR CONFIGURATION PROPERTIES 292
6.7. MONITORING DEBEZIUM ORACLE CONNECTOR PERFORMANCE 316
6.7.1. Debezium Oracle connector snapshot metrics 317
6.7.2. Debezium Oracle connector streaming metrics 319
6.7.3. Debezium Oracle connector schema history metrics 326
6.8. HOW DEBEZIUM ORACLE CONNECTORS HANDLE FAULTS AND PROBLEMS 327

CHAPTER 7. DEBEZIUM CONNECTOR FOR POSTGRESQL 330
7.1. OVERVIEW OF DEBEZIUM POSTGRESQL CONNECTOR 330
7.2. HOW DEBEZIUM POSTGRESQL CONNECTORS WORK 331
7.2.1. Security for PostgreSQL connector 332
7.2.2. How Debezium PostgreSQL connectors perform database snapshots 332
7.2.2.1. Ad hoc snapshots 333
7.2.2.2. Incremental snapshots 334
7.2.3. How Debezium PostgreSQL connectors stream change event records 338
7.2.4. Default names of Kafka topics that receive Debezium PostgreSQL change event records 339
7.2.5. Metadata in Debezium PostgreSQL change event records 340
7.2.6. Debezium PostgreSQL connector-generated events that represent transaction boundaries 341
7.3. DESCRIPTIONS OF DEBEZIUM POSTGRESQL CONNECTOR DATA CHANGE EVENTS 342
7.3.1. About keys in Debezium PostgreSQL change events 344
7.3.2. About values in Debezium PostgreSQL change events 346
7.4. HOW DEBEZIUM POSTGRESQL CONNECTORS MAP DATA TYPES 360

7.5. SETTING UP POSTGRESQL TO RUN A DEBEZIUM CONNECTOR 372
7.5.1. Configuring a replication slot for the Debezium pgoutput plug-in 373
7.5.2. Setting up PostgreSQL permissions for the Debezium connector 373
7.5.3. Setting privileges to enable Debezium to create PostgreSQL publications 374
7.5.4. Configuring PostgreSQL to allow replication with the Debezium connector host 375
7.5.5. Configuring PostgreSQL to manage Debezium WAL disk space consumption 375
7.6. DEPLOYMENT OF DEBEZIUM POSTGRESQL CONNECTORS 376
7.6.1. PostgreSQL connector deployment using AMQ Streams 376
7.6.2. Using AMQ Streams to deploy a Debezium PostgreSQL connector 377
7.6.3. Deploying a Debezium PostgreSQL connector by building a custom Kafka Connect container image from a Dockerfile 382
7.6.4. Verifying that the Debezium PostgreSQL connector is running 386
7.6.5. Descriptions of Debezium PostgreSQL connector configuration properties 390
7.7. MONITORING DEBEZIUM POSTGRESQL CONNECTOR PERFORMANCE 416
7.7.1. Monitoring Debezium during snapshots of PostgreSQL databases 416
7.7.2. Monitoring Debezium PostgreSQL connector record streaming 418
7.8. HOW DEBEZIUM POSTGRESQL CONNECTORS HANDLE FAULTS AND PROBLEMS 420

CHAPTER 8. DEBEZIUM CONNECTOR FOR SQL SERVER 423
8.1. OVERVIEW OF DEBEZIUM SQL SERVER CONNECTOR 423
8.2. HOW DEBEZIUM SQL SERVER CONNECTORS WORK 424
8.2.1. How Debezium SQL Server connectors perform database snapshots 424
8.2.1.1. Ad hoc snapshots 425
8.2.1.2. Incremental snapshots 426
8.2.2. How Debezium SQL Server connectors read change data tables 430
8.2.3. Limitations of Debezium SQL Server connector 430
8.2.4. Default names of Kafka topics that receive Debezium SQL Server change event records 430
8.2.5. How the Debezium SQL Server connector uses the schema change topic 431
8.2.6. Descriptions of Debezium SQL Server connector data change events 435
8.2.6.1. About keys in Debezium SQL Server change events 437
8.2.6.2. About values in Debezium SQL Server change events 438
8.2.7. Debezium SQL Server connector-generated events that represent transaction boundaries 447
8.2.7.1. Change data event enrichment 449
8.2.8. How Debezium SQL Server connectors map data types 449
8.3. SETTING UP SQL SERVER TO RUN A DEBEZIUM CONNECTOR 455
8.3.1. Enabling CDC on the SQL Server database 455
8.3.2. Enabling CDC on a SQL Server table 456
8.3.3. Verifying that the user has access to the CDC table 457
8.3.4. SQL Server on Azure 457
8.3.5. Effect of SQL Server capture job agent configuration on server load and latency 458
8.3.6. SQL Server capture job agent configuration parameters 458
8.4. DEPLOYMENT OF DEBEZIUM SQL SERVER CONNECTORS 459
8.4.1. SQL Server connector deployment using AMQ Streams 459
8.4.2. Using AMQ Streams to deploy a Debezium SQL Server connector 460
8.4.3. Deploying a Debezium SQL Server connector by building a custom Kafka Connect container image from a Dockerfile 465
8.4.4. Descriptions of Debezium SQL Server connector configuration properties 473
8.5. REFRESHING CAPTURE TABLES AFTER A SCHEMA CHANGE 493
8.5.1. Running an offline update after a schema change 494
8.5.2. Running an online update after a schema change 495
8.6. MONITORING DEBEZIUM SQL SERVER CONNECTOR PERFORMANCE 497
8.6.1. Debezium SQL Server connector snapshot metrics 497
8.6.2. Debezium SQL Server connector streaming metrics 499

8.6.3. Debezium SQL Server connector schema history metrics 501

CHAPTER 9. MONITORING DEBEZIUM 503
9.1. METRICS FOR MONITORING DEBEZIUM CONNECTORS 503
9.2. ENABLING JMX IN LOCAL INSTALLATIONS 503
9.2.1. Zookeeper JMX environment variables 503
9.2.2. Kafka JMX environment variables 504
9.2.3. Kafka Connect JMX environment variables 504
9.3. MONITORING DEBEZIUM ON OPENSHIFT 504

CHAPTER 10. DEBEZIUM LOGGING 505
10.1. DEBEZIUM LOGGING CONCEPTS 505
10.2. DEFAULT DEBEZIUM LOGGING CONFIGURATION 505
10.3. CONFIGURING DEBEZIUM LOGGING 506
10.3.1. Changing the Debezium logging level by configuring loggers 506
10.3.2. Dynamically changing the Debezium logging level with the Kafka Connect API 508
10.3.3. Changing the Debezium logging levely by adding mapped diagnostic contexts 508
10.4. DEBEZIUM LOGGING ON OPENSHIFT 509

CHAPTER 11. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION 510
11.1. CUSTOMIZATION OF KAFKA CONNECT AUTOMATIC TOPIC CREATION 510
11.1.1. Disabling automatic topic creation for the Kafka broker 511
11.1.2. Configuring automatic topic creation in Kafka Connect 511
11.1.3. Configuration of automatically created topics 512
11.1.3.1. Topic creation groups 512
11.1.3.2. Topic creation group configuration properties 512
11.1.3.3. Specifying the configuration for the Debezium default topic creation group 513
11.1.3.4. Specifying the configuration for Debezium custom topic creation groups 514
11.1.3.5. Registering Debezium custom topic creation groups 515
11.2. CONFIGURING DEBEZIUM CONNECTORS TO USE AVRO SERIALIZATION 517
11.2.1. About the Apicurio Registry 517
11.2.2. Overview of deploying a Debezium connector that uses Avro serialization 518
11.2.3. Deploying connectors that use Avro in Debezium containers 518
11.2.4. About Avro name requirements 522
11.3. EMITTING DEBEZIUM CHANGE EVENT RECORDS IN CLOUDEVENTS FORMAT 522
11.3.1. Example Debezium change event records in CloudEvents format 523
11.3.2. Example of configuring Debezium CloudEvents converter 525
11.3.3. Debezium CloudEvents converter configuration options 526
11.4. SENDING SIGNALS TO A DEBEZIUM CONNECTOR 526
11.4.1. Enabling Debezium signaling 527
11.4.1.1. Required structure of a Debezium signaling data collection 528
11.4.1.2. Creating a Debezium signaling data collection 528
11.4.2. Types of Debezium signal actions 529
11.4.2.1. Logging signals 529
11.4.2.2. Ad hoc snapshot signals 530
11.4.2.3. Incremental snapshots 530

CHAPTER 12. APPLYING TRANSFORMATIONS TO MODIFY MESSAGES EXCHANGED WITH APACHE KAFKA 532
12.1. APPLYING TRANSFORMATIONS SELECTIVELY WITH SMT PREDICATES 532
12.1.1. About SMT predicates 532
12.1.2. Defining SMT predicates 534
12.1.3. Ignoring tombstone events 535
12.2. ROUTING DEBEZIUM EVENT RECORDS TO TOPICS THAT YOU SPECIFY 536

12.2.1. Use case for routing Debezium records to topics that you specify 537
12.2.2. Example of routing Debezium records for multiple tables to one topic 537
12.2.3. Ensuring unique keys across Debezium records routed to the same topic 538
12.2.4. Options for applying the topic routing transformation selectively 539
12.2.5. Options for configuring Debezium topic routing transformation 539
12.3. ROUTING CHANGE EVENT RECORDS TO TOPICS ACCORDING TO EVENT CONTENT 540
12.3.1. Setting up the Debezium content-based-routing SMT 541
12.3.2. Example: Debezium basic content-based routing configuration 542
12.3.3. Variables for use in Debezium content-based routing expressions 542
12.3.4. Options for applying the content-based routing transformation selectively 543
12.3.5. Configuration of content-based routing conditions for other scripting languages 544
12.3.6. Options for configuring the content-based routing transformation 544
12.4. FILTERING DEBEZIUM CHANGE EVENT RECORDS 545
12.4.1. Setting up the Debezium filter SMT 546
12.4.2. Example: Debezium basic filter SMT configuration 546
12.4.3. Variables for use in filter expressions 547
12.4.4. Options for applying the filter transformation selectively 548
12.4.5. Filter condition configuration for other scripting languages 548
12.4.6. Options for configuring filter transformation 549
12.5. EXTRACTING SOURCE RECORD AFTER STATE FROM DEBEZIUM CHANGE EVENTS 550
12.5.1. Description of Debezium change event structure 550
12.5.2. Behavior of Debezium event flattening transformation 551
12.5.3. Configuration of Debezium event flattening transformation 552
12.5.4. Example of adding Debezium metadata to the Kafka record 553
12.5.5. Options for applying the event flattening transformation selectively 553
12.5.6. Options for configuring Debezium event flattening transformation 554
12.6. CONFIGURING DEBEZIUM CONNECTORS TO USE THE OUTBOX PATTERN 557
12.6.1. Example of a Debezium outbox message 558
12.6.2. Outbox table structure expected by Debezium outbox event router SMT 559
12.6.3. Basic Debezium outbox event router SMT configuration 560
12.6.4. Options for applying the Outbox event router transformation selectively 561
12.6.5. Using Avro as the payload format in Debezium outbox messages 561
12.6.6. Emitting additional fields in Debezium outbox messages 562
12.6.7. Expanding escaped JSON String as JSON 562
12.6.8. Options for configuring outbox event router transformation 563
12.7. CONFIGURING DEBEZIUM MONGODB CONNECTORS TO USE THE OUTBOX PATTERN 566
12.7.1. Example of a Debezium MongoDB outbox message 567
12.7.2. Outbox collection structure expected by Debezium mongodb outbox event router SMT 568
12.7.3. Basic Debezium MongoDB outbox event router SMT configuration 569
12.7.4. Options for applying the MongoDB outbox event router transformation selectively 570
12.7.5. Using Avro as the payload format in Debezium MongoDB outbox messages 570
12.7.6. Emitting additional fields in Debezium MongoDB outbox messages 571
12.7.7. Expanding escaped JSON String as JSON 571
12.7.8. Options for configuring outbox event router transformation 572

PREFACE

Debezium is a set of distributed services that capture row-level changes in your databases so that your applications can see and respond to those changes. Debezium records all row-level changes committed to each database table. Each application reads the transaction logs of interest to view all operations in the order in which they occurred.

This guide provides details about using the following Debezium topics:

- Chapter 1, *High level overview of Debezium*
- Chapter 2, *Required custom resource upgrades*
- Chapter 3, *Debezium connector for Db2*
- Chapter 4, *Debezium connector for MongoDB*
- Chapter 5, *Debezium connector for MySQL*
- Chapter 6, *Debezium Connector for Oracle*
- Chapter 7, *Debezium connector for PostgreSQL*
- Chapter 8, *Debezium connector for SQL Server*
- Chapter 9, *Monitoring Debezium*
- Chapter 10, *Debezium logging*
- Chapter 11, *Configuring Debezium connectors for your application*
- Chapter 12, *Applying transformations to modify messages exchanged with Apache Kafka*

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. HIGH LEVEL OVERVIEW OF DEBEZIUM

Debezium is a set of distributed services that capture changes in your databases. Your applications can consume and respond to those changes. Debezium captures each row-level change in each database table in a change event record and streams these records to Kafka topics. Applications read these streams, which provide the change event records in the same order in which they were generated.

More details are in the following sections:

- Section 1.1, “Debezium Features”
- Section 1.2, “Description of Debezium architecture”

1.1. DEBEZIUM FEATURES

Debezium is a set of source connectors for Apache Kafka Connect. Each connector ingests changes from a different database by using that database’s features for change data capture (CDC). Unlike other approaches, such as polling or dual writes, log-based CDC as implemented by Debezium:

- Ensures that **all data changes are captured**.
- Produces change events with a **very low delay** while avoiding increased CPU usage required for frequent polling. For example, for MySQL or PostgreSQL, the delay is in the millisecond range.
- Requires **no changes to your data model** such as a "Last Updated" column.
- Can **capture deletes**.
- Can **capture old record state and additional metadata** such as transaction ID and causing query, depending on the database’s capabilities and configuration.

Five Advantages of Log-Based Change Data Capture is a blog post that provides more details.

Debezium connectors capture data changes with a range of related capabilities and options:

- **Snapshots:** optionally, an initial snapshot of a database’s current state can be taken if a connector is started and not all logs still exist. Typically, this is the case when the database has been running for some time and has discarded transaction logs that are no longer needed for transaction recovery or replication. There are different modes for performing snapshots, including support for *incremental* snapshots, which can be triggered at connector runtime. For more details, see the documentation for the connector that you are using.
- **Filters:** you can configure the set of captured schemas, tables and columns with include/exclude list filters.
- **Masking:** the values from specific columns can be masked, for example, when they contain sensitive data.
- **Monitoring:** most connectors can be monitored by using JMX.

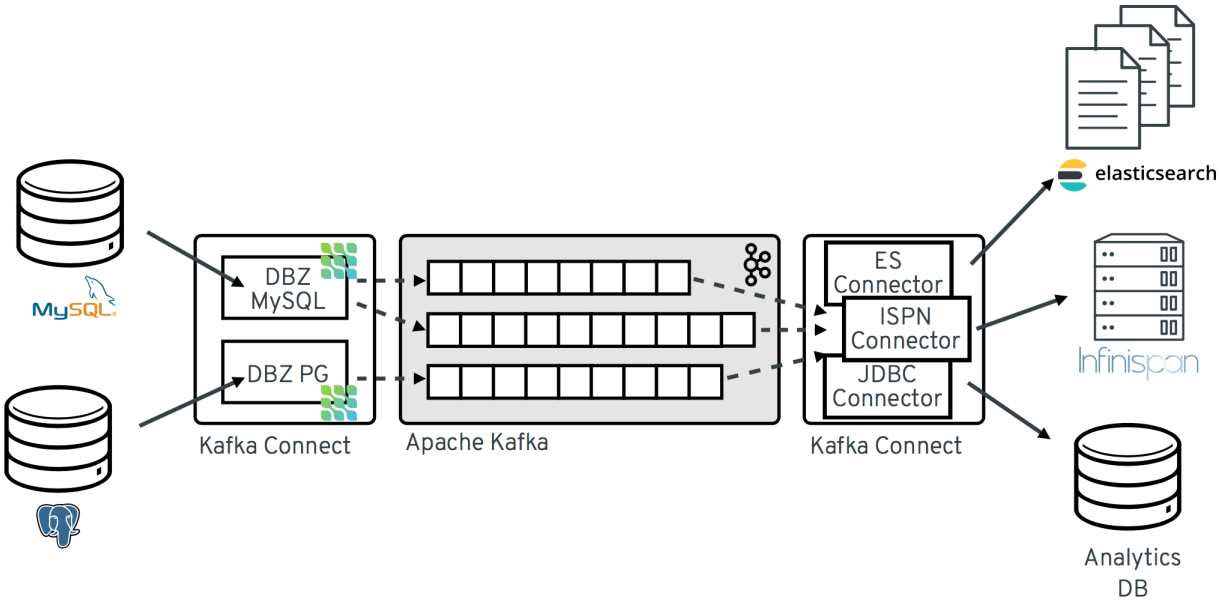
1.2. DESCRIPTION OF DEBEZIUM ARCHITECTURE

You deploy Debezium by means of Apache Kafka Connect. Kafka Connect is a framework and runtime for implementing and operating:

- Source connectors such as Debezium that send records into Kafka

- Sink connectors that propagate records from Kafka topics to other systems

The following image shows the architecture of a change data capture pipeline based on Debezium:



As shown in the image, the Debezium connectors for MySQL and PostgresSQL are deployed to capture changes to these two types of databases. Each Debezium connector establishes a connection to its source database:

- The MySQL connector uses a client library for accessing the **binlog**.
- The PostgreSQL connector reads from a logical replication stream.

Kafka Connect operates as a separate service besides the Kafka broker.

By default, changes from one database table are written to a Kafka topic whose name corresponds to the table name. If needed, you can adjust the destination topic name by configuring Debezium's topic routing transformation. For example, you can:

- Route records to a topic whose name is different from the table's name
- Stream change event records for multiple tables into a single topic

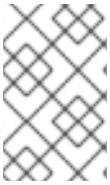
After change event records are in Apache Kafka, different connectors in the Kafka Connect eco-system can stream the records to other systems and databases such as Elasticsearch, data warehouses and analytics systems, or caches such as Infinispan. Depending on the chosen sink connector, you might need to configure Debezium's new record state extraction transformation. This Kafka Connect SMT propagates the **after** structure from Debezium's change event to the sink connector. This is in place of the verbose change event record that is propagated by default.

CHAPTER 2. REQUIRED CUSTOM RESOURCE UPGRADES

Debezium is a Kafka connector plugin that is deployed to an Apache Kafka cluster that runs on AMQ Streams on OpenShift. To prepare for OpenShift CRD **v1**, in the current version of AMQ Streams the required version of the custom resource definitions (CRD) API is now set to **v1beta2**. The **v1beta2** version of the API replaces the previously supported **v1beta1** and **v1alpha1** API versions. Support for the **v1alpha1** and **v1beta1** API versions is now deprecated in AMQ Streams. Those earlier versions are now removed from most AMQ Streams custom resources, including the KafkaConnect and KafkaConnector resources that you use to configure Debezium connectors.

The CRDs that are based on the **v1beta2** API version use the OpenAPI structural schema. Custom resources based on the superseded v1alpha1 or v1beta1 APIs do not support structural schemas, and are incompatible with the current version of AMQ Streams. Before you upgrade to AMQ Streams2.2, you must upgrade existing custom resources to use API version **kafka.strimzi.io/v1beta2**. You can upgrade custom resources any time after you upgrade to AMQ Streams 1.7. You must complete the upgrade to the v1beta2 API before you upgrade to AMQ Streams2.2 or newer.

To facilitate the upgrade of CRDs and custom resources, AMQ Streams provides an API conversion tool that automatically upgrades them to a format that is compatible with **v1beta2**. For more information about the tool and for the complete instructions about how to upgrade AMQ Streams, see Deploying and Upgrading AMQ Streams on OpenShift.



NOTE

The requirement to update custom resources applies only to Debezium deployments that run on AMQ Streams on OpenShift. The requirement does not apply to Debezium on Red Hat Enterprise Linux

CHAPTER 3. DEBEZIUM CONNECTOR FOR DB2

Debezium's Db2 connector can capture row-level changes in the tables of a Db2 database. For information about the Db2 Database versions that are compatible with this connector, see the Debezium Supported Configurations page.

This connector is strongly inspired by the Debezium implementation of SQL Server, which uses a SQL-based polling model that puts tables into "capture mode". When a table is in capture mode, the Debezium Db2 connector generates and streams a change event for each row-level update to that table.

A table that is in capture mode has an associated change-data table, which Db2 creates. For each change to a table that is in capture mode, Db2 adds data about that change to the table's associated change-data table. A change-data table contains an entry for each state of a row. It also has special entries for deletions. The Debezium Db2 connector reads change events from change-data tables and emits the events to Kafka topics.

The first time a Debezium Db2 connector connects to a Db2 database, the connector reads a consistent snapshot of the tables for which the connector is configured to capture changes. By default, this is all non-system tables. There are connector configuration properties that let you specify which tables to put into capture mode, or which tables to exclude from capture mode.

When the snapshot is complete the connector begins emitting change events for committed updates to tables that are in capture mode. By default, change events for a particular table go to a Kafka topic that has the same name as the table. Applications and services consume change events from these topics.

NOTE

The connector requires the use of the abstract syntax notation (ASN) libraries, which are available as a standard part of Db2 for Linux. To use the ASN libraries, you must have a license for IBM InfoSphere Data Replication (IIDR). You do not have to install IIDR to use the ASN libraries.

Information and procedures for using a Debezium Db2 connector is organized as follows:

- Section 3.1, "Overview of Debezium Db2 connector"
- Section 3.2, "How Debezium Db2 connectors work"
- Section 3.3, "Descriptions of Debezium Db2 connector data change events"
- Section 3.4, "How Debezium Db2 connectors map data types"
- Section 3.5, "Setting up Db2 to run a Debezium connector"
- Section 3.6, "Deployment of Debezium Db2 connectors"
- Section 3.7, "Monitoring Debezium Db2 connector performance"
- Section 3.8, "Managing Debezium Db2 connectors"
- Section 3.9, "Updating schemas for Db2 tables in capture mode for Debezium connectors"

3.1. OVERVIEW OF DEBEZIUM DB2 CONNECTOR

The Debezium Db2 connector is based on the ASN Capture/Apply agents that enable SQL Replication in Db2. A capture agent:

- Generates change-data tables for tables that are in capture mode.
- Monitors tables in capture mode and stores change events for updates to those tables in their corresponding change-data tables.

The Debezium connector uses a SQL interface to query change-data tables for change events.

The database administrator must put the tables for which you want to capture changes into capture mode. For convenience and for automating testing, there are Debezium user-defined functions (UDFs) in C that you can compile and then use to do the following management tasks:

- Start, stop, and reinitialize the ASN agent
- Put tables into capture mode
- Create the replication (ASN) schemas and change-data tables
- Remove tables from capture mode

Alternatively, you can use Db2 control commands to accomplish these tasks.

After the tables of interest are in capture mode, the connector reads their corresponding change-data tables to obtain change events for table updates. The connector emits a change event for each row-level insert, update, and delete operation to a Kafka topic that has the same name as the changed table. This is default behavior that you can modify. Client applications read the Kafka topics that correspond to the database tables of interest and can react to each row-level change event.

Typically, the database administrator puts a table into capture mode in the middle of the life of a table. This means that the connector does not have the complete history of all changes that have been made to the table. Therefore, when the Db2 connector first connects to a particular Db2 database, it starts by performing a *consistent snapshot* of each table that is in capture mode. After the connector completes the snapshot, the connector streams change events from the point at which the snapshot was made. In this way, the connector starts with a consistent view of the tables that are in capture mode, and does not drop any changes that were made while it was performing the snapshot.

Debezium connectors are tolerant of failures. As the connector reads and produces change events, it records the log sequence number (LSN) of the change-data table entry. The LSN is the position of the change event in the database log. If the connector stops for any reason, including communication failures, network problems, or crashes, upon restarting it continues reading the change-data tables where it left off. This includes snapshots. That is, if the snapshot was not complete when the connector stopped, upon restart the connector begins a new snapshot.

3.2. HOW DEBEZIUM DB2 CONNECTORS WORK

To optimally configure and run a Debezium Db2 connector, it is helpful to understand how the connector performs snapshots, streams change events, determines Kafka topic names, and handles schema changes.

Details are in the following topics:

- Section 3.2.1, “How Debezium Db2 connectors perform database snapshots”
- Section 3.2.2, “How Debezium Db2 connectors read change-data tables”

- Section 3.2.3, “Default names of Kafka topics that receive Debezium Db2 change event records”
- Section 3.2.4, “About the Debezium Db2 connector schema change topic”
- Section 3.2.5, “Debezium Db2 connector-generated events that represent transaction boundaries”

3.2.1. How Debezium Db2 connectors perform database snapshots

Db2`s replication feature is not designed to store the complete history of database changes. Consequently, when a Debezium Db2 connector connects to a database for the first time, it takes a consistent snapshot of tables that are in capture mode and streams this state to Kafka. This establishes the baseline for table content.

By default, when a Db2 connector performs a snapshot, it does the following:

1. Determines which tables are in capture mode, and thus must be included in the snapshot. By default, all non-system tables are in capture mode. Connector configuration properties, such as **table.exclude.list** and **table.include.list** let you specify which tables should be in capture mode.
2. Obtains a lock on each of the tables in capture mode. This ensures that no schema changes can occur in those tables during the snapshot. The level of the lock is determined by the **snapshot.isolation.mode** connector configuration property.
3. Reads the highest (most recent) LSN position in the server’s transaction log.
4. Captures the schema of all tables that are in capture mode. The connector persists this information in its internal database history topic.
5. Optional, releases the locks obtained in step 2. Typically, these locks are held for only a short time.
6. At the LSN position read in step 3, the connector scans the capture mode tables as well as their schemas. During the scan, the connector:
    a. Confirms that the table was created before the start of the snapshot. If it was not, the snapshot skips that table. After the snapshot is complete, and the connector starts emitting change events, the connector produces change events for any tables that were created during the snapshot.
    b. Produces a *read* event for each row in each table that is in capture mode. All *read* events contain the same LSN position, which is the LSN position that was obtained in step 3.
    c. Emits each *read* event to the Kafka topic that has the same name as the table.
7. Records the successful completion of the snapshot in the connector offsets.

3.2.1.1. Ad hoc snapshots

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing table data, Debezium

includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of tables.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling tables. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the tables to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the tables in the database.

You specify the tables to capture by sending an **execute-snapshot** message to the signaling table. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the tables to include in the snapshot, as described in the following table:

Table 3.1. Example of an ad hoc`execute-snapshot` signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.
data-collections	*N/A*	An array that contains the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling table. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each table. Based on the number of entries in the table, and the configured chunk size, Debezium divides the table into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

3.2.1.2. Incremental snapshots

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each table in phases, in a series of configurable chunks. You can specify the tables that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each table row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the other.
- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the table from the beginning.
- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a table to its **table.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each table by primary key and then splits the table into chunks based on the configured chunk size. Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying table records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a table row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same table row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified table chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and

the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the table's Kafka topic.

The connector repeats the process for each snapshot chunk.

Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling table on the source database. You submit signals to the table as SQL **INSERT** queries. After Debezium detects the change in the signaling table, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the tables to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.

NOTE

If the name of a table that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the table to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling table.

The following table describes the these parameters:

Table 3.2. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling table on the source database
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of table names to include in the snapshot. The array lists tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
```

```
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

WARNING

The Debezium connector for Db2 does not support schema changes while an incremental snapshot is running.

3.2.2. How Debezium Db2 connectors read change-data tables

After a complete snapshot, when a Debezium Db2 connector starts for the first time, the connector identifies the change-data table for each source table that is in capture mode. The connector does the following for each change-data table:

1. Reads change events that were created between the last stored, highest LSN and the current, highest LSN.
2. Orders the change events according to the commit LSN and the change LSN for each event. This ensures that the connector emits the change events in the order in which the table changes occurred.
3. Passes commit and change LSNs as offsets to Kafka Connect.
4. Stores the highest LSN that the connector passed to Kafka Connect.

After a restart, the connector resumes emitting change events from the offset (commit and change LSNs) where it left off. While the connector is running and emitting change events, if you remove a table from capture mode or add a table to capture mode, the connector detects the change, and modifies its behavior accordingly.

3.2.3. Default names of Kafka topics that receive Debezium Db2 change event records

By default, the Db2 connector writes change events for all of the **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table. The connector uses the following convention to name change event topics:

databaseName.schemaName.tableName

The following list provides definitions for the components of the default name:

databaseName

The logical name of the connector as specified by the **database.server.name** connector configuration property.

schemaName

The name of the schema in which the operation occurred.

tableName

The name of the table in which the operation occurred.

For example, consider a Db2 installation with the **mydatabase** database, which contains four tables: **PRODUCTS**, **PRODUCTS_ON_HAND**, **CUSTOMERS**, and **ORDERS** that are in the **MYSCHEMA** schema. The connector would emit events to these four Kafka topics:

- **mydatabase.MYSCHEMA.PRODUCTS**
- **mydatabase.MYSCHEMA.PRODUCTS_ON_HAND**
- **mydatabase.MYSCHEMA.CUSTOMERS**
- **mydatabase.MYSCHEMA.ORDERS**

The connector applies similar naming conventions to label its internal database history topics, schema change topics, and transaction metadata topics.

If the default topic name do not meet your requirements, you can configure custom topic names. To configure custom topic names, you specify regular expressions in the logical topic routing SMT. For more information about using the logical topic routing SMT to customize topic naming, see Topic routing.

3.2.4. About the Debezium Db2 connector schema change topic

You can configure a Debezium Db2 connector to produce schema change events that describe schema changes that are applied to captured tables in the database.

Debezium emits a message to the schema change topic when:

- A new table goes into capture mode.
- A table is removed from capture mode.
- During a database schema update, there is a change in the schema for a table that is in capture mode.

The connector writes schema change events to a Kafka schema change topic that has the name ***<serverName>*** where ***<serverName>*** is the logical server name that is specified in the

database.server.name connector configuration property. Messages that the connector sends to the schema change topic contain a payload that includes the following elements:

databaseName

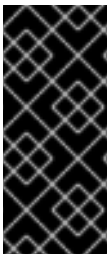
The name of the database to which the statements are applied. The value of **databaseName** serves as the message key.

pos

The position in the binlog where the statements appear.

tableChanges

A structured representation of the entire table schema after the schema change. The **tableChanges** field contains an array that includes entries for each column of the table. Because the structured representation presents data in JSON or Avro format, consumers can easily read messages without first processing them through a DDL parser.



IMPORTANT

For a table that is in capture mode, the connector not only stores the history of schema changes in the schema change topic, but also in an internal database history topic. The internal database history topic is for connector use only and it is not intended for direct use by consuming applications. Ensure that applications that require notifications about schema changes consume that information only from the schema change topic.

IMPORTANT

Never partition the database history topic. For the database history topic to function correctly, it must maintain a consistent, global order of the event records that the connector emits to it.

To ensure that the topic is not split among partitions, set the partition count for the topic by using one of the following methods:

- If you create the database history topic manually, specify a partition count of **1**.
- If you use the Apache Kafka broker to create the database history topic automatically, the topic is created, set the value of the Kafka **num.partitions** configuration option to **1**.

WARNING

The format of messages that a connector emits to its schema change topic is in an incubating state and can change without notice.

Example: Message emitted to the Db2 connector schema change topic

The following example shows a message in the schema change topic. The message contains a logical representation of the table schema.

```
{
  "schema": {
```

```
...
},
"payload": {
  "source": {
    "version": "1.9.7.Final",
    "connector": "db2",
    "name": "db2",
    "ts_ms": 1588252618953,
    "snapshot": "true",
    "db": "testdb",
    "schema": "DB2INST1",
    "table": "CUSTOMERS",
    "change_lsn": null,
    "commit_lsn": "00000025:00000d98:00a2",
    "event_serial_no": null
  },
  "databaseName": "TESTDB", (1)
  "schemaName": "DB2INST1",
  "ddl": null, (2)
  "tableChanges": [ (3)
    {
      "type": "CREATE", (4)
      "id": "\"DB2INST1\".\"CUSTOMERS\"", (5)
      "table": { (6)
        "defaultCharsetName": null,
        "primaryKeyColumnNames": [ (7)
          "ID"
        ],
        "columns": [ (8)
          {
            "name": "ID",
            "jdbcType": 4,
            "nativeType": null,
            "typeName": "int identity",
            "typeExpression": "int identity",
            "charsetName": null,
            "length": 10,
            "scale": 0,
            "position": 1,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "FIRST_NAME",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "varchar",
            "typeExpression": "varchar",
            "charsetName": null,
            "length": 255,
            "scale": null,
            "position": 2,
            "optional": false,
            "autoIncremented": false,
```

```
            "generated": false
          },
          {
            "name": "LAST_NAME",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "varchar",
            "typeExpression": "varchar",
            "charsetName": null,
            "length": 255,
            "scale": null,
            "position": 3,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "EMAIL",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "varchar",
            "typeExpression": "varchar",
            "charsetName": null,
            "length": 255,
            "scale": null,
            "position": 4,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          }
        ]
      }
    }
  ]
}
}
```


Table 3.3. Descriptions of fields in messages emitted to the schema change topic

Item	Field name	Description
1	**databaseName** **schemaName**	Identifies the database and the schema that contain the change.
2	**ddl**	Always **null** for the Db2 connector. For other connectors, this field contains the DDL responsible for the schema change. This DDL is not available to Db2 connectors.
3	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.

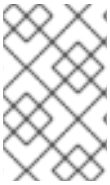
Item	Field name	Description
4	**type**	Describes the kind of change. The value is one of the following: • **CREATE** - table created • **ALTER** - table modified • **DROP** - table deleted
5	**id**	Full identifier of the table that was created, altered, or dropped.
6	**table**	Represents table metadata after the applied change.
7	**primaryKeyColumnNames**	List of columns that compose the table's primary key.
8	**columns**	Metadata for each column in the changed table.

In messages that the connector sends to the schema change topic, the message key is the name of the database that contains the schema change. In the following example, the **payload** field contains the key:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "databaseName"
      }
    ],
    "optional": false,
    "name": "io.debezium.connector.db2.SchemaChangeKey"
  },
  "payload": {
    "databaseName": "TESTDB"
  }
}
```

3.2.5. Debezium Db2 connector-generated events that represent transaction boundaries

Debezium can generate events that represent transaction boundaries and that enrich change data event messages.



LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

Debezium generates transaction boundary events for the **BEGIN** and **END** delimiters in every transaction. Transaction boundary events contain the following fields:

status

BEGIN or **END**.

id

String representation of the unique transaction identifier.

event_count (for **END** events)

Total number of events emitted by the transaction.

data_collections (for **END** events)

An array of pairs of **data_collection** and **event_count** elements. that indicates the number of events that the connector emits for changes that originate from a data collection.

Example

```
{
  "status": "BEGIN",
  "id": "00000025:00000d08:0025",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "00000025:00000d08:0025",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "testDB.dbo.tablea",
      "event_count": 1
    },
    {
      "data_collection": "testDB.dbo.tableb",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, the connector emits transaction events to the ***<database.server.name>*.transaction** topic.

Data change event enrichment

When transaction metadata is enabled the connector enriches the change event **Envelope** with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

id

String representation of unique transaction identifier.

total_order

The absolute position of the event among all events generated by the transaction.

data_collection_order

The per-data collection position of the event among all events that were emitted by the transaction.

Following is an example of a message:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "00000025:00000d08:0025",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

3.3. DESCRIPTIONS OF DEBEZIUM DB2 CONNECTOR DATA CHANGE EVENTS

The Debezium Db2 connector generates a data change event for each row-level **INSERT**, **UPDATE**, and **DELETE** operation. Each event contains a key and a value. The structure of the key and the value depends on the table that was changed.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events may change over time, which can be difficult for consumers to handle. To address this, each event contains the schema for its content or, if you are using a schema registry, a schema ID that a consumer can use to obtain the schema from the registry. This makes each event self-contained.

The following skeleton JSON shows the basic four parts of a change event. However, how you configure the Kafka Connect converter that you choose to use in your application determines the representation of these four parts in change events. A **schema** field is in a change event only when you configure the converter to produce it. Likewise, the event key and event payload are in a change event only if you configure a converter to produce it. If you use the JSON converter and you configure it to produce all four basic change event parts, change events have this structure:

```
{
 "schema": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

Table 3.4. Overview of change event basic content

Item	Field name	Description
1	**schema**	The first **schema** field is part of the event key. It specifies a Kafka Connect schema that describes what is in the event key's **payload** portion. In other words, the first **schema** field describes the structure of the primary key, or the unique key if the table does not have a primary key, for the table that was changed. It is possible to override the table's primary key by setting the **message.key.columns** connector configuration property. In this case, the first schema field describes the structure of the key identified by that property.
2	**payload**	The first **payload** field is part of the event key. It has the structure described by the previous **schema** field and it contains the key for the row that was changed.
3	**schema**	The second **schema** field is part of the event value. It specifies the Kafka Connect schema that describes what is in the event value's **payload** portion. In other words, the second **schema** describes the structure of the row that was changed. Typically, this schema contains nested schemas.
4	**payload**	The second **payload** field is part of the event value. It has the structure described by the previous **schema** field and it contains the actual data for the row that was changed.

By default, the connector streams change event records to topics with names that are the same as the event's originating table. See topic names.

WARNING

The Debezium Db2 connector ensures that all Kafka Connect schema names adhere to the Avro schema name format . This means that the logical server name must start with a Latin letter or an underscore, that is, a-z, A-Z, or _. Each remaining character in the logical server name and each character in the database and table names must be a Latin letter, a digit, or an underscore, that is, a-z, A-Z, 0-9, or _. If there is an invalid character it is replaced with an underscore character.

This can lead to unexpected conflicts if the logical server name, a database name, or a table name contains invalid characters, and the only characters that distinguish names from one another are invalid and thus replaced with underscores.

Also, Db2 names for databases, schemas, and tables can be case sensitive. This means that the connector could emit event records for more than one table to the same Kafka topic.

Details are in the following topics:

- Section 3.3.1, “About keys in Debezium db2 change events”
- Section 3.3.2, “About values in Debezium Db2 change events”

3.3.1. About keys in Debezium db2 change events

A change event’s key contains the schema for the changed table’s key and the changed row’s actual key. Both the schema and its corresponding payload contain a field for each column in the changed table’s **PRIMARY KEY** (or unique constraint) at the time the connector created the event.

Consider the following **customers** table, which is followed by an example of a change event key for this table.

Example table

```
CREATE TABLE customers (
 ID INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
 FIRST_NAME VARCHAR(255) NOT NULL,
 LAST_NAME VARCHAR(255) NOT NULL,
 EMAIL VARCHAR(255) NOT NULL UNIQUE
);
```

Example change event key

Every change event that captures a change to the **customers** table has the same event key schema. For as long as the **customers** table has the previous definition, every change event that captures a change to the **customers** table has the following key structure. In JSON, it looks like this:

```
{
   "schema": { 1
      "type": "struct",
      "fields": [ 2
         {
            "type": "int32",
            "optional": false,
            "field": "ID"
         }
      ],
      "optional": false, 3
      "name": "mydatabase.MYSCHEMA.CUSTOMERS.Key" 4
   },
   "payload": { 5
      "ID": 1004
   }
}
```

Table 3.5. Description of change event key

Item	Field name	Description

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's **payload** portion.
2	**fields**	Specifies each field that is expected in the **payload**, including each field's name, type, and whether it is required.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a table does not have a primary key.
4	**mydatabase.MYSCHEMA.CUSTOMERS.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format *connector-name.database-name.table-name*.**Key**. In this example: • **mydatabase** is the name of the connector that generated this event. • **MYSCHEMA** is the database schema that contains the table that was changed. • **CUSTOMERS** is the table that was updated.
5	**payload**	Contains the key for the row for which this change event was generated. In this example, the key, contains a single **ID** field whose value is **1004**.

3.3.2. About values in Debezium Db2 change events

The value in a change event is a bit more complicated than the key. Like the key, the value has a **schema** section and a **payload** section. The **schema** section contains the schema that describes the **Envelope** structure of the **payload** section, including its nested fields. Change events for operations that create, update or delete data all have a value payload with an envelope structure.

Consider the same sample table that was used to show an example of a change event key:

Example table

```
CREATE TABLE customers (
 ID INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
 FIRST_NAME VARCHAR(255) NOT NULL,
 LAST_NAME VARCHAR(255) NOT NULL,
 EMAIL VARCHAR(255) NOT NULL UNIQUE
);
```

The event value portion of every change event for the **customers** table specifies the same schema. The event value's payload varies according to the event type:

- *create* events
- *update* events

- *delete* events

create events

The following example shows the value portion of a change event that the connector generates for an operation that creates data in the **customers** table:

```
{
  "schema": {  1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "ID"
          },
          {
            "type": "string",
            "optional": false,
            "field": "FIRST_NAME"
          },
          {
            "type": "string",
            "optional": false,
            "field": "LAST_NAME"
          },
          {
            "type": "string",
            "optional": false,
            "field": "EMAIL"
          }
        ],
        "optional": true,
        "name": "mydatabase.MYSCHEMA.CUSTOMERS.Value",  2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "ID"
          },
          {
            "type": "string",
            "optional": false,
            "field": "FIRST_NAME"
          },
          {
            "type": "string",
            "optional": false,
            "field": "LAST_NAME"
```

```
        },
        {
          "type": "string",
          "optional": false,
          "field": "EMAIL"
        }
      ],
      "optional": true,
      "name": "mydatabase.MYSCHEMA.CUSTOMERS.Value",
      "field": "after"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "string",
          "optional": false,
          "field": "version"
        },
        {
          "type": "string",
          "optional": false,
          "field": "connector"
        },
        {
          "type": "string",
          "optional": false,
          "field": "name"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "ts_ms"
        },
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "string",
          "optional": false,
          "field": "db"
        },
        {
          "type": "string",
          "optional": false,
          "field": "schema"
        },
        {
          "type": "string",
          "optional": false,
          "field": "table"
        },
        {
```

```
            "type": "string",
            "optional": true,
            "field": "change_lsn"
          },
          {
            "type": "string",
            "optional": true,
            "field": "commit_lsn"
          },
        ],
        "optional": false,
        "name": "io.debezium.connector.db2.Source", 3
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mydatabase.MYSCHEMA.CUSTOMERS.Envelope" 4
  },
  "payload": { 5
    "before": null, 6
    "after": { 7
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "source": { 8
      "version": "1.9.7.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
    },
    "op": "c", 9
    "ts_ms": 1559729471739 10
  }
}
```

Table 3.6. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **mydatabase.MYSCHEMA.CUSTOMERS.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. The connector uses this schema for all rows in the **MYSCHEMA.CUSTOMERS** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.schemaName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.db2.Source** is the schema for the payload's **source** field. This schema is specific to the Db2 connector. The connector uses it for all events that it generates.
4	**name**	**mydatabase.MYSCHEMA.CUSTOMERS.Envelope** is the schema for the overall structure of the payload, where **mydatabase** is the database, **MYSCHEMA** is the schema, and **CUSTOMERS** is the table.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that JSON representations of events are much larger than the rows they describe. This is because a JSON representation must include the schema portion and the payload portion of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content.
7	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **ID**, **FIRST_NAME**, **LAST_NAME**, and **EMAIL** columns.

Item	Field name	Description
8	**source**	Mandatory field that describes the source metadata for the event. The **source** structure shows Db2 information about this change, which provides traceability. It also has information you can use to compare to other events in the same topic or in other topics to know whether this event occurred before, after, or as part of the same commit as other events. The source metadata includes: • Debezium version • Connector type and name • Timestamp for when the change was made in the database • Whether the event is part of an ongoing snapshot • Name of the database, schema, and table that contain the new row • Change LSN • Commit LSN (omitted if this event is part of a snapshot)
9	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots)
10	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

update events

The value of a change event for an update in the sample **customers** table has the same schema as a *create* event for that table. Likewise, the *update* event value's payload has the same structure. However, the event value payload contains different values in an *update* event. Here is an example of a change event value in an event that the connector generates for an update in the **customers** table:

```
{
  "schema": { ... },
  "payload": {
```

```
    "before": {  1
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "after": {  2
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "noreply@example.org"
    },
    "source": {  3
      "version": "1.9.7.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
    },
    "op": "u",  4
    "ts_ms": 1559729998706  5
  }
}
```

Table 3.7. Descriptions of *update* event value fields

Item	Field name	Description
1	**before**	An optional field that specifies the state of the row before the event occurred. In an *update* event value, the **before** field contains a field for each table column and the value that was in that column before the database commit. In this example, note that the **EMAIL** value is **john.doe@example.com**.
2	**after**	An optional field that specifies the state of the row after the event occurred. You can compare the **before** and **after** structures to determine what the update to this row was. In the example, the **EMAIL** value is now **noreply@example.com**.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure contains the same fields as in a *create* event, but some values are different, for example, the sample *update* event has different LSNs. You can use this information to compare this event to other events to know whether this event occurred before, after, or as part of the same commit as other events. The source metadata includes: • Debezium version • Connector type and name • Timestamp for when the change was made in the database • Whether the event is part of an ongoing snapshot • Name of the database, schema, and table that contain the new row • Change LSN • Commit LSN (omitted if this event is part of a snapshot)
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

Updating the columns for a row's primary/unique key changes the value of the row's key. When a key changes, Debezium outputs *three* events: a **DELETE** event and a tombstone event with the old key for the row, followed by an event with the new key for the row.

delete events

The value in a *delete* change event has the same **schema** portion as *create* and *update* events for the same table. The event value **payload** in a *delete* event for the sample **customers** table looks like this:

```
{
  "schema": { ... },
  },
  "payload": {
    "before": {  1
      "ID": 1005,
      "FIRST_NAME": "john",
```

```
    "LAST_NAME": "doe",
    "EMAIL": "noreply@example.org"
  },
  "after": null,  2
  "source": {  3
    "version": "1.9.7.Final",
    "connector": "db2",
    "name": "myconnector",
    "ts_ms": 1559730445243,
    "snapshot": false,
    "db": "mydatabase",
    "schema": "MYSCHEMA",
    "table": "CUSTOMERS",
    "change_lsn": "00000027:00000db0:0005",
    "commit_lsn": "00000027:00000db0:0007"
  },
  "op": "d",  4
  "ts_ms": 1559730450205  5
 }
}
```

Table 3.8. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit.
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and LSN field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector type and name • Timestamp for when the change was made in the database • Whether the event is part of an ongoing snapshot • Name of the database, schema, and table that contain the new row • Change LSN • Commit LSN (omitted if this event is part of a snapshot)

Item	Field name	Description
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

A *delete* change event record provides a consumer with the information it needs to process the removal of this row. The old values are included because some consumers might require them in order to properly handle the removal.

Db2 connector events are designed to work with Kafka log compaction. Log compaction enables removal of some older messages as long as at least the most recent message for every key is kept. This lets Kafka reclaim storage space while ensuring that the topic contains a complete data set and can be used for reloading key-based state.

When a row is deleted, the *delete* event value still works with log compaction, because Kafka can remove all earlier messages that have that same key. However, for Kafka to remove all messages that have that same key, the message value must be **null**. To make this possible, after Debezium's Db2 connector emits a *delete* event, the connector emits a special tombstone event that has the same key but a **null** value.

3.4. HOW DEBEZIUM DB2 CONNECTORS MAP DATA TYPES

Db2's data types are described in Db2 SQL Data Types.

The Db2 connector represents changes to rows with events that are structured like the table in which the row exists. The event contains a field for each column value. How that value is represented in the event depends on the Db2 data type of the column. This section describes these mappings. If the default data type conversions do not meet your needs, you can create a custom converter for the connector.

Details are in the following sections:

- Basic types
- Temporal types
- Timestamp types
- Table 3.12, "Decimal types"

Basic types

The following table describes how the connector maps each of the Db2 data types to a *literal type* and a *semantic type* in event fields.

- *literal type* describes how the value is represented using Kafka Connect schema types: **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**.
- *semantic type* describes how the Kafka Connect schema captures the *meaning* of the field using the name of the Kafka Connect schema for the field.

Table 3.9. Mappings for Db2 basic data types

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
BOOLEAN	**BOOLEAN**	Only snapshots can be taken from tables with BOOLEAN type columns. Currently SQL Replication on Db2 does not support BOOLEAN, so Debezium can not perform CDC on those tables. Consider using a different type.
BIGINT	**INT64**	n/a
BINARY	**BYTES**	n/a
BLOB	**BYTES**	n/a
CHAR[(N)]	**STRING**	n/a
CLOB	**STRING**	n/a
DATE	**INT32**	**io.debezium.time.Date** String representation of a timestamp without timezone information
DECFLOAT	**BYTES**	**org.apache.kafka.connect.data.Decimal**
DECIMAL	**BYTES**	**org.apache.kafka.connect.data.Decimal**
DBCLOB	**STRING**	n/a
DOUBLE	**FLOAT64**	n/a
INTEGER	**INT32**	n/a
REAL	**FLOAT32**	n/a
SMALLINT	**INT16**	n/a
TIME	**INT32**	**io.debezium.time.Time** String representation of a time without timezone information

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIMESTAMP	**INT64**	**io.debezium.time.MicroTimestamp** String representation of a timestamp without timezone information
VARBINARY	**BYTES**	n/a
VARCHAR[(N)]	**STRING**	n/a
VARGRAPHIC	**STRING**	n/a
XML	**STRING**	**io.debezium.data.Xml** String representation of an XML document

If present, a column's default value is propagated to the corresponding field's Kafka Connect schema. Change events contain the field's default value unless an explicit column value had been given. Consequently, there is rarely a need to obtain the default value from the schema.

Temporal types

Other than Db2's **DATETIMEOFFSET** data type, which contains time zone information, how temporal types are mapped depends on the value of the **time.precision.mode** connector configuration property. The following sections describe these mappings:

- **time.precision.mode=adaptive**
- **time.precision.mode=connect**

time.precision.mode=adaptive

When the **time.precision.mode** configuration property is set to **adaptive**, the default, the connector determines the literal type and semantic type based on the column's data type definition. This ensures that events *exactly* represent the values in the database.

Table 3.10. Mappings when time.precision.mode is adaptive

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**io.debezium.time.Date** Represents the number of days since the epoch.
TIME(0), **TIME(1)**, **TIME(2)**, **TIME(3)**	**INT32**	**io.debezium.time.Time** Represents the number of milliseconds past midnight, and does not include timezone information.

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIME(4), **TIME(5)**, **TIME(6)**	**INT64**	**io.debezium.time.MicroTime** Represents the number of microseconds past midnight, and does not include timezone information.
TIME(7)	**INT64**	**io.debezium.time.NanoTime** Represents the number of nanoseconds past midnight, and does not include timezone information.
DATETIME	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
SMALLDATETIME	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
DATETIME2(0), **DATETIME2(1)**, **DATETIME2(2)**, **DATETIME2(3)**	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
DATETIME2(4), **DATETIME2(5)**, **DATETIME2(6)**	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds since the epoch, and does not include timezone information.
DATETIME2(7)	**INT64**	**io.debezium.time.NanoTimestamp** Represents the number of nanoseconds past the epoch, and does not include timezone information.

time.precision.mode=connect

When the **time.precision.mode** configuration property is set to **connect**, the connector uses Kafka Connect logical types. This may be useful when consumers can handle only the built-in Kafka Connect logical types and are unable to handle variable-precision time values. However, since Db2 supports tenth of a microsecond precision, the events generated by a connector with the **connect** time precision **results in a loss of precision**when the database column has a *fractional second precision* value that is greater than 3.

Table 3.11. Mappings when **time.precision.mode** is **connect**

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**org.apache.kafka.connect.data.Time** Represents the number of milliseconds since midnight, and does not include timezone information. Db2 allows **P** to be in the range 0-7 to store up to tenth of a microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.
DATETIME	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
SMALLDATETIME	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
DATETIME2	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information. Db2 allows **P** to be in the range 0-7 to store up to tenth of a microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.

Timestamp types

The **DATETIME**, **SMALLDATETIME** and **DATETIME2** types represent a timestamp without time zone information. Such columns are converted into an equivalent Kafka Connect value based on UTC. For example, the **DATETIME2** value "2018-06-20 15:13:16.945104" is represented by an **io.debezium.time.MicroTimestamp** with the value "1529507596945104".

The timezone of the JVM running Kafka Connect and Debezium does not affect this conversion.

Table 3.12. Decimal types

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes

Db2 data type	Literal type (schema type)	Semantic type (schema name) and Notes
NUMERIC[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point is shifted. The **connect.decimal.precision** schema parameter contains an integer that represents the precision of the given decimal value.
DECIMAL[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point is shifted. The **connect.decimal.precision** schema parameter contains an integer that represents the precision of the given decimal value.
SMALLMONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point iss shifted. The **connect.decimal.precision** schema parameter contains an integer that represents the precision of the given decimal value.
MONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point is shifted. The **connect.decimal.precision** schema parameter contains an integer that represents the precision of the given decimal value.

3.5. SETTING UP DB2 TO RUN A DEBEZIUM CONNECTOR

For Debezium to capture change events that are committed to Db2 tables, a Db2 database administrator with the necessary privileges must configure tables in the database for change data capture. After you begin to run Debezium you can adjust the configuration of the capture agent to optimize performance.

For details about setting up Db2 for use with the Debezium connector, see the following sections:

- Section 3.5.1, “Configuring Db2 tables for change data capture”
- Section 3.5.2, “Effect of Db2 capture agent configuration on server load and latency”
- Section 3.5.3, “Db2 capture agent configuration parameters”

3.5.1. Configuring Db2 tables for change data capture

To put tables into capture mode, Debezium provides a set of user-defined functions (UDFs) for your convenience. The procedure here shows how to install and run these management UDFs. Alternatively, you can run Db2 control commands to put tables into capture mode. The administrator must then enable CDC for each table that you want Debezium to capture.

Prerequisites

- You are logged in to Db2 as the **db2instl** user.
- On the Db2 host, the Debezium management UDFs are available in the $HOME/asncdctools/src directory. UDFs are available from the Debezium examples repository.

Procedure

1. Compile the Debezium management UDFs on the Db2 server host by using the **bldrtn** command provided with Db2:

```
cd $HOME/asncdctools/src
```

```
./bldrtn asncdc
```

2. Start the database if it is not already running. Replace **DB_NAME** with the name of the database that you want Debezium to connect to.

```
db2 start db DB_NAME
```

3. Ensure that JDBC can read the Db2 metadata catalog:

```
cd $HOME/sqllib/bnd
```

```
db2 bind db2schema.bnd blocking all grant public sqlerror continue
```

4. Ensure that the database was recently backed-up. The ASN agents must have a recent starting point to read from. If you need to perform a backup, run the following commands, which prune the data so that only the most recent version is available. If you do not need to retain the older versions of the data, specify **dev/null** for the backup location.

   a. Back up the database. Replace **DB_NAME** and **BACK_UP_LOCATION** with appropriate values:

```
db2 backup db DB_NAME to BACK_UP_LOCATION
```

   b. Restart the database:

```
db2 restart db DB_NAME
```

5. Connect to the database to install the Debezium management UDFs. It is assumed that you are logged in as the **db2instl** user so the UDFs should be installed on the **db2inst1** user.

```
db2 connect to DB_NAME
```

6. Copy the Debezium management UDFs and set permissions for them:

```
cp $HOME/asncdctools/src/asncdc $HOME/sqllib/function
```

```
chmod 777 $HOME/sqllib/function
```

7. Enable the Debezium UDF that starts and stops the ASN capture agent:

```
db2 -tvmf $HOME/asncdctools/src/asncdc_UDF.sql
```

8. Create the ASN control tables:

```
$ db2 -tvmf $HOME/asncdctools/src/asncdctables.sql
```

9. Enable the Debezium UDF that adds tables to capture mode and removes tables from capture mode:

```
$ db2 -tvmf $HOME/asncdctools/src/asncdcaddremove.sql
```

After you set up the Db2 server, use the UDFs to control Db2 replication (ASN) with SQL commands. Some of the UDFs expect a return value in which case you use the SQL **VALUE** statement to invoke them. For other UDFs, use the SQL **CALL** statement.

10. Start the ASN agent:

```
VALUES ASNCDC.ASNCDCSERVICES('start','asncdc');
```

The preceding statement returns one of the following results:

- **asncap is already running**
- **start --> *<COMMAND>***
  In this case, enter the specified ***<COMMAND>*** in the terminal window as shown in the following example:

```
/database/config/db2inst1/sqllib/bin/asncap capture_schema=asncdc
capture_server=SAMPLE &
```

11. Put tables into capture mode. Invoke the following statement for each table that you want to put into capture. Replace **MYSCHEMA** with the name of the schema that contains the table you want to put into capture mode. Likewise, replace **MYTABLE** with the name of the table to put into capture mode:

```
CALL ASNCDC.ADDTABLE('MYSCHEMA', 'MYTABLE');
```

12. Reinitialize the ASN service:

```
VALUES ASNCDC.ASNCDCSERVICES('reinit','asncdc');
```

Additional resource

Reference table for Debezium Db2 management UDFs

3.5.2. Effect of Db2 capture agent configuration on server load and latency

When a database administrator enables change data capture for a source table, the capture agent begins to run. The agent reads new change event records from the transaction log and replicates the event records to a capture table. Between the time that a change is committed in the source table, and the time that the change appears in the corresponding change table, there is always a small latency interval. This latency interval represents a gap between when changes occur in the source table and when they become available for Debezium to stream to Apache Kafka.

Ideally, for applications that must respond quickly to changes in data, you want to maintain close synchronization between the source and capture tables. You might imagine that running the capture agent to continuously process change events as rapidly as possible might result in increased throughput and reduced latency – populating change tables with new event records as soon as possible after the events occur, in near real time. However, this is not necessarily the case. There is a performance penalty to pay in the pursuit of more immediate synchronization. Each time that the change agent queries the database for new event records, it increases the CPU load on the database host. The additional load on the server can have a negative effect on overall database performance, and potentially reduce transaction efficiency, especially during times of peak database use.

It's important to monitor database metrics so that you know if the database reaches the point where the server can no longer support the capture agent's level of activity. If you experience performance issues while running the capture agent, adjust capture agent settings to reduce CPU load.

3.5.3. Db2 capture agent configuration parameters

On Db2, the **IBMSNAP_CAPPARMS** table contains parameters that control the behavior of the capture agent. You can adjust the values for these parameters to balance the configuration of the capture process to reduce CPU load and still maintain acceptable levels of latency.

NOTE

Specific guidance about how to configure Db2 capture agent parameters is beyond the scope of this documentation.

In the **IBMSNAP_CAPPARMS** table, the following parameters have the greatest effect on reducing CPU load:

COMMIT_INTERVAL

- Specifies the number of seconds that the capture agent waits to commit data to the change data tables.
- A higher value reduces the load on the database host and increases latency.
- The default value is **30**.

SLEEP_INTERVAL

- Specifies the number of seconds that the capture agent waits to start a new commit cycle after it reaches the end of the active transaction log.
- A higher value reduces the load on the server, and increases latency.
- The default value is **5**.

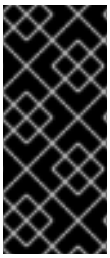
Additional resources

- For more information about capture agent parameters, see the Db2 documentation.

3.6. DEPLOYMENT OF DEBEZIUM DB2 CONNECTORS

You can use either of the following methods to deploy a Debezium Db2 connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .



IMPORTANT

Due to licensing requirements, the Debezium Db2 connector archive does not include the Db2 JDBC driver that Debezium requires to connect to a Db2 database. To enable the connector to access the database, you must add the driver to your connector environment. For information about how to obtain the driver, see Obtaining the Db2 JDBC driver.

Additional resources

- Section 3.6.6, "Descriptions of Debezium Db2 connector configuration properties"

3.6.1. Obtaining the Db2 JDBC driver

Due to licensing requirements, the Db2 JDBC driver file that Debezium requires to connect to an Db2 database is not included in the Debezium Db2 connector archive. The driver is available for download from Maven Central. Depending on the deployment method that you use, you retrieve the driver by adding a command to the Kafka Connect custom resource or to the Dockerfile that you use to build the connector image.

- If you use AMQ Streams to add the connector to your Kafka Connect image, add the Maven Central location for the driver to **builds.plugins.artifact.url** in the **KafkaConnect** custom resource as shown in Section 3.6.3, "Using AMQ Streams to deploy a Debezium Db2 connector" .
- If you use a Dockerfile to build a container image for the connector, insert a **curl** command in the Dockerfile to specify the URL for downloading the required driver file from Maven Central. For more information, see Section 3.6.4, "Deploying a Debezium Db2 connector by building a custom Kafka Connect container image from a Dockerfile".

3.6.2. Db2 connector deployment using AMQ Streams

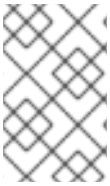
Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.



NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

3.6.3. Using AMQ Streams to deploy a Debezium Db2 connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.

- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

   Example 3.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

   In the example that follows, the custom resource is configured to download the following artifacts:

   - The Debezium Db2 connector archive.
   - The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
   - The Debezium scripting SMT archive and the associated language dependencies that you want to use with the Debezium connector. The SMT archive and language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.
   - The Db2 JDBC driver, which is required to connect to Db2 databases, but is not included in the connector archive.

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: debezium-kafka-connect-cluster
     annotations:
       strimzi.io/use-connector-resources: "true" 1
   spec:
     version: 3.00
     build: 2
       output: 3
         type: imagestream 4
         image: debezium-streams-connect:latest
       plugins: 5
         - name: debezium-connector-db2
           artifacts:
   ```

```
      - type: zip (6)
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-db2/1.9.7.Final-redhat-<build_number>/debezium-connector-db2-1.9.7.Final-redhat-<build_number>-plugin.zip (7)
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip (8)
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip (9)
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar (10)
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json3.0.11/groovy-json-3.0.11.jar
      - type: jar (11)
        url: https://repo1.maven.org/maven2/com/ibm/db2/jcc/11.5.0.0/jcc-11.5.0.0.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 3.13. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry like Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.

Item	Description
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. JDBC driver files are in **.jar** format. The **type** value must match the type of the file that is referenced in the **url** field.
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components, **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) The Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.
11	Specifies the location of the Db2 JDBC driver in Maven Central. The required driver is not included in the Debezium Db2 connector archive.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **db2-inventory-connector.yaml**

Example 3.2. db2-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-db2 (1)
spec:
  class: io.debezium.connector.db2.Db2ConnectorConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: db2.debezium-db2.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_db2 (10)
    database.include.list: public.inventory (11)
```

Table 3.14. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.
4	The connector’s configuration.
5	The address of the host database instance.

Item	Description
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.
8	The password for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium Db2 deployment.

3.6.4. Deploying a Debezium Db2 connector by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium Db2 connector, you must build a custom Kafka Connect container image that contains the Debezium connector archive, and then push this container image to a container registry. You then need to create the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.
- A **KafkaConnector** CR that defines your Debezium Db2 connector. Apply this CR to the same OpenShift instance where you applied the **KafkaConnect** CR.

Prerequisites

- Db2 is running and you completed the steps to set up Db2 to work with a Debezium connector .
- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift .
- Podman or Docker is installed.
- The Kafka Connect server has access to Maven Central to download the required JDBC driver for Db2. You can also use a local copy of the driver, or one that is available from a local Maven repository or other HTTP server.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium Db2 container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-db2.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-db2/1.9.7.Final-redhat-<build_number>/debezium-connector-db2-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-db2-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-db2-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://repo1.maven.org/maven2/com/ibm/db2/jcc/11.5.0.0/jcc-11.5.0.0.jar
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

   The command creates a Dockerfile with the name **debezium-container-for-db2.yaml** in the current directory.

   b. Build the container image from the **debezium-container-for-db2.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-db2:latest .
```

```
docker build -t debezium-container-for-db2:latest .
```

The preceding commands build a container image with the name **debezium-container-for-db2**.

c. Push your custom image to a container registry, such as quay.io or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-db2:latest
```

```
docker push <myregistry.io>/debezium-container-for-db2:latest
```

d. Create a new Debezium Db2 **KafkaConnect** custom resource (CR). For example, create a **KafkaConnect** CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  #...
  image: debezium-container-for-db2 2
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply the **KafkaConnect** CR to the OpenShift Kafka Connect environment by entering the following command:

```
oc create -f dbz-connect.yaml
```

The command adds a Kafka Connect instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium Db2 connector instance.
You configure a Debezium Db2 connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce events for a subset of the schemas and tables, or it might set properties so that Debezium

ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed.

The following example configures a Debezium connector that connects to a Db2 server host, **192.168.99.100**, on port **50000**. This host has a database named **mydatabase**, a table with the name **inventory**, and **fulfillment** is the server's logical name.

Db2 inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
  name: inventory-connector 1
  labels:
   strimzi.io/cluster: my-connect-cluster
  annotations:
   strimzi.io/use-connector-resources: 'true'
spec:
 class: io.debezium.connector.db2.Db2Connector 2
 tasksMax: 1 3
 config: 4
  database.hostname: 192.168.99.100 5
  database.port: 50000 6
  database.user: db2inst1 7
  database.password: Password! 8
  database.dbname: mydatabase 9
  database.server.name: fullfillment 10
  database.include.list: public.inventory 11
```

Table 3.15. Descriptions of connector configuration settings

Item	Description
1	The name of the connector when we register it with a Kafka Connect cluster.
2	The name of this Db2 connector class.
3	Only one task should operate at any one time.
4	The connector's configuration.
5	The database host, which is the address of the Db2 instance.
6	The port number of the Db2 instance.
7	The name of the Db2 user.
8	The password for the Db2 user.
9	The name of the database to capture changes from.

Item	Description
10	The logical name of the Db2 instance/cluster, which forms a namespace and is used in the names of the Kafka topics to which the connector writes, the names of Kafka Connect schemas, and the namespaces of the corresponding Avro schema when the Avro Connector is used.
11	A list of all tables whose changes Debezium should capture.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **inventory-connector.yaml** file, you would run the following command:

```
oc apply -f inventory-connector.yaml
```

The preceding command registers **inventory-connector** and the connector starts to run against the **mydatabase** database as defined in the **KafkaConnector** CR.

For the complete list of the configuration properties that you can set for the Debezium Db2 connector, see Db2 connector properties.

Results

After the connector starts, it performs a consistent snapshot of the Db2 database tables that the connector is configured to capture changes for. The connector then starts generating data change events for row-level operations and streaming change event records to Kafka topics.

3.6.5. Verifying that the Debezium Db2 connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-db2**.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
     a. Enter the following command:

     ```
     oc describe KafkaConnector <connector-name> -n <project>
     ```

     For example,

     ```
     oc describe KafkaConnector inventory-connector-db2 -n debezium
     ```

     The command returns status information that is similar to the following output:

     Example 3.3. KafkaConnector resource status

     ```
     Name:         inventory-connector-db2
     Namespace:    debezium
     Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
     Annotations:  <none>
     API Version:  kafka.strimzi.io/v1beta2
     Kind:         KafkaConnector

     ...

     Status:
       Conditions:
         Last Transition Time:  2021-12-08T17:41:34.897153Z
         Status:                True
         Type:                  Ready
       Connector Status:
         Connector:
           State:      RUNNING
           worker_id:  10.131.1.124:8083
         Name:         inventory-connector-db2
         Tasks:
           Id:               0
           State:            RUNNING
           worker_id:        10.131.1.124:8083
         Type:               source
       Observed Generation:  1
       Tasks Max:            1
     ```

```
  Topics:
    inventory_connector_db2
    inventory_connector_db2.inventory.addresses
    inventory_connector_db2.inventory.customers
    inventory_connector_db2.inventory.geom
    inventory_connector_db2.inventory.orders
    inventory_connector_db2.inventory.products
    inventory_connector_db2.inventory.products_on_hand
  Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-db2.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

   - From a terminal window:

     a. Enter the following command:

     ```
     oc get kafkatopics
     ```

     The command returns status information that is similar to the following output:

     Example 3.4. KafkaTopic resource status

     ```
     NAME                                                                CLUSTER
     PARTITIONS   REPLICATION FACTOR   READY
     connect-cluster-configs                                              debezium-
     kafka-cluster   1           1                 True
     connect-cluster-offsets                                              debezium-
     kafka-cluster   25          1                 True
     connect-cluster-status                                               debezium-
     kafka-cluster   5           1                 True
     consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
     debezium-kafka-cluster   50          1                 True
     inventory-connector-db2---a96f69b23d6118ff415f772679da623fbbb99421
     debezium-kafka-cluster   1           1                 True
     inventory-connector-db2.inventory.addresses---
     1b6beaf7b2eb57d177d92be90ca2b210c9a56480         debezium-kafka-cluster
     1           1                 True
     inventory-connector-db2.inventory.customers---
     9931e04ec92ecc0924f4406af3fdace7545c483b         debezium-kafka-cluster   1
     1                 True
     inventory-connector-db2.inventory.geom---
     ```

```
9f7e136091f071bf49ca59bf99e86c713ee58dd5              debezium-kafka-cluster
1          1                True
inventory-connector-db2.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d              debezium-kafka-cluster
1          1                True
inventory-connector-db2.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1                True
inventory-connector-db2.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1                True
schema-changes.inventory
debezium-kafka-cluster   1          1                True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1          1                True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1                True
```

3. Check topic content.

   - From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

   For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_db2.inventory.products_on_hand
```

   The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_db2.inventory.addresses**.

   For each event in the topic, the command returns information that is similar to the following output:

   Example 3.5. Content of a Debezium change event

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_conne
ctor_db2.inventory.products_on_hand.Key"},"payload":{"product_id":101}} {"schema":
{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_db2.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
```

{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector_db2.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":[{"type":"string","optional":false,"field":"version"},{"type":"string","optional":false,"field":"connector"},{"type":"string","optional":false,"field":"name"},{"type":"int64","optional":false,"field":"ts_ms"},{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":{"allowed":"true,last,false"},"default":"false","field":"snapshot"},{"type":"string","optional":false,"field":"db"},{"type":"string","optional":true,"field":"sequence"},{"type":"string","optional":true,"field":"table"},{"type":"int64","optional":false,"field":"server_id"},{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},{"type":"int64","optional":true,"field":"thread"},{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.db2.Source","field":"source"},{"type":"string","optional":false,"field":"op"},{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":[{"type":"string","optional":false,"field":"id"},{"type":"int64","optional":false,"field":"total_order"},{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transaction"}],"optional":false,"name":"inventory_connector_db2.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":{"version":"1.9.7.Final-redhat-00001","connector":"db2","name":"inventory_connector_db2","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"db2-bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"transaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

3.6.6. Descriptions of Debezium Db2 connector configuration properties

The Debezium Db2 connector has numerous configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values. Information about the properties is organized as follows:

- Required configuration properties
- Advanced configuration properties
- Database history connector configuration properties that control how Debezium processes events that it reads from the database history topic.
  - Pass-through database history properties
- Pass-through database driver properties that control the behavior of the database driver.

Required Debezium Db2 connector configuration properties

The following configuration properties are *required* unless a default value is available.

Property	Default	Description
name	No default	Unique name for the connector. Attempting to register again with the same name will fail. This property is required by all Kafka Connect connectors.
connector.class	No default	The name of the Java class for the connector. Always use a value of **io.debezium.connector.db2.Db2Connector** for the Db2 connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The Db2 connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname	No default	IP address or hostname of the Db2 database server.
database.port	**50000**	Integer port number of the Db2 database server.
database.user	No default	Name of the Db2 database user for connecting to the Db2 database server.
database.password	No default	Password to use when connecting to the Db2 database server.
database.dbname	No default	The name of the Db2 database from which to stream the changes

Property	Default	Description
database.server.name	No default	Logical name that identifies and provides a namespace for the particular Db2 database server that hosts the database for which Debezium is capturing changes. Only alphanumeric characters, hyphens, dots and underscores must be used in the database server logical name. The logical name should be unique across all other connectors, since it is used as a topic name prefix for all Kafka topics that receive records from this connector. + **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value. The connector is also unable to recover its database history topic.
table.include.list	No default	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you want the connector to capture. Any table not included in the include list does not have its changes captured. Each identifier is of the form *schemaName*.*tableName*. By default, the connector captures changes in every non-system table. Do not also set the **table.exclude.list** property.

Property	Default	Description
table.exclude.list	No default	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you do not want the connector to capture. The connector captures changes in each non-system table that is not included in the exclude list. Each identifier is of the form *schemaName.tableName*. Do not also set the **table.include.list** property.
column.exclude.list	*empty string*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to exclude from change event values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Primary key columns are always included in the event's key, even if they are excluded from the value.

Property	Default	Description
column.mask.hash.*hashAlgorithm*.with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In the resulting change event record, the values for the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. In the following example, **CzQMA0cB5K** is a randomly selected salt. `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts. Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting data set might not be completely masked.

Property	Default	Description
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision: **adaptive** captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type. **connect** always represents time and timestamp values by using Kafka Connect's built-in representations for **Time**, **Date**, and **Timestamp**, which uses millisecond precision regardless of the database columns' precision. See temporal values.
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. **true** - a delete operation is represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is emitted. After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case log compaction is enabled for the topic.
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded with a key that contains the database name and a value that is a JSON structure that describes the schema update. This is independent of how the connector internally records database history.

Property	Default	Description
column.truncate.to.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event records, values in these columns are truncated if they are longer than the number of characters specified by *length* in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, **column.truncate.to.20.chars**.
column.mask.with.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event values, the values in the specified table columns are replaced with *length* number of asterisk (*) characters. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer or zero. When you specify zero, the connector replaces a value with an empty string.
column.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*, or *databaseName.schemaName.tableName.columnName*. For each specified column, the connector adds the column's original type and original length as parameters to the corresponding field schemas in the emitted change records. Add the following schema parameters to propagate the original type name and the original length for variable-width types: **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** This property is useful for properly sizing corresponding columns in sink databases.

Property	Default	Description
datatype.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the database-specific data type name for some columns. Fully-qualified data type names are of the form *databaseName.tableName.typeName*, or *databaseName.schemaName.tableName.typeName*. For these data types, the connector adds parameters to the corresponding field schemas in emitted change records. The added parameters specify the original type and length of the column: **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** These parameters propagate a column's original type name and length, for variable-width types, respectively. This property is useful for properly sizing corresponding columns in sink databases. See Db2 data types for the list of Db2-specific data type names.

Property	Default	Description
message.key.columns	*empty string*	A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables. By default, Debezium uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns. To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format: ***<fully-qualified_tableName>:<keyColumn>,<keyColumn>*** To base a table key on multiple column names, insert commas between the column names. Each fully-qualified table name is a regular expression in the following format: ***<schemaName>.<tableName>*** The property can list entries for multiple tables. Use a semicolon to separate entries for different tables in the list. The following example sets the message key for the tables **inventory.customers** and **purchaseorders**: **inventory.customers:pk1,pk2; (.*).purchaseorders:pk3,pk4** In the preceding example, the columns **pk1** and **pk2** are specified as the message key for the table **inventory.customer**. For **purchaseorders** tables in any schema, the columns **pk3** and **pk4** serve as the message key.

Property	Default	Description
schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.

Advanced connector configuration properties

The following *advanced* configuration properties have defaults that work in most situations and therefore rarely need to be specified in the connector's configuration.

Property	Default	Description

Property	Default	Description

converters	No default	Enumerates a comma-separated list of the symbolic names of the custom converter instances that the connector can use. For example,
		
		isbn
		
		You must set the **converters** property to enable the connector to use a custom converter.
		
		For each converter that you configure for a connector, you must also add a **.type** property, which specifies the fully-qualifed name of the class that implements the converter interface. The **.type** property uses the following format:
		
		***<converterSymbolicName>*.type**
		
		For example,
		
		`isbn.type: io.debezium.test.IsbnConverter`
		
		If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate any additional configuration parameter with a converter, prefix the parameter names with the symbolic name of the converter.
		For example,
		
		`isbn.schema.name: io.debezium.db2.type.Isbn`

Property	Default	Description
snapshot.mode	**initial**	Specifies the criteria for performing a snapshot when the connector starts. Set the property to one of the following settings: **initial** - For tables in capture mode, the connector takes a snapshot of the schema for the table and the data in the table. Specify this value to populate Kafka topics with a complete representation of the data from the captured tables. **initial_only** - Similar to the **initial** setting, the connector takes a snapshot of structure and data of captured tables, but after the snapshot completes, the connector does not stream subsequent changes. **schema_only** - For tables in capture mode, the connector takes a snapshot of only the schema for the table. Use this setting when you want the snapshot to capture only the structure of tables, but after the snapshot completes, you want the connector to emit data for subsequent changes that occur in the captured tables. After the snapshot is complete, the connector continues to read change events from the database's redo logs.

Property	Default	Description
snapshot.isolation.mode	**repeatable_read**	During a snapshot, controls the transaction isolation level and how long the connector locks the tables that are in capture mode. The possible values are: **read_uncommitted** - Does not prevent other transactions from updating table rows during an initial snapshot. This mode has no data consistency guarantees; some data might be lost or corrupted. **read_committed** - Does not prevent other transactions from updating table rows during an initial snapshot. It is possible for a new record to appear twice: once in the initial snapshot and once in the streaming phase. However, this consistency level is appropriate for data mirroring. **repeatable_read** - Prevents other transactions from updating table rows during an initial snapshot. It is possible for a new record to appear twice: once in the initial snapshot and once in the streaming phase. However, this consistency level is appropriate for data mirroring. **exclusive** - Uses repeatable read isolation level but takes an exclusive lock for all tables to be read. This mode prevents other transactions from updating table rows during an initial snapshot. Only **exclusive** mode guarantees full consistency; the initial snapshot and streaming logs constitute a linear history.
event.processing.failure.handling.mode	**fail**	Specifies how the connector handles exceptions during processing of events. The possible values are: **fail** - The connector logs the offset of the problematic event and stops processing. **warn** - The connector logs the offset of the problematic event and continues processing with the next event. **skip** - The connector skips the problematic event and continues processing with the next event.

Property	Default	Description
poll.interval.ms	1000	Positive integer value that specifies the number of milliseconds the connector should wait for new change events to appear before it starts processing a batch of events. Defaults to 1000 milliseconds, or 1 second.
max.batch.size	2048	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
max.queue.size	8192	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.
max.queue.size.in.bytes	0	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.

Property	Default	Description
heartbeat.interval.ms	**0**	Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages. Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are useful when there are many updates in a database that is being tracked but only a tiny number of updates are in tables that are in capture mode. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that the connector has few opportunities to send the latest offset to Kafka. Sending heartbeat messages enables the connector to send the latest offset to Kafka.
heartbeat.topics.prefix	**__debezium-heartbeat**	Specifies the prefix for the name of the topic to which the connector sends heartbeat messages. The format for this topic name is **<heartbeat.topics.prefix>.<server.name>**.
snapshot.delay.ms	No default	An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
snapshot.include.collection.list	All tables specified in **table.include.list**	An optional, comma-separated list of regular expressions that match the fully-qualified names (***<schemaName>.<tableName>***) of the tables to include in a snapshot. The specified items must be named in the connector's table.include.list property. This property takes effect only if the connector's **snapshot.mode** property is set to a value other than **never**. This property does not affect the behavior of incremental snapshots.

Property	Default	Description
snapshot.fetch.size	**2000**	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
snapshot.lock.timeout.ms	**10000**	Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this interval, the snapshot fails. How the connector performs snapshots provides details. Other possible settings are: **0** - The connector immediately fails when it cannot obtain a lock. **-1** - The connector waits infinitely.

Property	Default	Description
snapshot.select.statement.overrides	No default	Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log. The property contains a comma-separated list of fully-qualified table names in the form ***<schemaName>.<tableName>***. For example, **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** For each table in the list, add a further configuration property that specifies the **SELECT** statement for the connector to run on the table when it takes a snapshot. The specified **SELECT** statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this **SELECT** statement property: **snapshot.select.statement.overrides.*<schemaName>.<tableName>***. For example, **snapshot.select.statement.overrides.customers.orders**. Example: From a **customers.orders** table that includes the soft-delete column, **delete_flag**, add the following properties if you want a snapshot to include only those records that are not soft-deleted: "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" In the resulting snapshot, the connector includes only the records for which **delete_flag = 0**.

Property	Default	Description
sanitize.field.names	**true** if connector configuration sets the **key.converter** or **value.converter** property to the Avro converter. **false** if not.	Indicates whether field names are sanitized to adhere to Avro naming requirements.
provide.transaction.metadata	**false**	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify **true** if you want the connector to do this. See Transaction metadata for details.
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name; defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**.
skipped.operations	No default	comma-separated list of operation types that will be skipped during streaming. The operations include: **c** for inserts/create, **u** for updates, and **d** for deletes. By default, no operations are skipped.
signal.data.collection	No default	Fully-qualified name of the data collection that is used to send signals to the connector. Use the following format to specify the collection name: ***<schemaName>.<tableName>***
incremental.snapshot.chunk.size	**1024**	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.

Debezium connector database history configuration properties

Debezium provides a set of **database.history.*** properties that control how the connector interacts with the schema history topic.

The following table describes the **database.history** properties for configuring the Debezium connector.

Table 3.16. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic	No default	The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.bootstrap.servers	No default	A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. This connection is used for retrieving the database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
database.history.kafka.recovery.poll.interval.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait during startup/recovery while polling for persisted data. The default is 100ms.
database.history.kafka.query.timeout.ms	**3000**	An integer value that specifies the maximum number of milliseconds the connector should wait while fetching cluster information using Kafka admin client.
database.history.kafka.recovery.attempts	**4**	The maximum number of times that the connector should try to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is **recovery.attempts** x **recovery.poll.interval.ms**.
database.history.skip.unparseable.ddl	**false**	A Boolean value that specifies whether the connector should ignore malformed or unknown database statements or stop processing so a human can fix the issue. The safe default is **false**. Skipping should be used only with care as it can lead to data loss or mangling when the binlog is being processed.
database.history.store.only.monitored.tables.ddl *Deprecated and scheduled for removal in a future release; use **database.history.store.only.captured.tables.ddl** instead.*	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Property	Default	Description
database.history.store.only.captured.tables.ddl	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Pass-through database history properties for configuring producer and consumer clients

Debezium relies on a Kafka producer to write schema changes to database history topics. Similarly, it relies on a Kafka consumer to read from database history topics when a connector starts. You define the configuration for the Kafka producer and consumer clients by assigning values to a set of pass-through configuration properties that begin with the **database.history.producer.*** and **database.history.consumer.*** prefixes. The pass-through producer and consumer database history properties control a range of behaviors, such as how these clients secure connections with the Kafka broker, as shown in the following example:

```
database.history.producer.security.protocol=SSL
database.history.producer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.producer.ssl.keystore.password=test1234
database.history.producer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.producer.ssl.truststore.password=test1234
database.history.producer.ssl.key.password=test1234

database.history.consumer.security.protocol=SSL
database.history.consumer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.consumer.ssl.keystore.password=test1234
database.history.consumer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.consumer.ssl.truststore.password=test1234
database.history.consumer.ssl.key.password=test1234
```

Debezium strips the prefix from the property name before it passes the property to the Kafka client.

See the Kafka documentation for more details about Kafka producer configuration properties and Kafka consumer configuration properties.

Debezium connector pass-through database driver configuration properties

The Debezium connector provides for pass-through configuration of the database driver. Pass-through database properties begin with the prefix **database.***. For example, the connector passes properties such as **database.foobar=false** to the JDBC URL.

As is the case with the pass-through properties for database history clients, Debezium strips the prefixes from the properties before it passes them to the database driver.

3.7. MONITORING DEBEZIUM DB2 CONNECTOR PERFORMANCE

The Debezium Db2 connector provides three types of metrics that are in addition to the built-in support for JMX metrics that Apache ZooKeeper, Apache Kafka, and Kafka Connect provide.

- Snapshot metrics provide information about connector operation while performing a snapshot.
- Streaming metrics provide information about connector operation when the connector is capturing changes and streaming change event records.
- Schema history metrics provide information about the status of the connector's schema history.

Debezium monitoring documentation provides details for how to expose these metrics by using JMX.

3.7.1. Monitoring Debezium during snapshots of Db2 databases

The **MBean** is **debezium.db2:type=connector-metrics,context=snapshot,server=*<db2.server.name>***. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.

Attributes	Type	Description
QueueRemainingCapacity	int	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	int	The total number of tables that are being included in the snapshot.
RemainingTableCount	int	The number of tables that the snapshot has yet to copy.
SnapshotRunning	boolean	Whether the snapshot was started.
SnapshotAborted	boolean	Whether the snapshot was aborted.
SnapshotCompleted	boolean	Whether the snapshot completed.
SnapshotDurationInSeconds	long	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	Map<String, Long>	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The connector also provides the following additional snapshot metrics when an incremental snapshot is executed:

Attributes	Type	Description
ChunkId	string	The identifier of the current snapshot chunk.
ChunkFrom	string	The lower bound of the primary key set defining the current chunk.
ChunkTo	string	The upper bound of the primary key set defining the current chunk.
TableFrom	string	The lower bound of the primary key set of the currently snapshotted table.
TableTo	string	The upper bound of the primary key set of the currently snapshotted table.

3.7.2. Monitoring Debezium Db2 connector record streaming

The **MBean** is **debezium.db2:type=connector-metrics,context=streaming,server=*<db2.server.name>***. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	string	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	long	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	long	The total number of events that this connector has seen since the last start or metrics reset.
TotalNumberOfCreateEventsSeen	long	The total number of create events that this connector has seen since the last start or metrics reset.

Attributes	Type	Description
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.

Attributes	Type	Description
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

3.7.3. Monitoring Debezium Db2 connector schema history

The MBean is **debezium.db2:type=connector-metrics,context=schema-history,server=*<db2.server.name>***.

The following table lists the schema history metrics that are available.

Attributes	Type	Description
Status	**string**	One of **STOPPED**, **RECOVERING** (recovering history from the storage), **RUNNING** describing the state of the database history.
RecoveryStartTime	**long**	The time in epoch seconds at what recovery has started.
ChangesRecovered	**long**	The number of changes that were read during recovery phase.
ChangesApplied	**long**	the total number of schema changes applied during recovery and runtime.

Attributes	Type	Description
MilliSecondsSinceLastRecoveredChange	long	The number of milliseconds that elapsed since the last change was recovered from the history store.
MilliSecondsSinceLastAppliedChange	long	The number of milliseconds that elapsed since the last change was applied.
LastRecoveredChange	string	The string representation of the last change recovered from the history store.
LastAppliedChange	string	The string representation of the last applied change.

3.8. MANAGING DEBEZIUM DB2 CONNECTORS

After you deploy a Debezium Db2 connector, use the Debezium management UDFs to control Db2 replication (ASN) with SQL commands. Some of the UDFs expect a return value in which case you use the SQL **VALUE** statement to invoke them. For other UDFs, use the SQL **CALL** statement.

Table 3.17. Descriptions of Debezium management UDFs

Task	Command and notes
Start the ASN agent	**VALUES ASNCDC.ASNCDCSERVICES('start','asncdc');**
Stop the ASN agent	**VALUES ASNCDC.ASNCDCSERVICES('stop','asncdc');**
Check the status of the ASN agent	**VALUES ASNCDC.ASNCDCSERVICES('status','asncdc');**
Put a table into capture mode	**CALL ASNCDC.ADDTABLE('MYSCHEMA', 'MYTABLE');** Replace **MYSCHEMA** with the name of the schema that contains the table you want to put into capture mode. Likewise, replace **MYTABLE** with the name of the table to put into capture mode.
Remove a table from capture mode	**CALL ASNCDC.REMOVETABLE('MYSCHEMA', 'MYTABLE');**

Task	Command and notes
Reinitialize the ASN service	**VALUES ASNCDC.ASNCDCSERVICES('reinit','asncdc');** Do this after you put a table into capture mode or after you remove a table from capture mode.

3.9. UPDATING SCHEMAS FOR DB2 TABLES IN CAPTURE MODE FOR DEBEZIUM CONNECTORS

While a Debezium Db2 connector can capture schema changes, to update a schema, you must collaborate with a database administrator to ensure that the connector continues to produce change events. This is required by the way that Db2 implements replication.

For each table in capture mode, Db2's replication feature creates a change-data table that contains all changes to that source table. However, change-data table schemas are static. If you update the schema for a table in capture mode then you must also update the schema of its corresponding change-data table. A Debezium Db2 connector cannot do this. A database administrator with elevated privileges must update schemas for tables that are in capture mode.

WARNING

It is vital to execute a schema update procedure completely before there is a new schema update on the same table. Consequently, the recommendation is to execute all DDLs in a single batch so the schema update procedure is done only once.

There are generally two procedures for updating table schemas:

- Offline - executed while Debezium is stopped
- Online - executed while Debezium is running

Each approach has advantages and disadvantages.

3.9.1. Performing offline schema updates for Debezium Db2 connectors

You stop the Debezium Db2 connector before you perform an offline schema update. While this is the safer schema update procedure, it might not be feasible for applications with high-availability requirements.

Prerequisites

- One or more tables that are in capture mode require schema updates.

Procedure

1. Suspend the application that updates the database.
2. Wait for the Debezium connector to stream all unstreamed change event records.
3. Stop the Debezium connector.
4. Apply all changes to the source table schema.
5. In the ASN register table, mark the tables with updated schemas as **INACTIVE**.
6. Reinitialize the ASN capture service .
7. Remove the source table with the old schema from capture mode by running the Debezium UDF for removing tables from capture mode.
8. Add the source table with the new schema to capture mode by running the Debezium UDF for adding tables to capture mode.
9. In the ASN register table, mark the updated source tables as **ACTIVE**.
10. Reinitialize the ASN capture service.
11. Resume the application that updates the database.
12. Restart the Debezium connector.

3.9.2. Performing online schema updates for Debezium Db2 connectors

An online schema update does not require application and data processing downtime. That is, you do not stop the Debezium Db2 connector before you perform an online schema update. Also, an online schema update procedure is simpler than the procedure for an offline schema update.

However, when a table is in capture mode, after a change to a column name, the Db2 replication feature continues to use the old column name. The new column name does not appear in Debezium change events. You must restart the connector to see the new column name in change events.

Prerequisites

- One or more tables that are in capture mode require schema updates.

Procedure when adding a column to the end of a table

1. Lock the source tables whose schema you want to change.
2. In the ASN register table, mark the locked tables as **INACTIVE**.
3. Reinitialize the ASN capture service.
4. Apply all changes to the schemas for the source tables.
5. Apply all changes to the schemas for the corresponding change-data tables.
6. In the ASN register table, mark the source tables as **ACTIVE**.
7. Reinitialize the ASN capture service.
8. Optional. Restart the connector to see updated column names in change events.

Procedure when adding a column to the middle of a table

1. Lock the source table(s) to be changed.
2. In the ASN register table, mark the locked tables as **INACTIVE**.
3. Reinitialize the ASN capture service.
4. For each source table to be changed:
    a. Export the data in the source table.
    b. Truncate the source table.
    c. Alter the source table and add the column.
    d. Load the exported data into the altered source table.
    e. Export the data in the source table's corresponding change-data table.
    f. Truncate the change-data table.
    g. Alter the change-data table and add the column.
    h. Load the exported data into the altered change-data table.
5. In the ASN register table, mark the tables as **INACTIVE**. This marks the old change-data tables as inactive, which allows the data in them to remain but they are no longer updated.
6. Reinitialize the ASN capture service.
7. Optional. Restart the connector to see updated column names in change events.

CHAPTER 4. DEBEZIUM CONNECTOR FOR MONGODB

Debezium's MongoDB connector tracks a MongoDB replica set or a MongoDB sharded cluster for document changes in databases and collections, recording those changes as events in Kafka topics. The connector automatically handles the addition or removal of shards in a sharded cluster, changes in membership of each replica set, elections within each replica set, and awaiting the resolution of communications problems.

For information about the MongoDB versions that are compatible with this connector, see the Debezium Supported Configurations page.

Information and procedures for using a Debezium MongoDB connector is organized as follows:

- Section 4.1, "Overview of Debezium MongoDB connector"
- Section 4.2, "How Debezium MongoDB connectors work"
- Section 4.3, "Descriptions of Debezium MongoDB connector data change events"
- Section 4.4, "Setting up MongoDB to work with a Debezium connector"
- Section 4.5, "Deployment of Debezium MongoDB connectors"
- Section 4.6, "Monitoring Debezium MongoDB connector performance"
- Section 4.7, "How Debezium MongoDB connectors handle faults and problems"

4.1. OVERVIEW OF DEBEZIUM MONGODB CONNECTOR

MongoDB's replication mechanism provides redundancy and high availability, and is the preferred way to run MongoDB in production. MongoDB connector captures the changes in a replica set or sharded cluster.

A MongoDB *replica set* consists of a set of servers that all have copies of the same data, and replication ensures that all changes made by clients to documents on the replica set's *primary* are correctly applied to the other replica set's servers, called *secondaries*. MongoDB replication works by having the primary record the changes in its *oplog* (or operation log), and then each of the secondaries reads the primary's oplog and applies in order all of the operations to their own documents. When a new server is added to a replica set, that server first performs an snapshot of all of the databases and collections on the primary, and then reads the primary's oplog to apply all changes that might have been made since it began the snapshot. This new server becomes a secondary (and able to handle queries) when it catches up to the tail of the primary's oplog.

MongoDB connector supports two distinct modes of capturing the changes controlled by the **capture.mode** option:

- oplog based
- change streams based

Oplog capture mode (legacy)

The Debezium MongoDB connector uses the same replication mechanism as described above, though it does not actually become a member of the replica set. Just like MongoDB secondaries, however, the connector always reads the oplog of the replica set's primary. And, when the connector sees a replica set for the first time, it looks at the oplog to get the last recorded transaction and then performs a snapshot of the primary's databases and collections. When all the data is copied, the connector then

starts streaming changes from the position it read earlier from the oplog. Operations in the MongoDB oplog are idempotent, so no matter how many times the operations are applied, they result in the same end state.

The disadvantage of this mode is that only *insert* change events will contain the full document, whereas *update* events only contain a representation of changed fields (i.e. unmodified fields cannot be obtained from an *update* event), and *delete* events contain no representation of the deleted document apart from its key.

This mode should be considered as the legacy one. It is not supported on MongoDB 5 and the user is strongly advised to not use it for MongoDB 4.x server.

Change streams mode

The Debezium MongoDB connector uses a similar replication mechanism to the one described above, though it does not actually become a member of the replica set. The main difference is that the connector does not read the oplog directly, but delegates capturing and decoding the oplog to MongoDB's Change Streams feature. With change streams, the MongoDB server exposes changes to collections as an event stream. The Debezium connector watches the stream and delivers the changes downstream. And, when the connector sees a replica set for the first time, it looks at the oplog to get the last recorded transaction and then performs a snapshot of the primary's databases and collections. When all the data is copied, the connector then creates a change stream from the position it read earlier from the oplog.

Beginning with MongoDB 4.x, change streams is the recommended mode. You cannot use change streams mode with MongoDB 3.x.

WARNING

Both capture modes use different values stored in offsets that allow them to resume streaming from the last position seen after a connector restart. Thus it is not possible to switch from the change streams mode to the oplog mode. To prevent any inadvertent capture mode changes, the connector has a built-in safety check.

When the connector is started it checks the stored offsets. If the original capture mode was oplog-based and the new mode is change streams based, then it will try to migrate to change streams. If the original capture mode was change streams based, it will keep using change streams, also if the new mode is oplog-based, and a warning about this will be emitted to the logs.

As the MongoDB connector processes changes, it periodically records the position in the oplog/stream where the event originated. When the connector stops, it records the last oplog/stream position that it processed, so that upon restart it simply begins streaming from that position. In other words, the connector can be stopped, upgraded or maintained, and restarted some time later, and it will pick up exactly where it left off without losing a single event. Of course, MongoDB's oplogs are usually capped at a maximum size, which means that the connector should not be stopped for too long, or else some of the operations in the oplog might be purged before the connector has a chance to read them. In this case, upon restart the connector will detect the missing oplog operations, perform a snapshot, and then proceed with streaming the changes.

The MongoDB connector is also quite tolerant of changes in membership and leadership of the replica

sets, of additions or removals of shards within a sharded cluster, and network problems that might cause communication failures. The connector always uses the replica set's primary node to stream changes, so when the replica set undergoes an election and a different node becomes primary, the connector will immediately stop streaming changes, connect to the new primary, and start streaming changes using the new primary node. Likewise, if connector experiences any problems communicating with the replica set primary, it will try to reconnect (using exponential backoff so as to not overwhelm the network or replica set) and continue streaming changes from where it last left off. In this way the connector is able to dynamically adjust to changes in replica set membership and to automatically handle communication failures.

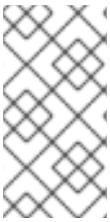
Additional resources

- Replication mechanism
- Replica set
- Replica set elections
- Sharded cluster
- Shard addition
- Shard removal
- Change Streams

4.2. HOW DEBEZIUM MONGODB CONNECTORS WORK

An overview of the MongoDB topologies that the connector supports is useful for planning your application.

When a MongoDB connector is configured and deployed, it starts by connecting to the MongoDB servers at the seed addresses, and determines the details about each of the available replica sets. Since each replica set has its own independent oplog, the connector will try to use a separate task for each replica set. The connector can limit the maximum number of tasks it will use, and if not enough tasks are available the connector will assign multiple replica sets to each task, although the task will still use a separate thread for each replica set.



NOTE

When running the connector against a sharded cluster, use a value of **tasks.max** that is greater than the number of replica sets. This will allow the connector to create one task for each replica set, and will let Kafka Connect coordinate, distribute, and manage the tasks across all of the available worker processes.

The following topics provide details about how the Debezium MongoDB connector works:

- Section 4.2.1, "MongoDB topologies supported by Debezium connectors"
- Section 4.2.2, "How Debezium MongoDB connectors use logical names for replica sets and sharded clusters"
- Section 4.2.3, "How Debezium MongoDB connectors perform snapshots"
- Section 4.2.4, "How the Debezium MongoDB connector streams change event records"

- Section 4.2.5, "Default names of Kafka topics that receive Debezium MongoDB change event records"
- Section 4.2.6, "How event keys control topic partitioning for the Debezium MongoDB connector"
- Section 4.2.7, "Debezium MongoDB connector-generated events that represent transaction boundaries"

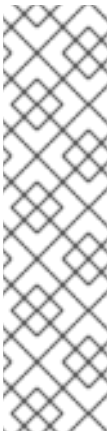
4.2.1. MongoDB topologies supported by Debezium connectors

The MongoDB connector supports the following MongoDB topologies:

MongoDB replica set

The Debezium MongoDB connector can capture changes from a single MongoDB replica set. Production replica sets require a minimum of at least three members.

To use the MongoDB connector with a replica set, provide the addresses of one or more replica set servers as *seed addresses* through the connector's **mongodb.hosts** property. The connector will use these seeds to connect to the replica set, and then once connected will get from the replica set the complete set of members and which member is primary. The connector will start a task to connect to the primary and capture the changes from the primary's oplog. When the replica set elects a new primary, the task will automatically switch over to the new primary.



NOTE

When MongoDB is fronted by a proxy (such as with Docker on OS X or Windows), then when a client connects to the replica set and discovers the members, the MongoDB client will exclude the proxy as a valid member and will attempt and fail to connect directly to the members rather than go through the proxy.

In such a case, set the connector's optional **mongodb.members.auto.discover** configuration property to **false** to instruct the connector to forgo membership discovery and instead simply use the first seed address (specified via the **mongodb.hosts** property) as the primary node. This may work, but still make cause issues when election occurs.

MongoDB sharded cluster

A MongoDB sharded cluster consists of:

- One or more *shards*, each deployed as a replica set;
- A separate replica set that acts as the cluster's *configuration server*
- One or more *routers* (also called **mongos**) to which clients connect and that routes requests to the appropriate shards

  To use the MongoDB connector with a sharded cluster, configure the connector with the host addresses of the *configuration server* replica set. When the connector connects to this replica set, it discovers that it is acting as the configuration server for a sharded cluster, discovers the information about each replica set used as a shard in the cluster, and will then start up a separate task to capture the changes from each replica set. If new shards are added to the cluster or existing shards removed, the connector will automatically adjust its tasks accordingly.

MongoDB standalone server

The MongoDB connector is not capable of monitoring the changes of a standalone MongoDB server, since standalone servers do not have an oplog. The connector will work if the standalone server is converted to a replica set with one member.

NOTE

MongoDB does not recommend running a standalone server in production. For more information, see the MongoDB documentation.

4.2.2. How Debezium MongoDB connectors use logical names for replica sets and sharded clusters

The connector configuration property **mongodb.name** serves as a *logical name* for the MongoDB replica set or sharded cluster. The connector uses the logical name in a number of ways: as the prefix for all topic names, and as a unique identifier when recording the oplog/change stream position of each replica set.

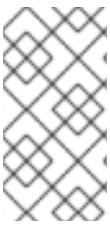
You should give each MongoDB connector a unique logical name that meaningfully describes the source MongoDB system. We recommend logical names begin with an alphabetic or underscore character, and remaining characters that are alphanumeric or underscore.

4.2.3. How Debezium MongoDB connectors perform snapshots

When a task starts up using a replica set, it uses the connector's logical name and the replica set name to find an *offset* that describes the position where the connector previously stopped reading changes. If an offset can be found and it still exists in the oplog, then the task immediately proceeds with streaming changes, starting at the recorded offset position.

However, if no offset is found or if the oplog no longer contains that position, the task must first obtain the current state of the replica set contents by performing a *snapshot*. This process starts by recording the current position of the oplog and recording that as the offset (along with a flag that denotes a snapshot has been started). The task will then proceed to copy each collection, spawning as many threads as possible (up to the value of the **snapshot.max.threads** configuration property) to perform this work in parallel. The connector will record a separate *read event* for each document it sees, and that read event will contain the object's identifier, the complete state of the object, and *source* information about the MongoDB replica set where the object was found. The source information will also include a flag that denotes the event was produced during a snapshot.

This snapshot will continue until it has copied all collections that match the connector's filters. If the connector is stopped before the tasks' snapshots are completed, upon restart the connector begins the snapshot again.



NOTE

Try to avoid task reassignment and reconfiguration while the connector performs snapshots of any replica sets. The connector generates log messages to report on the progress of the snapshot. To provide for the greatest control, run a separate Kafka Connect cluster for each connector.

4.2.3.1. Ad hoc snapshots

IMPORTANT

Ad hoc snapshots are a Technology Preview feature for the Debezium MongoDB connector. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about support scope, see Technology Preview Features Support Scope.

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing collection data, Debezium includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of collections.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a collection for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling collections. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling collection.

When you initiate an ad hoc snapshot of an existing collection, the connector appends content to the topic that already exists for the collection. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the collections to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the collections in the database.

You specify the collections to capture by sending an **execute-snapshot** message to the signaling collection. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the collections to include in the snapshot, as described in the following table:

Table 4.1. Example of an ad hoc`execute-snapshot` signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.
data-collections	*N/A*	An array that contains the fully-qualified names of the collection to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling collection. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each collection. Based on the number of entries in the collection, and the configured chunk size, Debezium divides the collection into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

4.2.3.2. Incremental snapshots

IMPORTANT

Incremental snapshots are a Technology Preview feature for the Debezium MongoDB connector. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about support scope, see Technology Preview Features Support Scope.

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each collection in phases, in a series of configurable chunks. You can specify the collections that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each collection row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the other.
- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the collection from the beginning.
- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a collection to its **collection.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each collection by primary key and then splits the collection into chunks based on the configured chunk size. Working chunk by chunk, it then captures

each collection row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying collection records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a collection row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same collection row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified collection chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the collection's Kafka topic.

The connector repeats the process for each snapshot chunk.

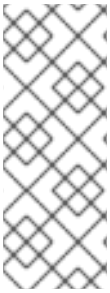
Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling collection on the source database. You submit signals to the collection as SQL **INSERT** queries. After Debezium detects the change in the signaling collection, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the collections to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the collections to include in the snapshot, provide a **data-collections** array that lists the collections, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.



NOTE

If the name of a collection that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the collection to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling collection:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling collection.

The following collection describes the these parameters:

Table 4.2. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling collection

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling collection on the source database

Value	Description
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling collection. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of collection names to include in the snapshot. The array lists collections by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling collection in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling collection is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling collection is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

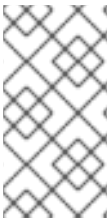
NOTE

Incremental snapshots are currently supported for single replica set deployments only.

4.2.4. How the Debezium MongoDB connector streams change event records

After the connector task for a replica set records an offset, it uses the offset to determine the position in the oplog where it should start streaming changes. The task then (depending on the configuration) either connects to the replica set's primary node or connects to a replica-set-wide change stream and starts streaming changes from that position. It processes all of create, insert, and delete operations, and converts them into Debezium change events. Each change event includes the position in the oplog where the operation was found, and the connector periodically records this as its most recent offset. The interval at which the offset is recorded is governed by **offset.flush.interval.ms**, which is a Kafka Connect worker configuration property.

When the connector is stopped gracefully, the last offset processed is recorded so that, upon restart, the connector will continue exactly where it left off. If the connector's tasks terminate unexpectedly, however, then the tasks may have processed and generated events after it last records the offset but before the last offset is recorded; upon restart, the connector begins at the last *recorded* offset, possibly generating some the same events that were previously generated just prior to the crash.



NOTE

When everything is operating nominally, Kafka consumers will actually see every message ***exactly once***. However, when things go wrong Kafka can only guarantee consumers will see every message ***at least once***. Therefore, your consumers need to anticipate seeing messages more than once.

As mentioned above, the connector tasks always use the replica set's primary node to stream changes from the oplog, ensuring that the connector sees the most up-to-date operations as possible and can capture the changes with lower latency than if secondaries were to be used instead. When the replica set elects a new primary, the connector immediately stops streaming changes, connects to the new primary, and starts streaming changes from the new primary node at the same position. Likewise, if the connector experiences any problems communicating with the replica set members, it tries to reconnect, by using exponential backoff so as to not overwhelm the replica set, and once connected it continues streaming changes from where it last left off. In this way, the connector is able to dynamically adjust to changes in replica set membership and automatically handle communication failures.

To summarize, the MongoDB connector continues running in most situations. Communication problems might cause the connector to wait until the problems are resolved.

4.2.5. Default names of Kafka topics that receive Debezium MongoDB change event records

The MongoDB connector writes events for all insert, update, and delete operations to documents in each collection to a single Kafka topic. The name of the Kafka topics always takes the form *logicalName.databaseName.collectionName*, where *logicalName* is the logical name of the connector as specified with the **mongodb.name** configuration property, *databaseName* is the name of the database where the operation occurred, and *collectionName* is the name of the MongoDB collection in which the affected document existed.

For example, consider a MongoDB replica set with an **inventory** database that contains four collections: **products**, **products_on_hand**, **customers**, and **orders**. If the connector monitoring this database were given a logical name of **fulfillment**, then the connector would produce events on these four Kafka topics:

- **fulfillment.inventory.products**
- **fulfillment.inventory.products_on_hand**
- **fulfillment.inventory.customers**
- **fulfillment.inventory.orders**

Notice that the topic names do not incorporate the replica set name or shard name. As a result, all changes to a sharded collection (where each shard contains a subset of the collection's documents) all go to the same Kafka topic.

You can set up Kafka to auto-create the topics as they are needed. If not, then you must use Kafka administration tools to create the topics before starting the connector.

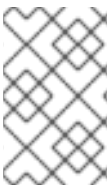
4.2.6. How event keys control topic partitioning for the Debezium MongoDB connector

The MongoDB connector does not make any explicit determination about how to partition topics for events. Instead, it allows Kafka to determine how to partition topics based on event keys. You can change Kafka's partitioning logic by defining the name of the **Partitioner** implementation in the Kafka Connect worker configuration.

Kafka maintains total order only for events written to a single topic partition. Partitioning the events by key does mean that all events with the same key always go to the same partition. This ensures that all events for a specific document are always totally ordered.

4.2.7. Debezium MongoDB connector-generated events that represent transaction boundaries

Debezium can generate events that represents transaction metadata boundaries and enrich change data event messages.



LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

For every transaction **BEGIN** and **END**, Debezium generates an event that contains the following fields:

status

BEGIN or **END**

id

String representation of unique transaction identifier.

event_count (for **END** events)

Total number of events emitted by the transaction.

data_collections (for **END** events)

An array of pairs of **data_collection** and **event_count** that provides number of events emitted by changes originating from given data collection.

The following example shows a typical message:

```
{
  "status": "BEGIN",
  "id": "1462833718356672513",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "1462833718356672513",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "rs0.testDB.collectiona",
      "event_count": 1
    },
    {
      "data_collection": "rs0.testDB.collectionb",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, transaction events are written to the topic named ***database.server.name*.transaction**.

Change data event enrichment

When transaction metadata is enabled, the data message **Envelope** is enriched with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

id

String representation of unique transaction identifier.

total_order

The absolute position of the event among all events generated by the transaction.

data_collection_order

The per-data collection position of the event among all events that were emitted by the transaction.

Following is an example of what a message looks like:

```
{
  "patch": null,
  "after": "{\"_id\" : {\"$numberLong\" : \"1004\"},\"first_name\" : \"Anne\",\"last_name\" :
\"Kretchmar\",\"email\" : \"annek@noanswer.org\"}",
  "source": {
...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "1460833718356672513",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

4.3. DESCRIPTIONS OF DEBEZIUM MONGODB CONNECTOR DATA CHANGE EVENTS

The Debezium MongoDB connector generates a data change event for each document-level operation that inserts, updates, or deletes data. Each event contains a key and a value. The structure of the key and the value depends on the collection that was changed.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events may change over time, which can be difficult for consumers to handle. To address this, each event contains the schema for its content or, if you are using a schema registry, a schema ID that a consumer can use to obtain the schema from the registry. This makes each event self-contained.

The following skeleton JSON shows the basic four parts of a change event. However, how you configure the Kafka Connect converter that you choose to use in your application determines the representation of these four parts in change events. A **schema** field is in a change event only when you configure the converter to produce it. Likewise, the event key and event payload are in a change event only if you configure a converter to produce it. If you use the JSON converter and you configure it to produce all four basic change event parts, change events have this structure:

```
{
 "schema": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

Table 4.3. Overview of change event basic content

Item	Field name	Description
1	**schema**	The first **schema** field is part of the event key. It specifies a Kafka Connect schema that describes what is in the event key's **payload** portion. In other words, the first **schema** field describes the structure of the key for the document that was changed.
2	**payload**	The first **payload** field is part of the event key. It has the structure described by the previous **schema** field and it contains the key for the document that was changed.
3	**schema**	The second **schema** field is part of the event value. It specifies the Kafka Connect schema that describes what is in the event value's **payload** portion. In other words, the second **schema** describes the structure of the document that was changed. Typically, this schema contains nested schemas.
4	**payload**	The second **payload** field is part of the event value. It has the structure described by the previous **schema** field and it contains the actual data for the document that was changed.

By default, the connector streams change event records to topics with names that are the same as the event's originating collection. See topic names.

WARNING

The MongoDB connector ensures that all Kafka Connect schema names adhere to the Avro schema name format. This means that the logical server name must start with a Latin letter or an underscore, that is, a-z, A-Z, or _. Each remaining character in the logical server name and each character in the database and collection names must be a Latin letter, a digit, or an underscore, that is, a-z, A-Z, 0-9, or _. If there is an invalid character it is replaced with an underscore character.

This can lead to unexpected conflicts if the logical server name, a database name, or a collection name contains invalid characters, and the only characters that distinguish names from one another are invalid and thus replaced with underscores.

For more information, see the following topics:

- Section 4.3.1, "About keys in Debezium MongoDB change events"
- Section 4.3.2, "About values in Debezium MongoDB change events"

4.3.1. About keys in Debezium MongoDB change events

A change event's key contains the schema for the changed document's key and the changed document's actual key. For a given collection, both the schema and its corresponding payload contain a single **id** field. The value of this field is the document's identifier represented as a string that is derived

from MongoDB extended JSON serialization strict mode .

Consider a connector with a logical name of **fulfillment**, a replica set containing an **inventory** database, and a **customers** collection that contains documents such as the following.

Example document

```
{
  "_id": 1004,
  "first_name": "Anne",
  "last_name": "Kretchmar",
  "email": "annek@noanswer.org"
}
```

Example change event key

Every change event that captures a change to the **customers** collection has the same event key schema. For as long as the **customers** collection has the previous definition, every change event that captures a change to the **customers** collection has the following key structure. In JSON, it looks like this:

```
{
  "schema": { 1
    "type": "struct",
    "name": "fulfillment.inventory.customers.Key", 2
    "optional": false, 3
    "fields": [ 4
      {
        "field": "id",
        "type": "string",
        "optional": false
      }
    ]
  },
  "payload": { 5
    "id": "1004"
  }
}
```

Table 4.4. Description of change event key

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's **payload** portion.

Item	Field name	Description
2	**fulfillment.inventory.customers.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the key for the document that was changed. Key schema names have the format *connector-name.database-name.collection-name*.**Key**. In this example: • **fulfillment** is the name of the connector that generated this event. • **inventory** is the database that contains the collection that was changed. • **customers** is the collection that contains the document that was updated.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a document does not have a key.
4	**fields**	Specifies each field that is expected in the **payload**, including each field's name, type, and whether it is required.
5	**payload**	Contains the key for the document for which this change event was generated. In this example, the key contains a single **id** field of type **string** whose value is **1004**.

This example uses a document with an integer identifier, but any valid MongoDB document identifier works the same way, including a document identifier. For a document identifier, an event key's **payload.id** value is a string that represents the updated document's original **_id** field as a MongoDB extended JSON serialization that uses strict mode. The following table provides examples of how different types of **_id** fields are represented.

Table 4.5. Examples of representing document_id fields in event key payloads

Type	MongoDB **_id** Value	Key's payload
Integer	1234	**{ "id" : "1234" }**
Float	12.34	**{ "id" : "12.34" }**
String	"1234"	**{ "id" : "\"1234\"" }**
Document	**{ "hi" : "kafka", "nums" : [10.0, 100.0, 1000.0] }**	**{ "id" : "{\"hi\" : \"kafka\", \"nums\" : [10.0, 100.0, 1000.0]}" }**
ObjectId	**ObjectId("596e275826f08b2730779e1f")**	**{ "id" : "{\"$oid\" : \"596e275826f08b2730779e1f\"}" }**

Type	MongoDB _id Value	Key's payload
Binary	**BinData("a2Fma2E=",0)**	**{ "id" : "{\"$binary\" : \"a2Fma2E=\", \"$type\" : \"00\"}" }**

4.3.2. About values in Debezium MongoDB change events

The value in a change event is a bit more complicated than the key. Like the key, the value has a **schema** section and a **payload** section. The **schema** section contains the schema that describes the **Envelope** structure of the **payload** section, including its nested fields. Change events for operations that create, update or delete data all have a value payload with an envelope structure.

Consider the same sample document that was used to show an example of a change event key:

Example document

```
{
  "_id": 1004,
  "first_name": "Anne",
  "last_name": "Kretchmar",
  "email": "annek@noanswer.org"
}
```

The value portion of a change event for a change to this document is described for each event type:

- *create* events
- *update* events
- *delete* events
- Tombstone events

create events

The following example shows the value portion of a change event that the connector generates for an operation that creates data in the **customers** collection:

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "name": "io.debezium.data.Json", 2
        "version": 1,
        "field": "after"
      },
      {
        "type": "string",
        "optional": true,
        "name": "io.debezium.data.Json",
```

```
      "version": 1,
      "field": "patch"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "string",
          "optional": false,
          "field": "version"
        },
        {
          "type": "string",
          "optional": false,
          "field": "connector"
        },
        {
          "type": "string",
          "optional": false,
          "field": "name"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "ts_ms"
        },
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "string",
          "optional": false,
          "field": "db"
        },
        {
          "type": "string",
          "optional": false,
          "field": "rs"
        },
        {
          "type": "string",
          "optional": false,
          "field": "collection"
        },
        {
          "type": "int32",
          "optional": false,
          "field": "ord"
        },
        {
          "type": "int64",
          "optional": true,
          "field": "h"
```

```
            }
           ],
           "optional": false,
           "name": "io.debezium.connector.mongo.Source", 3
           "field": "source"
          },
          {
           "type": "string",
           "optional": true,
           "field": "op"
          },
          {
           "type": "int64",
           "optional": true,
           "field": "ts_ms"
          }
         ],
         "optional": false,
         "name": "dbserver1.inventory.customers.Envelope" 4
        },
        "payload": { 5
         "after": "{\"_id\" : {\"$numberLong\" : \"1004\"},\"first_name\" : \"Anne\",\"last_name\" :
\"Kretchmar\",\"email\" : \"annek@noanswer.org\"}", 6
         "patch": null,
         "source": { 7
           "version": "1.9.7.Final",
           "connector": "mongodb",
           "name": "fulfillment",
           "ts_ms": 1558965508000,
           "snapshot": false,
           "db": "inventory",
           "rs": "rs0",
           "collection": "customers",
           "ord": 31,
           "h": 1546547425148721999
         },
         "op": "c", 8
         "ts_ms": 1558965515240 9
        }
      }
```

Table 4.6. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular collection.

Item	Field name	Description
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **io.debezium.data.Json** is the schema for the payload's **after**, **patch**, and **filter** fields. This schema is specific to the **customers** collection. A *create* event is the only kind of event that contains an **after** field. An *update* event contains a **filter** field and a **patch** field. A *delete* event contains a **filter** field, but not an **after** field nor a **patch** field.
3	**name**	**io.debezium.connector.mongo.Source** is the schema for the payload's **source** field. This schema is specific to the MongoDB connector. The connector uses it for all events that it generates.
4	**name**	**dbserver1.inventory.customers.Envelope** is the schema for the overall structure of the payload, where **dbserver1** is the connector name, **inventory** is the database, and **customers** is the collection. This schema is specific to the collection.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the documents they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**after**	An optional field that specifies the state of the document after the event occurred. In this example, the **after** field contains the values of the new document's **_id**, **first_name**, **last_name**, and **email** fields. The **after** value is always a string. By convention, it contains a JSON representation of the document. MongoDB's oplog entries contain the full state of a document only for _create_ events and also for **update** events, when the **capture.mode** option is set to **change_streams_update_full**; in other words, a *create* event is the only kind of event that contains an *after* field, when the **capture.mode** option is set either to **oplog** or **change_streams**.

Item	Field name	Description
7	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: ● Debezium version. ● Name of the connector that generated the event. ● Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. ● Names of the collection and database that contain the new document. ● If the event was part of a snapshot. ● Timestamp for when the change was made in the database and ordinal of the event within the timestamp. ● Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the **h** field in the oplog event, or a field named **stxnid**, which represents the **lsid** and **txnNumber** fields from the oplog event (oplog capture mode only). ● Unique identifiers of the MongoDB session **lsid** and transaction number **txnNumber** in case the change was executed inside a transaction (change streams capture mode only).
8	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a document. Valid values are: ● **c** = create ● **u** = update ● **d** = delete ● **r** = read (applies to only snapshots)
9	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

Oplog capture mode (legacy)

The value of a change event for an update in the sample **customers** collection has the same schema as a *create* event for that collection. Likewise, the event value's payload has the same structure. However, the event value payload contains different values in an *update* event. An *update* event does not have an **after** value. Instead, it has these two fields:

- **patch** is a string field that contains the JSON representation of the idempotent update operation
- **filter** is a string field that contains the JSON representation of the selection criteria for the update. The **filter** string can include multiple shard key fields for sharded collections.

Here is an example of a change event value in an event that the connector generates for an update in the **customers** collection:

```
{
  "schema": { ... },
  "payload": {
    "op": "u", (1)
    "ts_ms": 1465491461815, (2)
    "patch": "{\"$set\":{\"first_name\":\"Anne Marie\"}}", (3)
    "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", (4)
    "source": { (5)
      "version": "1.9.7.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": false,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 6,
      "h": 1546547425148721999
    }
  }
}
```

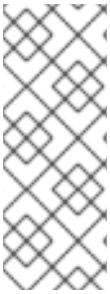
Table 4.7. Descriptions of *update* event value fields

Item	Field name	Description
1	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **u** indicates that the operation updated a document.
2	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

Item	Field name	Description
3	**patch**	Contains the JSON string representation of the actual MongoDB idempotent change to the document. In this example, the update changed the **first_name** field to a new value. An *update* event value does not contain an**after** field.
4	**filter**	Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be updated.
5	**source**	Mandatory field that describes the source metadata for the event. This field contains the same information as a *create* event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes: ● Debezium version. ● Name of the connector that generated the event. ● Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. ● Names of the collection and database that contain the updated document. ● If the event was part of a snapshot. ● Timestamp for when the change was made in the database and ordinal of the event within the timestamp. ● Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the **h** field in the oplog event, or a field named **stxnid**, which represents the**lsid** and **txnNumber** fields from the oplog event.

WARNING

In a Debezium change event, MongoDB provides the content of the **patch** field. The format of this field depends on the version of the MongoDB database. Consequently, be prepared for potential changes to the format when you upgrade to a newer MongoDB database version. Examples in this document were obtained from MongoDB 3.4, In your application, event formats might be different.



NOTE

In MongoDB's oplog, *update* events do not contain the *before* or *after* states of the changed document. Consequently, it is not possible for a Debezium connector to provide this information. However, a Debezium connector provides a document's starting state in *create* and *read* events. Downstream consumers of the stream can reconstruct document state by keeping the latest state for each document and comparing the state in a new event with the saved state. Debezium connector's are not able to keep this state.

Change streams capture mode

The value of a change event for an update in the sample **customers** collection has the same schema as a *create* event for that collection. Likewise, the event value's payload has the same structure. However, the event value payload contains different values in an *update* event. An *update* event does have an **after** value only if the **capture.mode** option is set to **change_streams_update_full**. There is a new structured field **updateDescription** with a few additional fields in this case:

- **updatedFields** is a string field that contains the JSON representation of the updated document fields with their values
- **removedFields** is a list of field names that were removed from the document
- **truncatedArrays** is a list of arrays in the document that were truncated

Here is an example of a change event value in an event that the connector generates for an update in the **customers** collection:


```
{
   "schema": { ... },
   "payload": {
     "op": "u", (1)
     "ts_ms": 1465491461815, (2)
     "after":"{\"_id\": {\"$numberLong\": \"1004\"},\"first_name\": \"Anne Marie\",\"last_name\":
\"Kretchmar\",\"email\": \"annek@noanswer.org\"}", (3)
     "updateDescription": {
       "removedFields": null,
       "updatedFields": "{\"first_name\": \"Anne Marie\"}", (4)
       "truncatedArrays": null
     },
     "source": { (5)
       "version": "1.9.7.Final",
       "connector": "mongodb",
       "name": "fulfillment",
       "ts_ms": 1558965508000,
       "snapshot": false,
       "db": "inventory",
       "rs": "rs0",
       "collection": "customers",
       "ord": 1,
       "h": null,
       "tord": null,
       "stxnid": null,
       "lsid":"{\"id\": {\"$binary\": \"FA7YEzXgQXSX9OxmzlIH2w==\",\"$type\": \"04\"},\"uid\":
{\"$binary\": \"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=\",\"$type\": \"00\"}}",
       "txnNumber":1
```

```
      }
    }
  }
```

Table 4.8. Descriptions of *update* event value fields

Item	Field name	Description
1	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **u** indicates that the operation updated a document.
2	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
3	**after**	Contains the JSON string representation of the actual MongoDB document. An *update* event value does not contain an **after** field if the capture mode is not set to **change_streams_update_full**
4	**updatedFields**	Contains the JSON string representation of the updated field values of the document. In this example, the update changed the **first_name** field to a new value.
5	**source**	Mandatory field that describes the source metadata for the event. This field contains the same information as a *create* event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes: • Debezium version. • Name of the connector that generated the event. • Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. • Names of the collection and database that contain the updated document. • If the event was part of a snapshot. • Timestamp for when the change was made in the database and ordinal of the event within the timestamp. • Unique identifiers of the MongoDB session **lsid** and transaction number **txnNumber** in case the change was executed inside a transaction.

WARNING

The **after** value in the event should be handled as the at-point-of-time value of the document. The value is not calculated dynamically but is obtained from the collection. It is thus possible if multiple updates are closely following one after the other, that all *update* updates events will contain the same **after** value which will be representing the last value stored in the document.

If your application depends on gradual change evolution then you should rely on **updateDescription** only.

delete events

The value in a *delete* change event has the same **schema** portion as *create* and *update* events for the same collection. The **payload** portion in a *delete* event contains values that are different from *create* and *update* events for the same collection. In particular, a *delete* event contains neither an **after** value nor a **patch** or **updateDescription** values. Here is an example of a *delete* event for a document in the **customers** collection:

```
{
    "schema": { ... },
    "payload": {
      "op": "d", 1
      "ts_ms": 1465495462115, 2
      "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", 3
      "source": { 4
        "version": "1.9.7.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": true,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

Table 4.9. Descriptions of *delete* event value fields

Item	Field name	Description
1	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this document was deleted.

Item	Field name	Description
2	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
3	**filter**	Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be deleted (oplog capture mode only).
4	**source**	Mandatory field that describes the source metadata for the event. This field contains the same information as a *create* or *update* event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes: ● Debezium version. ● Name of the connector that generated the event. ● Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. ● Names of the collection and database that contained the deleted document. ● If the event was part of a snapshot. ● Timestamp for when the change was made in the database and ordinal of the event within the timestamp. ● Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the **h** field in the oplog event, or a field named **stxnid**, which represents the **lsid** and **txnNumber** fields from the oplog event (oplog capture mode only). ● Unique identifiers of the MongoDB session **lsid** and transaction number **txnNumber** in case the change was executed inside a transaction (change streams capture mode only).

MongoDB connector events are designed to work with Kafka log compaction. Log compaction enables removal of some older messages as long as at least the most recent message for every key is kept. This lets Kafka reclaim storage space while ensuring that the topic contains a complete data set and can be used for reloading key-based state.

Tombstone events

All MongoDB connector events for a uniquely identified document have exactly the same key. When a document is deleted, the *delete* event value still works with log compaction because Kafka can remove all earlier messages that have that same key. However, for Kafka to remove all messages that have that

key, the message value must be **null**. To make this possible, after Debezium's MongoDB connector emits a *delete* event, the connector emits a special tombstone event that has the same key but a **null** value. A tombstone event informs Kafka that all messages with that same key can be removed.

4.4. SETTING UP MONGODB TO WORK WITH A DEBEZIUM CONNECTOR

The MongoDB connector uses MongoDB's oplog/change streams to capture the changes, so the connector works only with MongoDB replica sets or with sharded clusters where each shard is a separate replica set. See the MongoDB documentation for setting up a replica set or sharded cluster. Also, be sure to understand how to enable access control and authentication with replica sets.

You must also have a MongoDB user that has the appropriate roles to read the **admin** database where the oplog can be read. Additionally, the user must also be able to read the **config** database in the configuration server of a sharded cluster and must have **listDatabases** privilege action. When change streams are used (the default) the user also must have cluster-wide privilege actions **find** and **changeStream**.

4.5. DEPLOYMENT OF DEBEZIUM MONGODB CONNECTORS

You can use either of the following methods to deploy a Debezium MongoDB connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .

Additional resources

- Section 4.5.5, "Descriptions of Debezium MongoDB connector configuration properties"

4.5.1. MongoDB connector deployment using AMQ Streams

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.

NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

4.5.2. Using AMQ Streams to deploy a Debezium MongoDB connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.
- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

Example 4.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

In the example that follows, the custom resource is configured to download the following artifacts:

- The Debezium MongoDB connector archive.
- The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
- The Debezium scripting SMT archive and the associated scripting engine that you want to use with the Debezium connector. The SMT archive and scripting language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-mongodb
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mongodb/1.9.7.Final-redhat-<build_number>/debezium-connector-mongodb-1.9.7.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip (8)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip (9)
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar (10)
          - type: jar
```

```
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
jsr223/3.0.11/groovy-jsr223-3.0.11.jar
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
json3.0.11/groovy-json-3.0.11.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 4.10. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry such as Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. The **type** value must match the type of the file that is referenced in the **url** field.
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.

Item	Description
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) As an alternative, the Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **mongodb-inventory-connector.yaml**

Example 4.2. mongodb-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
```

```
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-mongodb (1)
spec:
  class: io.debezium.connector.mongodb.MongoDbConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: mongodb.debezium-mongodb.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_mongodb (10)
    database.include.list: public.inventory (11)
```

Table 4.11. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.
4	The connector's configuration.
5	The address of the host database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.
8	The password for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.

Item	Description
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium MongoDB deployment.

4.5.3. Deploying a Debezium MongoDB connector by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium MongoDB connector, you must build a custom Kafka Connect container image that contains the Debezium connector archive and then push this container image to a container registry. You then create two custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.
- A **KafkaConnector** CR that defines your Debezium MongoDB connector. Apply this CR to the same OpenShift instance where you apply the **KafkaConnect** CR.

Prerequisites

- MongoDB is running and you completed the steps to set up MongoDB to work with a Debezium connector.
- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift .
- Podman or Docker is installed.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium MongoDB container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-mongodb.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mongodb/1.9.7.Final-redhat-<build_number>/debezium-connector-mongodb-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-mongodb-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-mongodb-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

The command creates a Dockerfile with the name **debezium-container-for-mongodb.yaml** in the current directory.

b. Build the container image from the **debezium-container-for-mongodb.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-mongodb:latest .
```

```
docker build -t debezium-container-for-mongodb:latest .
```

The preceding commands build a container image with the name **debezium-container-for-mongodb**.

c. Push your custom image to a container registry, such as **quay.io** or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-mongodb:latest
```

```
docker push <myregistry.io>/debezium-container-for-mongodb:latest
```

d. Create a new Debezium MongoDB **KafkaConnect** custom resource (CR). For example, create a **KafkaConnect** CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
```

```
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  #...
  image: debezium-container-for-mongodb (2)
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply the **KafkaConnect** CR to the OpenShift Kafka Connect environment by entering the following command:

```
oc create -f dbz-connect.yaml
```

The command adds a Kafka Connect instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium MongoDB connector instance.
You configure a Debezium MongoDB connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce change events for a subset of MongoDB replica sets or sharded clusters. Optionally, you can set properties that filter out collections that are not needed.

The following example configures a Debezium connector that connects to a MongoDB replica set **rs0** at port **27017** on **192.168.99.100**, and captures changes that occur in the **inventory** collection. **fullfillment** is the logical name of the replica set.

MongoDB inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
   name: inventory-connector (1)
   labels: strimzi.io/cluster: my-connect-cluster
 spec:
   class: io.debezium.connector.mongodb.MongoDbConnector (2)
   config:
    mongodb.hosts: rs0/192.168.99.100:27017 (3)
    mongodb.name: fulfillment (4)
    collection.include.list: inventory[.]* (5)
```

The name that is used to register the connector with Kafka Connect.

The name of the MongoDB connector class.

The host addresses to use to connect to the MongoDB replica set.

The *logical name* of the MongoDB replica set, which forms a namespace for generated events and is used in all the names of the Kafka topics to which the connector writes, the Kafka Connect schema names, and the namespaces of the corresponding Avro schema when the Avro converter is used.

An optional list of regular expressions that match the collection namespaces (for example, <dbName>.<collectionName>) of all collections to be monitored.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **inventory-connector.yaml** file, you would run the following command:

```
oc apply -f inventory-connector.yaml
```

The preceding command registers **inventory-connector** and the connector starts to run against the **inventory** collection as defined in the **KafkaConnector** CR.

For the complete list of the configuration properties that you can set for the Debezium MongoDB connector, see MongoDB connector configuration properties.

Results

After the connector starts, it completes the following actions:

- Performs a consistent snapshot of the collections in your MongoDB replica sets.
- Reads the oplogs/change streams for the replica sets.
- Produces change events for every inserted, updated, and deleted document.
- Streams change event records to Kafka topics.

4.5.4. Verifying that the Debezium MongoDB connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.

- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-mongodb**.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
     a. Enter the following command:

```
oc describe KafkaConnector <connector-name> -n <project>
```

     For example,

```
oc describe KafkaConnector inventory-connector-mongodb -n debezium
```

     The command returns status information that is similar to the following output:

Example 4.3. KafkaConnector resource status

```
Name:         inventory-connector-mongodb
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
```

```
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  10.131.1.124:8083
    Name:         inventory-connector-mongodb
    Tasks:
      Id:               0
      State:            RUNNING
      worker_id:        10.131.1.124:8083
    Type:               source
  Observed Generation:  1
  Tasks Max:            1
  Topics:
    inventory_connector_mongodb
    inventory_connector_mongodb.inventory.addresses
    inventory_connector_mongodb.inventory.customers
    inventory_connector_mongodb.inventory.geom
    inventory_connector_mongodb.inventory.orders
    inventory_connector_mongodb.inventory.products
    inventory_connector_mongodb.inventory.products_on_hand
  Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-mongodb.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

   - From a terminal window:

     a. Enter the following command:

     ```
     oc get kafkatopics
     ```

     The command returns status information that is similar to the following output:

     Example 4.4. KafkaTopic resource status

     ```
     NAME                                                                CLUSTER
     PARTITIONS   REPLICATION FACTOR   READY
     connect-cluster-configs                                              debezium-
     kafka-cluster   1            1                True
     connect-cluster-offsets                                              debezium-
     ```

```
kafka-cluster   25          1                   True
connect-cluster-status                                                    debezium-
kafka-cluster   5           1                   True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
debezium-kafka-cluster   50          1                   True
inventory-connector-mongodb---a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster   1           1                   True
inventory-connector-mongodb.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480          debezium-kafka-cluster
1           1                   True
inventory-connector-mongodb.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1                   True
inventory-connector-mongodb.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5              debezium-kafka-cluster
1           1                   True
inventory-connector-mongodb.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d             debezium-kafka-cluster
1           1                   True
inventory-connector-mongodb.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1                   True
inventory-connector-mongodb.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1                   True
schema-changes.inventory
debezium-kafka-cluster   1           1                   True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1           1                   True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1                   True
```

3. Check topic content.

   - From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

   For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_mongodb.inventory.products_on_hand
```

   The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_mongodb.inventory.addresses**.

For each event in the topic, the command returns information that is similar to the following output:

Example 4.5. Content of a Debezium change event


```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_conne
ctor_mongodb.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_mongodb.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_mongodb.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connecto
r.mongodb.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transacti
on"}],"optional":false,"name":"inventory_connector_mongodb.inventory.products_on_hand.
Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.7.Final-redhat-
00001","connector":"mongodb","name":"inventory_connector_mongodb","ts_ms":16389852
47805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","serve
r_id":0,"gtid":null,"file":"mongodb-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}
```


In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

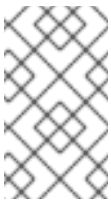
4.5.5. Descriptions of Debezium MongoDB connector configuration properties

The Debezium MongoDB connector has numerous configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values. Information about the properties is organized as follows:

- Required Debezium MongoDB connector configuration properties
- Advanced Debezium MongoDB connector configuration properties

The following configuration properties are *required* unless a default value is available.

Table 4.12. Required Debezium MongoDB connector configuration properties

Property	Default	Description
name		Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.mongodb.MongoDbConnector** for the MongoDB connector.
mongodb.hosts		The comma-separated list of hostname and port pairs (in the form 'host' or 'host:port') of the MongoDB servers in the replica set. The list can contain a single hostname and port pair. If **mongodb.members.auto.discover** is set to **false**, then the host and port pair should be prefixed with the replica set name (e.g., **rs0/localhost:27017**). +  **NOTE** It is mandatory to provide the current primary address. This limitation will be removed in the next Debezium release.

Property	Default	Description
mongodb.name		A unique name that identifies the connector and/or MongoDB replica set or sharded cluster that this connector monitors. Each server should be monitored by at most one Debezium connector, since this server name prefixes all persisted Kafka topics emanating from the MongoDB replica set or cluster. Use only alphanumeric characters, hyphens, dots and underscores to form the name. The logical name should be unique across all other connectors, because the name is used as the prefix in naming the Kafka topics that receive records from this connector. + **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value.
mongodb.user		Name of the database user to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.
mongodb.password		Password to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.
mongodb.authsource	**admin**	Database (authentication source) containing MongoDB credentials. This is required only when MongoDB is configured to use authentication with another authentication database than **admin**.

Property	Default	Description
mongodb.ssl.enabled	**false**	Connector will use SSL to connect to MongoDB instances.
mongodb.ssl.invalid.hostname.allowed	**false**	When SSL is enabled this setting controls whether strict hostname checking is disabled during connection phase. If **true** the connection will not prevent man-in-the-middle attacks.
database.include.list	*empty string*	An optional comma-separated list of regular expressions that match database names to be monitored; any database name not included in **database.include.list** is excluded from monitoring. By default all databases are monitored. Must not be used with **database.exclude.list**.
database.exclude.list	*empty string*	An optional comma-separated list of regular expressions that match database names to be excluded from monitoring; any database name not included in **database.exclude.list** is monitored. Must not be used with **database.include.list**.
collection.include.list	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified namespaces for MongoDB collections to be monitored; any collection not included in **collection.include.list** is excluded from monitoring. Each identifier is of the form *databaseName.collectionName*. By default the connector will monitor all collections except those in the **local** and **admin** databases. Must not be used with **collection.exclude.list**.
collection.exclude.list	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified namespaces for MongoDB collections to be excluded from monitoring; any collection not included in **collection.exclude.list** is monitored. Each identifier is of the form *databaseName.collectionName*. Must not be used with **collection.include.list**.

Property	Default	Description
snapshot.mode	**initial**	Specifies the criteria for running a snapshot upon startup of the connector. The default is **initial**, and specifies that the connector reads a snapshot when either no offset is found or if the oplog/change stream no longer contains the previous offset. The **never** option specifies that the connector should never use snapshots, instead the connector should proceed to tail the log.
capture.mode	**change_streams_update_full**	Specifies the method used to capture changes from the MongoDB server. The default is **change_streams_update_full**, and specifies that the connector captures changes via MongoDB Change Streams mechanism, and that *update* events should contain the full document. The **change_streams** mode will use the same capturing method, but *update* events won't contain the full document. The **oplog** mode specifies that the MongoDB oplog will be accessed directly; this is the legacy method and should not be used for new connector instances.
snapshot.include.collection.list	All collections specified in **collection.include.list**	An optional, comma-separated list of regular expressions that match names of schemas specified in **collection.include.list** for which you **want** to take the snapshot.
field.exclude.list	*empty string*	An optional comma-separated list of the fully-qualified names of fields that should be excluded from change event message values. Fully-qualified names for fields are of the form *databaseName.collectionName.fieldName.nestedFieldName*, where *databaseName* and *collectionName* may contain the wildcard (*) which matches any characters.

Property	Default	Description
field.renames	*empty string*	An optional comma-separated list of the fully-qualified replacements of fields that should be used to rename fields in change event message values. Fully-qualified replacements for fields are of the form *databaseName.collectionName.fieldName.nestedFieldName:newNestedFieldName*, where *databaseName* and *collectionName* may contain the wildcard (*) which matches any characters, the colon character (:) is used to determine rename mapping of field. The next field replacement is applied to the result of the previous field replacement in the list, so keep this in mind when renaming multiple fields that are in the same path.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The MongoDB connector will attempt to use a separate task for each replica set, so the default is acceptable when using the connector with a single MongoDB replica set. When using the connector with a MongoDB sharded cluster, we recommend specifying a value that is equal to or more than the number of shards in the cluster, so that the work for each replica set can be distributed by Kafka Connect.
snapshot.max.threads	**1**	Positive integer value that specifies the maximum number of threads used to perform an intial sync of the collections in a replica set. Defaults to 1.
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. **true** - a delete operation is represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is emitted. After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case log compaction is enabled for the topic.

Property	Default	Description
snapshot.delay.ms		An interval in milliseconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size	**0**	Specifies the maximum number of documents that should be read in one go from each collection while taking a snapshot. The connector will read the collection contents in multiple batches of this size. Defaults to 0, which indicates that the server chooses an appropriate fetch size.
schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.

The following *advanced* configuration properties have good defaults that will work in most situations and therefore rarely need to be specified in the connector's configuration.

Table 4.13. Debezium MongoDB connector advanced configuration properties

Property	Default	Description
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.

Property	Default	Description
max.queue.size	8192	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.
max.queue.size.in.bytes	0	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.
poll.interval.ms	1000	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.
connect.backoff.initial.delay.ms	1000	Positive integer value that specifies the initial delay when trying to reconnect to a primary after the first failed connection attempt or when no primary is available. Defaults to 1 second (1000 ms).
connect.backoff.max.delay.ms	1000	Positive integer value that specifies the maximum delay when trying to reconnect to a primary after repeated failed connection attempts or when no primary is available. Defaults to 120 seconds (120,000 ms).

Property	Default	Description
connect.max.attempts	**16**	Positive integer value that specifies the maximum number of failed connection attempts to a replica set primary before an exception occurs and task is aborted. Defaults to 16, which with the defaults for **connect.backoff.initial.delay.ms** and **connect.backoff.max.delay.ms** results in just over 20 minutes of attempts before failing.
mongodb.members.auto.discover	**true**	Boolean value that specifies whether the addresses in 'mongodb.hosts' are seeds that should be used to discover all members of the cluster or replica set (**true**), or whether the address(es) in **mongodb.hosts** should be used as is (**false**). The default is **true** and should be used in all cases except where MongoDB is fronted by a proxy.
heartbeat.interval.ms	**0**	Controls how frequently heartbeat messages are sent. This property contains an interval in milliseconds that defines how frequently the connector sends messages into a heartbeat topic. This can be used to monitor whether the connector is still receiving change events from the database. You also should leverage heartbeat messages in cases where only records in non-captured collections are changed for a longer period of time. In such situation the connector would proceed to read the oplog/change stream from the database but never emit any change messages into Kafka, which in turn means that no offset updates are committed to Kafka. This will cause the oplog files to be rotated out but connector will not notice it so on restart some events are no longer available which leads to the need of re-execution of the initial snapshot. Set this parameter to **0** to not send heartbeat messages at all. Disabled by default.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.

Property	Default	Description
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names are sanitized to adhere to Avro naming requirements.
skipped.operations		comma-separated list of operation types that will be skipped during streaming. The operations include: **c** for inserts/create, **u** for updates, and **d** for deletes. By default, no operations are skipped.
snapshot.collection.filter.overrides		Controls which collection items are included in snapshot. This property affects snapshots only. Specify a comma-separated list of collection names in the form *databaseName.collectionName*. For each collection that you specify, also specify another configuration property: **snapshot.collection.filter.overrides.*databaseName.collectionName***. For example, the name of the other configuration property might be: **snapshot.collection.filter.overrides.customers.orders**. Set this property to a valid filter expression that retrieves only the items that you want in the snapshot. When the connector performs a snapshot, it retrieves only the items that matches the filter expression.
provide.transaction.metadata	**false**	When set to **true** Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.

Property	Default	Description
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name (see Section 4.2.2, "How Debezium MongoDB connectors use logical names for replica sets and sharded clusters"); defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**.
retriable.restart.connector.wait.ms	10000 (10 seconds)	The number of milliseconds to wait before restarting a connector after a retriable error occurs.
mongodb.poll.interval.ms	**30000**	The interval in which the connector polls for new, removed, or changed replica sets.
mongodb.connect.timeout.ms	10000 (10 seconds)	The number of milliseconds the driver will wait before a new connection attempt is aborted.
mongodb.socket.timeout.ms	0	The number of milliseconds before a send/receive on the socket can take before a timeout occurs. A value of **0** disables this behavior.
mongodb.server.selection.timeout.ms	30000 (30 seconds)	The number of milliseconds the driver will wait to select a server before it times out and throws an error.
cursor.max.await.time.ms	**0**	Specifies the maximum number of milliseconds the oplog/change stream cursor will wait for the server to produce a result before causing an execution timeout exception. A value of **0** indicates using the server/driver default wait timeout.
signal.data.collection	No default	Fully-qualified name of the data collection that is used to send signals to the connector. Use the following format to specify the collection name: ***<databaseName>.<collectionName>*** Signaling is a Technology Preview feature for the Debezium MongoDB connector.

Property	Default	Description
incremental.snapshot.chunk.size	1024	The maximum number of documents that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment. Incremental snapshots is a Technology Preview feature for the Debezium MongoDB connector.

4.6. MONITORING DEBEZIUM MONGODB CONNECTOR PERFORMANCE

The Debezium MongoDB connector has two metric types in addition to the built-in support for JMX metrics that Zookeeper, Kafka, and Kafka Connect have.

- Snapshot metrics provide information about connector operation while performing a snapshot.
- Streaming metrics provide information about connector operation when the connector is capturing changes and streaming change event records.

The Debezium monitoring documentation provides details about how to expose these metrics by using JMX.

4.6.1. Monitoring Debezium during MongoDB snapshots

The **MBean** is **debezium.mongodb:type=connector-metrics,context=snapshot,server=*<mongodb.server.name>*,task=*<task.id>***. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	string	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	long	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	long	The total number of events that this connector has seen since last started or reset.

Attributes	Type	Description
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.

Attributes	Type	Description
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

The Debezium MongoDB connector also provides the following custom snapshot metrics:

Attribute	Type	Description
NumberOfDisconnects	**long**	Number of database disconnects.

4.6.2. Monitoring Debezium MongoDB connector record streaming

The **MBean** is **debezium.mongodb:type=connector-metrics,context=streaming,server=*<mongodb.server.name>*,task=*<task.id>***. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since the last start or metrics reset.

Attributes	Type	Description
TotalNumberOfCreateEventsSeen	**long**	The total number of create events that this connector has seen since the last start or metrics reset.
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.

Attributes	Type	Description
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

The Debezium MongoDB connector also provides the following custom streaming metrics:

Attribute	Type	Description
NumberOfDisconnects	**long**	Number of database disconnects.
NumberOfPrimaryElections	**long**	Number of primary node elections.

4.7. HOW DEBEZIUM MONGODB CONNECTORS HANDLE FAULTS AND PROBLEMS

Debezium is a distributed system that captures all changes in multiple upstream databases, and will never miss or lose an event. When the system is operating normally and is managed carefully, then Debezium provides *exactly once* delivery of every change event.

If a fault occurs, the system does not lose any events. However, while it is recovering from the fault, it might repeat some change events. In such situations, Debezium, like Kafka, provides *at least once* delivery of change events.

The following topics provide details about how the Debezium MongoDB connector handles various kinds of faults and problems.

- Configuration and startup errors
- MongoDB becomes unavailable
- Kafka Connect process stops gracefully
- Kafka Connect process crashes
- Kafka becomes unavailable
- Connector is stopped for a long interval
- MongoDB loses writes

Configuration and startup errors

In the following situations, the connector fails when trying to start, reports an error or exception in the log, and stops running:

- The connector's configuration is invalid.
- The connector cannot successfully connect to MongoDB by using the specified connection parameters.

After a failure, the connector attempts to reconnect by using exponential backoff. You can configure the maximum number of reconnection attempts.

In these cases, the error will have more details about the problem and possibly a suggested work around. The connector can be restarted when the configuration has been corrected or the MongoDB problem has been addressed.

MongoDB becomes unavailable

Once the connector is running, if the primary node of any of the MongoDB replica sets become unavailable or unreachable, the connector will repeatedly attempt to reconnect to the primary node, using exponential backoff to prevent saturating the network or servers. If the primary remains unavailable after the configurable number of connection attempts, the connector will fail.

The attempts to reconnect are controlled by three properties:

- **connect.backoff.initial.delay.ms** - The delay before attempting to reconnect for the first time, with a default of 1 second (1000 milliseconds).
- **connect.backoff.max.delay.ms** - The maximum delay before attempting to reconnect, with a default of 120 seconds (120,000 milliseconds).
- **connect.max.attempts** - The maximum number of attempts before an error is produced, with a default of 16.

Each delay is double that of the prior delay, up to the maximum delay. Given the default values, the following table shows the delay for each failed connection attempt and the total accumulated time before failure.

Reconnection attempt number	Delay before attempt, in seconds	Total delay before attempt, in minutes and seconds
1	1	00:01
2	2	00:03
3	4	00:07
4	8	00:15
5	16	00:31
6	32	01:03
7	64	02:07
8	120	04:07
9	120	06:07
10	120	08:07
11	120	10:07
12	120	12:07
13	120	14:07
14	120	16:07
15	120	18:07
16	120	20:07

Kafka Connect process stops gracefully

If Kafka Connect is being run in distributed mode, and a Kafka Connect process is stopped gracefully, then prior to shutdown of that processes Kafka Connect will migrate all of the process' connector tasks to another Kafka Connect process in that group, and the new connector tasks will pick up exactly where the prior tasks left off. There is a short delay in processing while the connector tasks are stopped gracefully and restarted on the new processes.

If the group contains only one process and that process is stopped gracefully, then Kafka Connect will stop the connector and record the last offset for each replica set. Upon restart, the replica set tasks will continue exactly where they left off.

Kafka Connect process crashes

If the Kafka Connector process stops unexpectedly, then any connector tasks it was running will terminate without recording their most recently-processed offsets. When Kafka Connect is being run in distributed mode, it will restart those connector tasks on other processes. However, the MongoDB connectors will resume from the last offset *recorded* by the earlier processes, which means that the new replacement tasks may generate some of the same change events that were processed just prior to the crash. The number of duplicate events depends on the offset flush period and the volume of data changes just before the crash.

NOTE

Because there is a chance that some events may be duplicated during a recovery from failure, consumers should always anticipate some events may be duplicated. Debezium changes are idempotent, so a sequence of events always results in the same state.

Debezium also includes with each change event message the source-specific information about the origin of the event, including the MongoDB event's unique transaction identifier (**h**) and timestamp (**sec** and **ord**). Consumers can keep track of other of these values to know whether it has already seen a particular event.

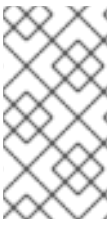
Kafka becomes unavailable

As the connector generates change events, the Kafka Connect framework records those events in Kafka using the Kafka producer API. Kafka Connect will also periodically record the latest offset that appears in those change events, at a frequency that you have specified in the Kafka Connect worker configuration. If the Kafka brokers become unavailable, the Kafka Connect worker process running the connectors will simply repeatedly attempt to reconnect to the Kafka brokers. In other words, the connector tasks will simply pause until a connection can be reestablished, at which point the connectors will resume exactly where they left off.

Connector is stopped for a long interval

If the connector is gracefully stopped, the replica sets can continue to be used and any new changes are recorded in MongoDB's oplog. When the connector is restarted, it will resume streaming changes for each replica set where it last left off, recording change events for all of the changes that were made while the connector was stopped. If the connector is stopped long enough such that MongoDB purges from its oplog some operations that the connector has not read, then upon startup the connector will perform a snapshot.

A properly configured Kafka cluster is capable of massive throughput. Kafka Connect is written with Kafka best practices, and given enough resources will also be able to handle very large numbers of database change events. Because of this, when a connector has been restarted after a while, it is very likely to catch up with the database, though how quickly will depend upon the capabilities and performance of Kafka and the volume of changes being made to the data in MongoDB.



NOTE

If the connector remains stopped for long enough, MongoDB might purge older oplog files and the connector's last position may be lost. In this case, when the connector configured with *initial* snapshot mode (the default) is finally restarted, the MongoDB server will no longer have the starting point and the connector will fail with an error.

MongoDB loses writes

In certain failure situations, MongoDB can lose commits, which results in the MongoDB connector being unable to capture the lost changes. For example, if the primary crashes suddenly after it applies a change and records the change to its oplog, the oplog might become unavailable before secondary nodes can read its contents. As a result, the secondary node that is elected as the new primary node might be missing the most recent changes from its oplog.

At this time, there is no way to prevent this side effect in MongoDB.

CHAPTER 5. DEBEZIUM CONNECTOR FOR MYSQL

MySQL has a binary log (binlog) that records all operations in the order in which they are committed to the database. This includes changes to table schemas as well as changes to the data in tables. MySQL uses the binlog for replication and recovery.

The Debezium MySQL connector reads the binlog, produces change events for row-level **INSERT**, **UPDATE**, and **DELETE** operations, and emits the change events to Kafka topics. Client applications read those Kafka topics.

As MySQL is typically set up to purge binlogs after a specified period of time, the MySQL connector performs an initial *consistent snapshot* of each of your databases. The MySQL connector reads the binlog from the point at which the snapshot was made.

For information about the MySQL Database versions that are compatible with this connector, see the Debezium Supported Configurations page.

Information and procedures for using a Debezium MySQL connector are organized as follows:

- Section 5.1, "How Debezium MySQL connectors work"
- Section 5.2, "Descriptions of Debezium MySQL connector data change events"
- Section 5.3, "How Debezium MySQL connectors map data types"
- Section 5.4, "Setting up MySQL to run a Debezium connector"
- Section 5.5, "Deployment of Debezium MySQL connectors"
- Section 5.6, "Monitoring Debezium MySQL connector performance"
- Section 5.7, "How Debezium MySQL connectors handle faults and problems"

5.1. HOW DEBEZIUM MYSQL CONNECTORS WORK

An overview of the MySQL topologies that the connector supports is useful for planning your application. To optimally configure and run a Debezium MySQL connector, it is helpful to understand how the connector tracks the structure of tables, exposes schema changes, performs snapshots, and determines Kafka topic names.

Details are in the following topics:

- Section 5.1.1, "MySQL topologies supported by Debezium connectors"
- Section 5.1.2, "How Debezium MySQL connectors handle database schema changes"
- Section 5.1.3, "How Debezium MySQL connectors expose database schema changes"
- Section 5.1.4, "How Debezium MySQL connectors perform database snapshots"
- Section 5.1.5, "Default names of Kafka topics that receive Debezium MySQL change event records"

5.1.1. MySQL topologies supported by Debezium connectors

The Debezium MySQL connector supports the following MySQL topologies:

Standalone

When a single MySQL server is used, the server must have the binlog enabled (*and optionally GTIDs enabled*) so the Debezium MySQL connector can monitor the server. This is often acceptable, since the binary log can also be used as an incremental backup. In this case, the MySQL connector always connects to and follows this standalone MySQL server instance.

Primary and replica

The Debezium MySQL connector can follow one of the primary servers or one of the replicas (*if that replica has its binlog enabled*), but the connector sees changes in only the cluster that is visible to that server. Generally, this is not a problem except for the multi-primary topologies.
The connector records its position in the server's binlog, which is different on each server in the cluster. Therefore, the connector must follow just one MySQL server instance. If that server fails, that server must be restarted or recovered before the connector can continue.

High available clusters

A variety of high availability solutions exist for MySQL, and they make it significantly easier to tolerate and almost immediately recover from problems and failures. Most HA MySQL clusters use GTIDs so that replicas are able to keep track of all changes on any of the primary servers.

Multi-primary

Network Database (NDB) cluster replication uses one or more MySQL replica nodes that each replicate from multiple primary servers. This is a powerful way to aggregate the replication of multiple MySQL clusters. This topology requires the use of GTIDs.
A Debezium MySQL connector can use these multi-primary MySQL replicas as sources, and can fail over to different multi-primary MySQL replicas as long as the new replica is caught up to the old replica. That is, the new replica has all transactions that were seen on the first replica. This works even if the connector is using only a subset of databases and/or tables, as the connector can be configured to include or exclude specific GTID sources when attempting to reconnect to a new multi-primary MySQL replica and find the correct position in the binlog.

Hosted

There is support for the Debezium MySQL connector to use hosted options such as Amazon RDS and Amazon Aurora.
Because these hosted options do not allow a global read lock, table-level locks are used to create the *consistent snapshot*.

5.1.2. How Debezium MySQL connectors handle database schema changes

When a database client queries a database, the client uses the database's current schema. However, the database schema can be changed at any time, which means that the connector must be able to identify what the schema was at the time each insert, update, or delete operation was recorded. Also, a connector cannot just use the current schema because the connector might be processing events that are relatively old that were recorded before the tables' schemas were changed.

To ensure correct processing of changes that occur after a schema change, MySQL includes in the binlog not only the row-level changes to the data, but also the DDL statements that are applied to the database. As the connector reads the binlog and comes across these DDL statements, it parses them and updates an in-memory representation of each table's schema. The connector uses this schema representation to identify the structure of the tables at the time of each insert, update, or delete operation and to produce the appropriate change event. In a separate database history Kafka topic, the connector records all DDL statements along with the position in the binlog where each DDL statement appeared.

When the connector restarts after having crashed or been stopped gracefully, the connector starts

reading the binlog from a specific position, that is, from a specific point in time. The connector rebuilds the table structures that existed at this point in time by reading the database history Kafka topic and parsing all DDL statements up to the point in the binlog where the connector is starting.

This database history topic is for connector use only. The connector can optionally emit schema change events to a different topic that is intended for consumer applications.

When the MySQL connector captures changes in a table to which a schema change tool such as **gh-ost** or **pt-online-schema-change** is applied, there are helper tables created during the migration process. The connector needs to be configured to capture change to these helper tables. If consumers do not need the records generated for helper tables, then a single message transform can be applied to filter them out.

See default names for topics that receive Debezium event records.

5.1.3. How Debezium MySQL connectors expose database schema changes

You can configure a Debezium MySQL connector to produce schema change events that describe schema changes that are applied to captured tables in the database. The connector writes schema change events to a Kafka topic named ***<serverName>***, where ***serverName*** is the logical server name that is specified in the **database.server.name** connector configuration property. Messages that the connector sends to the schema change topic contain a payload, and, optionally, also contain the schema of the change event message.

The payload of a schema change event message includes the following elements:

ddl

Provides the SQL **CREATE**, **ALTER**, or **DROP** statement that results in the schema change.

databaseName

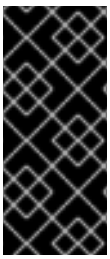
The name of the database to which the DDL statements are applied. The value of **databaseName** serves as the message key.

pos

The position in the binlog where the statements appear.

tableChanges

A structured representation of the entire table schema after the schema change. The **tableChanges** field contains an array that includes entries for each column of the table. Because the structured representation presents data in JSON or Avro format, consumers can easily read messages without first processing them through a DDL parser.



IMPORTANT

For a table that is in capture mode, the connector not only stores the history of schema changes in the schema change topic, but also in an internal database history topic. The internal database history topic is for connector use only and it is not intended for direct use by consuming applications. Ensure that applications that require notifications about schema changes consume that information only from the schema change topic.

IMPORTANT

Never partition the database history topic. For the database history topic to function correctly, it must maintain a consistent, global order of the event records that the connector emits to it.

To ensure that the topic is not split among partitions, set the partition count for the topic by using one of the following methods:

- If you create the database history topic manually, specify a partition count of **1**.
- If you use the Apache Kafka broker to create the database history topic automatically, the topic is created, set the value of the Kafka **num.partitions** configuration option to **1**.

WARNING

The format of the messages that a connector emits to its schema change topic is in an incubating state and is subject to change without notice.

Example: Message emitted to the MySQL connector schema change topic

The following example shows a typical schema change message in JSON format. The message contains a logical representation of the table schema.

```
{
  "schema": {
  ...
  },
  "payload": {
        "source": {  // (1)
        "version": "1.9.7.Final",
        "connector": "mysql",
        "name": "dbserver1",
        "ts_ms": 0,
        "snapshot": "false",
        "db": "inventory",
        "sequence": null,
        "table": "customers",
        "server_id": 0,
        "gtid": null,
        "file": "mysql-bin.000003",
        "pos": 219,
        "row": 0,
        "thread": null,
        "query": null
    },
    "databaseName": "inventory", // (2)
    "schemaName": null,
    "ddl": "ALTER TABLE customers ADD COLUMN middle_name VARCHAR(2000)", // (3)
    "tableChanges": [ // (4)
```

```
{
"type": "ALTER", // (5)
"id": "\"inventory\".\"customers\"",  // (6)
"table": { // (7)
    "defaultCharsetName": "latin1",
    "primaryKeyColumnNames": [  // (8)
        "id"
    ],
    "columns": [ // (9)
        {
        "name": "id",
        "jdbcType": 4,
        "nativeType": null,
        "typeName": "INT",
        "typeExpression": "INT",
        "charsetName": null,
        "length": 11,
        "scale": null,
        "position": 1,
        "optional": false,
        "autoIncremented": true,
        "generated": true
    },
    {
        "name": "first_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "VARCHAR",
        "typeExpression": "VARCHAR",
        "charsetName": "latin1",
        "length": 255,
        "scale": null,
        "position": 2,
        "optional": false,
        "autoIncremented": false,
        "generated": false
    },                      {
        "name": "last_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "VARCHAR",
        "typeExpression": "VARCHAR",
        "charsetName": "latin1",
        "length": 255,
        "scale": null,
        "position": 3,
        "optional": false,
        "autoIncremented": false,
        "generated": false
    },
    {
        "name": "email",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "VARCHAR",
        "typeExpression": "VARCHAR",
```

```
            "charsetName": "latin1",
            "length": 255,
            "scale": null,
            "position": 4,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "middle_name",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "VARCHAR",
            "typeExpression": "VARCHAR",
            "charsetName": "latin1",
            "length": 2000,
            "scale": null,
            "position": 5,
            "optional": true,
            "autoIncremented": false,
            "generated": false
          }
        ]
      }
    }
  ]
 },
 "payload": {
  "databaseName": "inventory",
  "ddl": "CREATE TABLE products ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255) NOT NULL, description VARCHAR(512), weight FLOAT ); ALTER TABLE
products AUTO_INCREMENT = 101;",
  "source" : {
   "version": "1.9.7.Final",
   "name": "mysql-server-1",
   "server_id": 0,
   "ts_ms": 0,
   "gtid": null,
   "file": "mysql-bin.000003",
   "pos": 154,
   "row": 0,
   "snapshot": true,
   "thread": null,
   "db": null,
   "table": null,
   "query": null
  }
 }
}
```


Table 5.1. Descriptions of fields in messages emitted to the schema change topic

Item	Field name	Description

Item	Field name	Description
1	**source**	The **source** field is structured exactly as standard data change events that the connector writes to table-specific topics. This field is useful to correlate events on different topics.
2	**databaseName** **schemaName**	Identifies the database and the schema that contains the change. The value of the **databaseName** field is used as the message key for the record.
3	**ddl**	This field contains the DDL that is responsible for the schema change. The **ddl** field can contain multiple DDL statements. Each statement applies to the database in the **databaseName** field. Multiple DDL statements appear in the order in which they were applied to the database. Clients can submit multiple DDL statements that apply to multiple databases. If MySQL applies them atomically, the connector takes the DDL statements in order, groups them by database, and creates a schema change event for each group. If MySQL applies them individually, the connector creates a separate schema change event for each statement.
4	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.
5	**type**	Describes the kind of change. The value is one of the following: **CREATE** Table created. **ALTER** Table modified. **DROP** Table deleted.
6	**id**	Full identifier of the table that was created, altered, or dropped. In the case of a table rename, this identifier is a concatenation of ***<old>,<new>*** table names.
7	**table**	Represents table metadata after the applied change.
8	**primaryKeyColumnNames**	List of columns that compose the table's primary key.
9	**columns**	Metadata for each column in the changed table.

See also: schema history topic.

5.1.4. How Debezium MySQL connectors perform database snapshots

When a Debezium MySQL connector is first started, it performs an initial *consistent snapshot* of your database. The following flow describes how the connector creates this snapshot. This flow is for the default snapshot mode, which is **initial**. For information about other snapshot modes, see the MySQL connector **snapshot.mode** configuration property.

Table 5.2. Workflow for performing an initial snapshot with a global read lock

Step	Action
1	Grabs a global read lock that blocks *writes* by other database clients. The snapshot itself does not prevent other clients from applying DDL that might interfere with the connector's attempt to read the binlog position and table schemas. The connector keeps the global read lock while it reads the binlog position, and releases the lock as described in a later step.
2	Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the *consistent snapshot*.
3	Reads the current binlog position.
4	Reads the schema of the databases and tables for which the connector is configured to capture changes.
5	Releases the global read lock. Other database clients can now write to the database.
6	If applicable, writes the DDL changes to the schema change topic, including all necessary **DROP...** and **CREATE...** DDL statements.
7	Scans the database tables. For each row, the connector emits **CREATE** events to the relevant table-specific Kafka topics.
8	Commits the transaction.
9	Records the completed snapshot in the connector offsets.

Connector restarts

If the connector fails, stops, or is rebalanced while performing the *initial snapshot*, then after the connector restarts, it performs a new snapshot. After that *intial snapshot* is completed, the Debezium MySQL connector restarts from the same position in the binlog so it does not miss any updates.
If the connector stops for long enough, MySQL could purge old binlog files and the connector's position would be lost. If the position is lost, the connector reverts to the *initial snapshot* for its starting position. For more tips on troubleshooting the Debezium MySQL connector, see behavior when things go wrong.

Global read locks not allowed

Some environments do not allow global read locks. If the Debezium MySQL connector detects that

global read locks are not permitted, the connector uses table-level locks instead and performs a snapshot with this method. This requires the database user for the Debezium connector to have **LOCK TABLES** privileges.

Table 5.3. Workflow for performing an initial snapshot with table-level locks

Step	Action
1	Obtains table-level locks.
2	Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the *consistent snapshot*.
3	Reads and filters the names of the databases and tables.
4	Reads the current binlog position.
5	Reads the schema of the databases and tables for which the connector is configured to capture changes.
6	If applicable, writes the DDL changes to the schema change topic, including all necessary **DROP...** and **CREATE...** DDL statements.
7	Scans the database tables. For each row, the connector emits **CREATE** events to the relevant table-specific Kafka topics.
8	Commits the transaction.
9	Releases the table-level locks.
10	Records the completed snapshot in the connector offsets.

5.1.4.1. Ad hoc snapshots

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing table data, Debezium includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of tables.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling tables. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the tables to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the tables in the database.

You specify the tables to capture by sending an **execute-snapshot** message to the signaling table. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the tables to include in the snapshot, as described in the following table:

Table 5.4. Example of an ad hoc`execute-snapshot` signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.
data-collections	*N/A*	An array that contains the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling table. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each table. Based on the number of entries in the table, and the configured chunk size, Debezium divides the table into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

5.1.4.2. Incremental snapshots

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each table in phases, in a series of configurable chunks. You can specify the tables that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each table row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the other.

- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the table from the beginning.

- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a table to its **table.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each table by primary key and then splits the table into chunks based on the configured chunk size. Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying table records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a table row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same table row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified table chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event

logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the table's Kafka topic.

The connector repeats the process for each snapshot chunk.

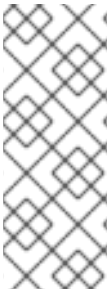
Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling table on the source database. You submit signals to the table as SQL **INSERT** queries. After Debezium detects the change in the signaling table, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the tables to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.



NOTE

If the name of a table that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the table to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling table.

The following table describes the these parameters:

Table 5.5. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling table on the source database
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of table names to include in the snapshot. The array lists tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
```

```
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

5.1.5. Default names of Kafka topics that receive Debezium MySQL change event records

By default, the MySQL connector writes change events for all of the **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table.

The connector uses the following convention to name change event topics:

serverName.databaseName.tableName

Suppose that **fulfillment** is the server name, **inventory** is the database name, and the database contains tables named **orders**, **customers**, and **products**. The Debezium MySQL connector emits events to three Kafka topics, one for each table in the database:

```
fulfillment.inventory.orders
fulfillment.inventory.customers
fulfillment.inventory.products
```

The following list provides definitions for the components of the default name:

serverName

The logical name of the server as specified by the **database.server.name** connector configuration property.

schemaName

The name of the schema in which the operation occurred.

tableName

The name of the table in which the operation occurred.

The connector applies similar naming conventions to label its internal database history topics, schema change topics, and transaction metadata topics.

If the default topic name do not meet your requirements, you can configure custom topic names. To configure custom topic names, you specify regular expressions in the logical topic routing SMT. For more information about using the logical topic routing SMT to customize topic naming, see Topic routing.

Transaction metadata

Debezium can generate events that represent transaction boundaries and that enrich data change event messages.

LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

Debezium generates transaction boundary events for the **BEGIN** and **END** delimiters in every transaction. Transaction boundary events contain the following fields:

status

BEGIN or **END**.

id

String representation of the unique transaction identifier.

event_count (for **END** events)

Total number of events emitted by the transaction.

data_collections (for **END** events)

An array of pairs of **data_collection** and **event_count** elements. that indicates the number of events that the connector emits for changes that originate from a data collection.

Example

```
{
  "status": "BEGIN",
  "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "s1.a",
      "event_count": 1
    },
    {
      "data_collection": "s2.a",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, the connector emits transaction events to the ***<database.server.name>*.transaction** topic.

Change data event enrichment

When transaction metadata is enabled the data message **Envelope** is enriched with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

- **id** - string representation of unique transaction identifier
- **total_order** - absolute position of the event among all events generated by the transaction
- **data_collection_order** - the per-data collection position of the event among all events that were emitted by the transaction

Following is an example of a message:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

For systems which don't have GTID enabled, the transaction identifier is constructed using the combination of binlog filename and binlog position. For example, if the binlog filename and position corresponding to the transaction BEGIN event are mysql-bin.000002 and 1913 respectively then the Debezium constructed transaction identifier would be **file=mysql-bin.000002,pos=1913**.

5.2. DESCRIPTIONS OF DEBEZIUM MYSQL CONNECTOR DATA CHANGE EVENTS

The Debezium MySQL connector generates a data change event for each row-level **INSERT**, **UPDATE**, and **DELETE** operation. Each event contains a key and a value. The structure of the key and the value depends on the table that was changed.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events may change over time, which can be difficult for consumers to handle. To address this, each event contains the schema for its content or, if you are using a schema registry, a schema ID that a consumer can use to obtain the schema from the registry. This makes each event self-contained.

The following skeleton JSON shows the basic four parts of a change event. However, how you configure the Kafka Connect converter that you choose to use in your application determines the representation of these four parts in change events. A **schema** field is in a change event only when you configure the converter to produce it. Likewise, the event key and event payload are in a change event only if you configure a converter to produce it. If you use the JSON converter and you configure it to produce all four basic change event parts, change events have this structure:

```
{
 "schema": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

Table 5.6. Overview of change event basic content

Item	Field name	Description
1	**schema**	The first **schema** field is part of the event key. It specifies a Kafka Connect schema that describes what is in the event key's **payload** portion. In other words, the first **schema** field describes the structure of the primary key, or the unique key if the table does not have a primary key, for the table that was changed. It is possible to override the table's primary key by setting the **message.key.columns** connector configuration property. In this case, the first schema field describes the structure of the key identified by that property.
2	**payload**	The first **payload** field is part of the event key. It has the structure described by the previous **schema** field and it contains the key for the row that was changed.
3	**schema**	The second **schema** field is part of the event value. It specifies the Kafka Connect schema that describes what is in the event value's **payload** portion. In other words, the second **schema** describes the structure of the row that was changed. Typically, this schema contains nested schemas.
4	**payload**	The second **payload** field is part of the event value. It has the structure described by the previous **schema** field and it contains the actual data for the row that was changed.

By default, the connector streams change event records to topics with names that are the same as the event's originating table. See topic names.

WARNING

The MySQL connector ensures that all Kafka Connect schema names adhere to the Avro schema name format. This means that the logical server name must start with a Latin letter or an underscore, that is, a-z, A-Z, or _. Each remaining character in the logical server name and each character in the database and table names must be a Latin letter, a digit, or an underscore, that is, a-z, A-Z, 0-9, or _. If there is an invalid character it is replaced with an underscore character.

This can lead to unexpected conflicts if the logical server name, a database name, or a table name contains invalid characters, and the only characters that distinguish names from one another are invalid and thus replaced with underscores.

More details are in the following topics:

- Section 5.2.1, "About keys in Debezium MySQL change events"
- Section 5.2.2, "About values in Debezium MySQL change events"

5.2.1. About keys in Debezium MySQL change events

A change event's key contains the schema for the changed table's key and the changed row's actual key. Both the schema and its corresponding payload contain a field for each column in the changed table's **PRIMARY KEY** (or unique constraint) at the time the connector created the event.

Consider the following **customers** table, which is followed by an example of a change event key for this table.

```
CREATE TABLE customers (
  id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE KEY
) AUTO_INCREMENT=1001;
```

Every change event that captures a change to the **customers** table has the same event key schema. For as long as the **customers** table has the previous definition, every change event that captures a change to the **customers** table has the following key structure. In JSON, it looks like this:

```
{
 "schema": { 1
   "type": "struct",
   "name": "mysql-server-1.inventory.customers.Key", 2
   "optional": false, 3
   "fields": [ 4
     {
       "field": "id",
       "type": "int32",
       "optional": false
     }
```

```
    ]
  },
  "payload": { 5
    "id": 1001
  }
}
```

Table 5.7. Description of change event key

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's **payload** portion.
2	**mysql-server-1.inventory.customers.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format *connector-name.database-name.table-name*.**Key**. In this example: • **mysql-server-1** is the name of the connector that generated this event. • **inventory** is the database that contains the table that was changed. • **customers** is the table that was updated.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a table does not have a primary key.
4	**fields**	Specifies each field that is expected in the **payload**, including each field's name, type, and whether it is required.
5	**payload**	Contains the key for the row for which this change event was generated. In this example, the key, contains a single **id** field whose value is **1001**.

5.2.2. About values in Debezium MySQL change events

The value in a change event is a bit more complicated than the key. Like the key, the value has a **schema** section and a **payload** section. The **schema** section contains the schema that describes the **Envelope** structure of the **payload** section, including its nested fields. Change events for operations that create, update or delete data all have a value payload with an envelope structure.

Consider the same sample table that was used to show an example of a change event key:

```
CREATE TABLE customers (
  id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE KEY
) AUTO_INCREMENT=1001;
```

The value portion of a change event for a change to this table is described for:

- *create* events
- *update* events
- Primary key updates
- *delete* events
- Tombstone events

create events

The following example shows the value portion of a change event that the connector generates for an operation that creates data in the **customers** table:

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
```

```
        },
        {
          "type": "string",
          "optional": false,
          "field": "first_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "mysql-server-1.inventory.customers.Value",
      "field": "after"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "string",
          "optional": false,
          "field": "version"
        },
        {
          "type": "string",
          "optional": false,
          "field": "connector"
        },
        {
          "type": "string",
          "optional": false,
          "field": "name"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "ts_ms"
        },
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "string",
          "optional": false,
          "field": "db"
        },
        {
```

```
            "type": "string",
            "optional": true,
            "field": "table"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          },
          {
            "type": "string",
            "optional": true,
            "field": "gtid"
          },
          {
            "type": "string",
            "optional": false,
            "field": "file"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "pos"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "row"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "query"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source", (3)
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
```

```
    "name": "mysql-server-1.inventory.customers.Envelope" 4
  },
  "payload": { 5
    "op": "c", 6
    "ts_ms": 1465491411815, 7
    "before": null, 8
    "after": { 9
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 10
      "version": "1.9.7.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
    }
  }
}
```

Table 5.8. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **mysql-server-1.inventory.customers.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.

Item	Field name	Description
3	**name**	**io.debezium.connector.mysql.Source** is the schema for the payload's **source** field. This schema is specific to the MySQL connector. The connector uses it for all events that it generates.
4	**name**	**mysql-server-1.inventory.customers.Envelope** is the schema for the overall structure of the payload, where **mysql-server-1** is the connector name, **inventory** is the database, and **customers** is the table.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots)
7	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
8	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content.
9	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **id**, **first_name**, **last_name**, and **email** columns.

Item	Field name	Description
10	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: • Debezium version • Connector name • binlog name where the event was recorded • binlog position • Row within the event • If the event was part of a snapshot • Name of the database and table that contain the new row • ID of the MySQL thread that created the event (non-snapshot only) • MySQL server ID (if available) • Timestamp for when the change was made in the database If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration **include.query** property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event.

update events

The value of a change event for an update in the sample **customers** table has the same schema as a *create* event for that table. Likewise, the event value's payload has the same structure. However, the event value payload contains different values in an *update* event. Here is an example of a change event value in an event that the connector generates for an update in the **customers** table:

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { 2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
```

```
    "source": { 3
      "version": "1.9.7.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581029100,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u", 4
    "ts_ms": 1465581029523 5
  }
}
```

Table 5.9. Descriptions of *update* event value fields

Item	Field name	Description
1	**before**	An optional field that specifies the state of the row before the event occurred. In an *update* event value, the **before** field contains a field for each table column and the value that was in that column before the database commit. In this example, the **first_name** value is **Anne.**
2	**after**	An optional field that specifies the state of the row after the event occurred. You can compare the **before** and **after** structures to determine what the update to this row was. In the example, the **first_name** value is now **Anne Marie**.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different, for example, the sample *update* event is from a different position in the binlog. The source metadata includes: • Debezium version • Connector name • binlog name where the event was recorded • binlog position • Row within the event • If the event was part of a snapshot • Name of the database and table that contain the updated row • ID of the MySQL thread that created the event (non-snapshot only) • MySQL server ID (if available) • Timestamp for when the change was made in the database If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration **include.query** property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event.
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

Updating the columns for a row's primary/unique key changes the value of the row's key. When a key changes, Debezium outputs *three* events: a **DELETE** event and a tombstone event with the old key for the row, followed by an event with the new key for the row. Details are in the next section.

Primary key updates

An **UPDATE** operation that changes a row's primary key field(s) is known as a primary key change. For a primary key change, in place of an **UPDATE** event record, the connector emits a **DELETE** event record

for the old key and a **CREATE** event record for the new (updated) key. These events have the usual structure and content, and in addition, each one has a message header related to the primary key change:

- The **DELETE** event record has **__debezium.newkey** as a message header. The value of this header is the new primary key for the updated row.
- The **CREATE** event record has **__debezium.oldkey** as a message header. The value of this header is the previous (old) primary key that the updated row had.

delete events

The value in a *delete* change event has the same **schema** portion as *create* and *update* events for the same table. The **payload** portion in a *delete* event for the sample **customers** table looks like this:

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, 2
    "source": { 3
      "version": "1.9.7.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581902300,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d", 4
    "ts_ms": 1465581902461 5
  }
}
```

Table 5.10. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit.

Item	Field name	Description
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and **pos** field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector name • binlog name where the event was recorded • binlog position • Row within the event • If the event was part of a snapshot • Name of the database and table that contain the updated row • ID of the MySQL thread that created the event (non-snapshot only) • MySQL server ID (if available) • Timestamp for when the change was made in the database If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration **include.query** property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event.
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

A *delete* change event record provides a consumer with the information it needs to process the removal of this row. The old values are included because some consumers might require them in order to properly handle the removal.

MySQL connector events are designed to work with Kafka log compaction. Log compaction enables

removal of some older messages as long as at least the most recent message for every key is kept. This lets Kafka reclaim storage space while ensuring that the topic contains a complete data set and can be used for reloading key-based state.

Tombstone events

When a row is deleted, the *delete* event value still works with log compaction, because Kafka can remove all earlier messages that have that same key. However, for Kafka to remove all messages that have that same key, the message value must be **null**. To make this possible, after Debezium's MySQL connector emits a *delete* event, the connector emits a special tombstone event that has the same key but a **null** value.

5.3. HOW DEBEZIUM MYSQL CONNECTORS MAP DATA TYPES

The Debezium MySQL connector represents changes to rows with events that are structured like the table in which the row exists. The event contains a field for each column value. The MySQL data type of that column dictates how Debezium represents the value in the event.

Columns that store strings are defined in MySQL with a character set and collation. The MySQL connector uses the column's character set when reading the binary representation of the column values in the binlog events.

The connector can map MySQL data types to both *literal* and *semantic* types.

- **Literal type**: how the value is represented using Kafka Connect schema types.
- **Semantic type**: how the Kafka Connect schema captures the meaning of the field (schema name).

If the default data type conversions do not meet your needs, you can create a custom converter for the connector.

Details are in the following sections:

- Basic types
- Temporal types
- Decimal types
- Boolean values
- Spatial types

Basic types

The following table shows how the connector maps basic MySQL data types.

Table 5.11. Descriptions of basic type mappings

MySQL type	Literal type	Semantic type
BOOLEAN, BOOL	**BOOLEAN**	*n/a*
BIT(1)	**BOOLEAN**	*n/a*

MySQL type	Literal type	Semantic type
BIT(>1)	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits. The **byte[]** contains the bits in *little-endian* form and is sized to contain the specified number of bits. For example, where **n** is bits: **numBytes = n/8 + (n%8== 0 ? 0 : 1)**
TINYINT	**INT16**	*n/a*
SMALLINT[(M)]	**INT16**	*n/a*
MEDIUMINT[(M)]	**INT32**	*n/a*
INT, INTEGER[(M)]	**INT32**	*n/a*
BIGINT[(M)]	**INT64**	*n/a*
REAL[(M,D)]	**FLOAT32**	*n/a*
FLOAT[(M,D)]	**FLOAT64**	*n/a*
DOUBLE[(M,D)]	**FLOAT64**	*n/a*
CHAR(M)]	**STRING**	*n/a*
VARCHAR(M)]	**STRING**	*n/a*
BINARY(M)]	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting.
VARBINARY(M)]	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting.
TINYBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting.
TINYTEXT	**STRING**	*n/a*

MySQL type	Literal type	Semantic type
BLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting. Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.
TEXT	**STRING**	*n/a* Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.
MEDIUMBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting.
MEDIUMTEXT	**STRING**	*n/a*
LONGBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting. Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.
LONGTEXT	**STRING**	*n/a* Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.
JSON	**STRING**	**io.debezium.data.Json** Contains the string representation of a **JSON** document, array, or scalar.
ENUM	**STRING**	**io.debezium.data.Enum** The **allowed** schema parameter contains the comma-separated list of allowed values.
SET	**STRING**	**io.debezium.data.EnumSet** The **allowed** schema parameter contains the comma-separated list of allowed values.
YEAR[(2|4)]	**INT32**	**io.debezium.time.Year**

MySQL type	Literal type	Semantic type
TIMESTAMP[(M)]	**STRING**	**io.debezium.time.ZonedTimestamp** In ISO 8601 format with microsecond precision. MySQL allows **M** to be in the range of**0-6**.

Temporal types

Excluding the **TIMESTAMP** data type, MySQL temporal types depend on the value of the **time.precision.mode** connector configuration property. For **TIMESTAMP** columns whose default value is specified as **CURRENT_TIMESTAMP** or **NOW**, the value **1970-01-01 00:00:00** is used as the default value in the Kafka Connect schema.

MySQL allows zero-values for **DATE**, **DATETIME**, and **TIMESTAMP** columns because zero-values are sometimes preferred over null values. The MySQL connector represents zero-values as null values when the column definition allows null values, or as the epoch day when the column does not allow null values.

Temporal values without time zones

The **DATETIME** type represents a local date and time such as "2018-01-13 09:48:27". As you can see, there is no time zone information. Such columns are converted into epoch milliseconds or microseconds based on the column's precision by using UTC. The **TIMESTAMP** type represents a timestamp without time zone information. It is converted by MySQL from the server (or session's) current time zone into UTC when writing and from UTC into the server (or session's) current time zone when reading back the value. For example:

- **DATETIME** with a value of **2018-06-20 06:37:03** becomes **1529476623000**.
- **TIMESTAMP** with a value of **2018-06-20 06:37:03** becomes **2018-06-20T13:37:03Z**.

Such columns are converted into an equivalent **io.debezium.time.ZonedTimestamp** in UTC based on the server (or session's) current time zone. The time zone will be queried from the server by default. If this fails, it must be specified explicitly by the database **connectionTimeZone** MySQL configuration option. For example, if the database's time zone (either globally or configured for the connector by means of the **connectionTimeZone** option) is "America/Los_Angeles", the TIMESTAMP value "2018-06-20 06:37:03" is represented by a **ZonedTimestamp** with the value "2018-06-20T13:37:03Z".

The time zone of the JVM running Kafka Connect and Debezium does not affect these conversions.

More details about properties related to temporal values are in the documentation for MySQL connector configuration properties.

time.precision.mode=adaptive_time_microseconds(default)

The MySQL connector determines the literal type and semantic type based on the column's data type definition so that events represent exactly the values in the database. All time fields are in microseconds. Only positive **TIME** field values in the range of **00:00:00.000000** to **23:59:59.999999** can be captured correctly.

Table 5.12. Mappings when **time.precision.mode=adaptive_time_microseconds**

MySQL type	Literal type	Semantic type
DATE	**INT32**	**io.debezium.time.Date** Represents the number of days since the epoch.

MySQL type	Literal type	Semantic type
TIME[(M)]	**INT64**	**io.debezium.time.MicroTime** Represents the time value in microseconds and does not include time zone information. MySQL allows **M** to be in the range of **0-6**.
DATETIME, **DATETIME(0),** **DATETIME(1),** **DATETIME(2),** **DATETIME(3)**	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds past the epoch and does not include time zone information.
DATETIME(4), **DATETIME(5),** **DATETIME(6)**	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds past the epoch and does not include time zone information.

time.precision.mode=connect

The MySQL connector uses defined Kafka Connect logical types. This approach is less precise than the default approach and the events could be less precise if the database column has a *fractional second precision* value of greater than **3**. Values in only the range of **00:00:00.000** to **23:59:59.999** can be handled. Set **time.precision.mode=connect** only if you can ensure that the **TIME** values in your tables never exceed the supported ranges. The **connect** setting is expected to be removed in a future version of Debezium.

Table 5.13. Mappings when **time.precision.mode=connect**

MySQL type	Literal type	Semantic type
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Time** Represents the time value in microseconds since midnight and does not include time zone information.
DATETIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include time zone information.

Decimal types

Debezium connectors handle decimals according to the setting of the **decimal.handling.mode** connector configuration property.

decimal.handling.mode=precise

Table 5.14. Mappings when **decimal.handling.mode=precise**

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
DECIMAL[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.

decimal.handling.mode=double

Table 5.15. Mappings when **decimal.handling.mode=double**

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	**FLOAT64**	*n/a*
DECIMAL[(M[,D])]	**FLOAT64**	*n/a*

decimal.handling.mode=string

Table 5.16. Mappings when **decimal.handling.mode=string**

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	**STRING**	*n/a*
DECIMAL[(M[,D])]	**STRING**	*n/a*

Boolean values

MySQL handles the **BOOLEAN** value internally in a specific way. The **BOOLEAN** column is internally mapped to the **TINYINT(1)** data type. When the table is created during streaming then it uses proper **BOOLEAN** mapping as Debezium receives the original DDL. During snapshots, Debezium executes **SHOW CREATE TABLE** to obtain table definitions that return **TINYINT(1)** for both **BOOLEAN** and **TINYINT(1)** columns. Debezium then has no way to obtain the original type mapping and so maps to **TINYINT(1)**.

To enable you to convert source columns to Boolean data types, Debezium provides a **TinyIntOneToBooleanConverter** custom converter that you can use in one of the following ways:

- Map all **TINYINT(1)** or **TINYINT(1) UNSIGNED** columns to **BOOLEAN** types.
- Enumerate a subset of columns by using a comma-separated list of regular expressions. To use this type of conversion, you must set the **converters** configuration property with the **selector** parameter, as shown in the following example:

```
converters=boolean
boolean.type=io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter
boolean.selector=db1.table1.*, db1.table2.column1
```

- NOTE: MySQL8 not showing the length of **tinyint unsigned** type when snapshot executes **SHOW CREATE TABLE**, which means this converter doesn't work. The new option **length.checker** can solve this issue, the default value is **true**. Disable the **length.checker** and specify the columns that need to be converted to **selector** property instead of converting all columns based on type, as shown in the following example:

```
converters=boolean
boolean.type=io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter
boolean.length.checker=false
boolean.selector=db1.table1.*, db1.table2.column1
```

Spatial types

Currently, the Debezium MySQL connector supports the following spatial data types.

Table 5.17. Description of spatial type mappings

MySQL type	Literal type	Semantic type
GEOMETRY, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	**STRUCT**	**io.debezium.data.geometry.Geometry** Contains a structure with two fields: • **srid (INT32**: spatial reference system ID that defines the type of geometry object stored in the structure • **wkb (BYTES)**: binary representation of the geometry object encoded in the Well-Known-Binary (wkb) format. See the Open Geospatial Consortium for more details.

5.4. SETTING UP MYSQL TO RUN A DEBEZIUM CONNECTOR

Some MySQL setup tasks are required before you can install and run a Debezium connector.

Details are in the following sections:

- Section 5.4.1, "Creating a MySQL user for a Debezium connector"
- Section 5.4.2, "Enabling the MySQL binlog for Debezium"
- Section 5.4.3, "Enabling MySQL Global Transaction Identifiers for Debezium"
- Section 5.4.4, "Configuring MySQL session timesouts for Debezium"
- Section 5.4.5, "Enabling query log events for Debezium MySQL connectors"

5.4.1. Creating a MySQL user for a Debezium connector

A Debezium MySQL connector requires a MySQL user account. This MySQL user must have appropriate permissions on all databases for which the Debezium MySQL connector captures changes.

Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.

Procedure

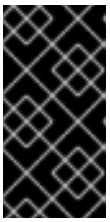
1. Create the MySQL user:

```
mysql> CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';
```

2. Grant the required permissions to the user:

```
mysql> GRANT SELECT, RELOAD, SHOW DATABASES, REPLICATION SLAVE,
REPLICATION CLIENT ON *.* TO 'user' IDENTIFIED BY 'password';
```

The table below describes the permissions.



IMPORTANT

If using a hosted option such as Amazon RDS or Amazon Aurora that does not allow a global read lock, table-level locks are used to create the *consistent snapshot*. In this case, you need to also grant **LOCK TABLES** permissions to the user that you create. See snapshots for more details.

3. Finalize the user's permissions:

```
mysql> FLUSH PRIVILEGES;
```

Table 5.18. Descriptions of user permissions

Keyword	Description
SELECT	Enables the connector to select rows from tables in databases. This is used only when performing a snapshot.
RELOAD	Enables the connector the use of the **FLUSH** statement to clear or reload internal caches, flush tables, or acquire locks. This is used only when performing a snapshot.
SHOW DATABASES	Enables the connector to see database names by issuing the **SHOW DATABASE** statement. This is used only when performing a snapshot.
REPLICATION SLAVE	Enables the connector to connect to and read the MySQL server binlog.

Keyword	Description
REPLICATION CLIENT	Enables the connector the use of the following statements: • **SHOW MASTER STATUS** • **SHOW SLAVE STATUS** • **SHOW BINARY LOGS** The connector always requires this.
ON	Identifies the database to which the permissions apply.
TO 'user'	Specifies the user to grant the permissions to.
IDENTIFIED BY 'password'	Specifies the user's MySQL password.

5.4.2. Enabling the MySQL binlog for Debezium

You must enable binary logging for MySQL replication. The binary logs record transaction updates for replication tools to propagate changes.

Prerequisites

- A MySQL server.
- Appropriate MySQL user privileges.

Procedure

1. Check whether the **log-bin** option is already on:

```
// for MySql 5.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
// for MySql 8.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM performance_schema.global_variables WHERE variable_name='log_bin';
```

2. If it is **OFF**, configure your MySQL server configuration file with the following properties, which are described in the table below:

```
server-id        = 223344
log_bin          = mysql-bin
binlog_format    = ROW
binlog_row_image  = FULL
expire_logs_days  = 10
```

3. Confirm your changes by checking the binlog status once more:

```
// for MySql 5.x
```

```
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
// for MySql 8.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM performance_schema.global_variables WHERE variable_name='log_bin';
```

Table 5.19. Descriptions of MySQL binlog configuration properties

Property	Description
server-id	The value for the **server-id** must be unique for each server and replication client in the MySQL cluster. During MySQL connector set up, Debezium assigns a unique server ID to the connector.
log_bin	The value of **log_bin** is the base name of the sequence of binlog files.
binlog_format	The **binlog-format** must be set to **ROW** or **row**.
binlog_row_image	The **binlog_row_image** must be set to **FULL** or **full**.
expire_logs_days	This is the number of days for automatic binlog file removal. The default is **0**, which means no automatic removal. Set the value to match the needs of your environment. See MySQL purges binlog files.

5.4.3. Enabling MySQL Global Transaction Identifiers for Debezium

Global transaction identifiers (GTIDs) uniquely identify transactions that occur on a server within a cluster. Though not required for a Debezium MySQL connector, using GTIDs simplifies replication and enables you to more easily confirm if primary and replica servers are consistent.

GTIDs are available in MySQL 5.6.5 and later. See the MySQL documentation for more details.

Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

Procedure

1. Enable **gtid_mode**:

```
mysql> gtid_mode=ON
```

2. Enable **enforce_gtid_consistency**:

```
mysql> enforce_gtid_consistency=ON
```

3. Confirm the changes:

```
mysql> show global variables like '%GTID%';
```

Result

```
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| enforce_gtid_consistency | ON    |
| gtid_mode                | ON    |
+--------------------------+-------+
```

Table 5.20. Descriptions of GTID options

Option	Description
gtid_mode	Boolean that specifies whether GTID mode of the MySQL server is enabled or not. • **ON** = enabled • **OFF** = disabled
enforce_gtid_consistency	Boolean that specifies whether the server enforces GTID consistency by allowing the execution of statements that can be logged in a transactionally safe manner. Required when using GTIDs. • **ON** = enabled • **OFF** = disabled

5.4.4. Configuring MySQL session timesouts for Debezium

When an initial consistent snapshot is made for large databases, your established connection could timeout while the tables are being read. You can prevent this behavior by configuring **interactive_timeout** and **wait_timeout** in your MySQL configuration file.

Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

Procedure

1. Configure **interactive_timeout**:

```
mysql> interactive_timeout=<duration-in-seconds>
```

2. Configure **wait_timeout**:

```
mysql> wait_timeout=<duration-in-seconds>
```

Table 5.21. Descriptions of MySQL session timeout options

Option	Description
interactive_timeout	The number of seconds the server waits for activity on an interactive connection before closing it. See MySQL's documentation for more details.
wait_timeout	The number of seconds the server waits for activity on a non-interactive connection before closing it. See MySQL's documentation for more details.

5.4.5. Enabling query log events for Debezium MySQL connectors

You might want to see the original **SQL** statement for each binlog event. Enabling the **binlog_rows_query_log_events** option in the MySQL configuration file allows you to do this.

This option is available in MySQL 5.6 and later.

Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

Procedure

- Enable **binlog_rows_query_log_events**:

  ```
  mysql> binlog_rows_query_log_events=ON
  ```

  binlog_rows_query_log_events is set to a value that enables/disables support for including the original **SQL** statement in the binlog entry.

  - **ON** = enabled
  - **OFF** = disabled

5.5. DEPLOYMENT OF DEBEZIUM MYSQL CONNECTORS

You can use either of the following methods to deploy a Debezium MySQL connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .

Additional resources

- Section 5.5.5, "Descriptions of Debezium MySQL connector configuration properties"

5.5.1. MySQL connector deployment using AMQ Streams

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.

NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

5.5.2. Using AMQ Streams to deploy a Debezium MySQL connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.
- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

   Example 5.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

   In the example that follows, the custom resource is configured to download the following artifacts:

   - The Debezium MySQL connector archive.
   - The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
   - The Debezium scripting SMT archive and the associated scripting engine that you want to use with the Debezium connector. The SMT archive and scripting language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: debezium-kafka-connect-cluster
     annotations:
   ```

```
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-mysql
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/1.9.7.Final-redhat-<build_number>/debezium-connector-mysql-1.9.7.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip (8)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip (9)
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar (10)
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json3.0.11/groovy-json-3.0.11.jar

    bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 5.22. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry such as Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation

Item	Description
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. The **type** value must match the type of the file that is referenced in the **url** field.
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) As an alternative, the Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **mysql-inventory-connector.yaml**

Example 5.2. mysql-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-mysql (1)
spec:
  class: io.debezium.connector.mysql.MySqlConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: mysql.debezium-mysql.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_mysql (10)
    database.include.list: public.inventory (11)
```

Table 5.23. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.

Item	Description
4	The connector's configuration.
5	The address of the host database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.
8	The password for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium MySQL deployment.

5.5.3. Deploying Debezium MySQL connectors by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium MySQL connector, you must build a custom Kafka Connect container image that contains the Debezium connector archive, and then push this container image to a container registry. You then need to create the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.

- A **KafkaConnector** CR that defines your Debezium MySQL connector. Apply this CR to the same OpenShift instance where you apply the **KafkaConnect** CR.

Prerequisites

- MySQL is running and you completed the steps to set up MySQL to work with a Debezium connector.
- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift .
- Podman or Docker is installed.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium MySQL container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-mysql.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/1.9.7.Final-redhat-<build_number>/debezium-connector-mysql-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-mysql-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-mysql-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

The command creates a Dockerfile with the name **debezium-container-for-mysql.yaml** in the current directory.

   b. Build the container image from the **debezium-container-for-mysql.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-mysql:latest .
```

```
docker build -t debezium-container-for-mysql:latest .
```

The preceding commands build a container image with the name **debezium-container-for-mysql**.

c. Push your custom image to a container registry, such as **quay.io** or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-mysql:latest
```

```
docker push <myregistry.io>/debezium-container-for-mysql:latest
```

d. Create a new Debezium MySQL **KafkaConnect** custom resource (CR). For example, create a **KafkaConnect** CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  #...
  image: debezium-container-for-mysql 2
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply the **KafkaConnect** CR to the OpenShift Kafka Connect environment by entering the following command:

```
oc create -f dbz-connect.yaml
```

The command adds a Kafka Connect instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium MySQL connector instance.
You configure a Debezium MySQL connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce

events for a subset of the schemas and tables, or it might set properties so that Debezium ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed.

The following example configures a Debezium connector that connects to a MySQL host, **192.168.99.100**, on port **3306**, and captures changes to the **inventory** database. **dbserver1** is the server's logical name.

MySQL inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector (1)
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 1 (2)
  config: (3)
    database.hostname: mysql (4)
    database.port: 3306
    database.user: debezium
    database.password: dbz
    database.server.id: 184054 (5)
    database.server.name: dbserver1 (6)
    database.include.list: inventory (7)
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 (8)
    database.history.kafka.topic: schema-changes.inventory (9)
```

Table 5.24. Descriptions of connector configuration settings

Item	Description
1	The name of the connector.
2	Only one task should operate at any one time. Because the MySQL connector reads the MySQL server's **binlog**, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.
3	The connector's configuration.
4	The database host, which is the name of the container running the MySQL server (**mysql**).
5	Unique ID of the connector.

Item	Description
6	Logical name of the MySQL server or cluster. This name is used as the prefix for all Kafka topics that receive change event records.
7	Changes in only the **inventory** database are captured.
8	The list of Kafka brokers that this connector will use to write and recover DDL statements to the database history topic. Upon restart, the connector recovers the schemas of the database that existed at the point in time in the binlog when the connector should begin reading.
9	The name of the database history topic. This topic is for internal use only and should not be used by consumers.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **inventory-connector.yaml** file, you would run the following command:

```
oc apply -f inventory-connector.yaml
```

The preceding command registers **inventory-connector** and the connector starts to run against the **inventory** database as defined in the **KafkaConnector** CR.

For the complete list of the configuration properties that you can set for the Debezium MySQL connector, see MySQL connector configuration properties.

Results

After the connector starts, it performs a consistent snapshot of the MySQL databases that the connector is configured for. The connector then starts generating data change events for row-level operations and streaming change event records to Kafka topics.

5.5.4. Verifying that the Debezium MySQL connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.

- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-mysql**.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
     a. Enter the following command:

```
oc describe KafkaConnector <connector-name> -n <project>
```

     For example,

```
oc describe KafkaConnector inventory-connector-mysql -n debezium
```

     The command returns status information that is similar to the following output:

Example 5.3. KafkaConnector resource status

```
Name:         inventory-connector-mysql
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  10.131.1.124:8083
    Name:         inventory-connector-mysql
    Tasks:
      Id:               0
```

```
    State:           RUNNING
    worker_id:       10.131.1.124:8083
  Type:              source
 Observed Generation:  1
 Tasks Max:           1
 Topics:
  inventory_connector_mysql
  inventory_connector_mysql.inventory.addresses
  inventory_connector_mysql.inventory.customers
  inventory_connector_mysql.inventory.geom
  inventory_connector_mysql.inventory.orders
  inventory_connector_mysql.inventory.products
  inventory_connector_mysql.inventory.products_on_hand
Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-mysql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

   - From a terminal window:

     a. Enter the following command:

     ```
     oc get kafkatopics
     ```

     The command returns status information that is similar to the following output:

     Example 5.4. KafkaTopic resource status

     ```
     NAME                                                                CLUSTER
     PARTITIONS   REPLICATION FACTOR   READY
     connect-cluster-configs                                              debezium-
     kafka-cluster   1            1                 True
     connect-cluster-offsets                                              debezium-
     kafka-cluster   25           1                 True
     connect-cluster-status                                               debezium-
     kafka-cluster   5            1                 True
     consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
     debezium-kafka-cluster   50           1                 True
     inventory-connector-mysql---a96f69b23d6118ff415f772679da623fbbb99421
     debezium-kafka-cluster   1            1                 True
     inventory-connector-mysql.inventory.addresses---
     1b6beaf7b2eb57d177d92be90ca2b210c9a56480         debezium-kafka-cluster
     ```

```
1          1                 True
inventory-connector-mysql.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1                 True
inventory-connector-mysql.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5              debezium-kafka-cluster
1          1                 True
inventory-connector-mysql.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d             debezium-kafka-cluster
1          1                 True
inventory-connector-mysql.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1                 True
inventory-connector-mysql.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1                 True
schema-changes.inventory
debezium-kafka-cluster   1          1                 True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1          1                 True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1                 True
```

3. Check topic content.

   - From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

   For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_mysql.inventory.products_on_hand
```

   The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_mysql.inventory.addresses**.

   For each event in the topic, the command returns information that is similar to the following output:

Example 5.5. Content of a Debezium change event

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_conne
ctor_mysql.inventory.products_on_hand.Key"},"payload":{"product_id":101}} {"schema":
```

{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_mysql.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_mysql.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connecto
r.mysql.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transacti
on"}],"optional":false,"name":"inventory_connector_mysql.inventory.products_on_hand.Env
elope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.7.Final-redhat-
00001","connector":"mysql","name":"inventory_connector_mysql","ts_ms":1638985247805,"
snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"
gtid":null,"file":"mysql-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

5.5.5. Descriptions of Debezium MySQL connector configuration properties

The Debezium MySQL connector has numerous configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values. Information about the properties is organized as follows:

- Required connector configuration properties
- Advanced connector configuration properties
- Database history connector configuration properties that control how Debezium processes events that it reads from the database history topic.
    - Pass-through database history properties

- Pass-through database driver properties that control the behavior of the database driver.

The following configuration properties are *required* unless a default value is available.

Table 5.25. Required Debezium MySQL connector configuration properties

Property	Default	Description
name	No default	Unique name for the connector. Attempting to register again with the same name fails. This property is required by all Kafka Connect connectors.
connector.class	No default	The name of the Java class for the connector. Always specify **io.debezium.connector.mysql.MySqlConnector** for the MySQL connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The MySQL connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname	No default	IP address or host name of the MySQL database server.
database.port	**3306**	Integer port number of the MySQL database server.
database.user	No default	Name of the MySQL user to use when connecting to the MySQL database server.
database.password	No default	Password to use when connecting to the MySQL database server.

Property	Default	Description
database.server.name	No default	Logical name that identifies and provides a namespace for the particular MySQL database server/cluster in which Debezium is capturing changes. The logical name should be unique across all other connectors, since it is used as a prefix for all Kafka topic names that receive events emitted by this connector. Only alphanumeric characters, hyphens, dots and underscores must be used in the database server logical name. + **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value. The connector is also unable to recover its database history topic.
database.server.id	*random*	A numeric ID of this database client, which must be unique across all currently-running database processes in the MySQL cluster. This connector joins the MySQL database cluster as another server (with this unique ID) so it can read the binlog. By default, a random number between 5400 and 6400 is generated, though the recommendation is to explicitly set a value.

Property	Default	Description
database.include.list	*empty string*	An optional, comma-separated list of regular expressions that match the names of the databases for which to capture changes. The connector does not capture changes in any database whose name is not in **database.include.list**. By default, the connector captures changes in all databases. Do not also set the **database.exclude.list** connector confiuration property.
database.exclude.list	*empty string*	An optional, comma-separated list of regular expressions that match the names of databases for which you do not want to capture changes. The connector captures changes in any database whose name is not in the **database.exclude.list**. Do not also set the **database.include.list** connector configuration property.
table.include.list	*empty string*	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers of tables whose changes you want to capture. The connector does not capture changes in any table not included in **table.include.list**. Each identifier is of the form *databaseName.tableName*. By default, the connector captures changes in every non-system table in each database whose changes are being captured. Do not also specify the **table.exclude.list** connector configuration property.
table.exclude.list	*empty string*	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you do not want to capture. The connector captures changes in any table not included in **table.exclude.list**. Each identifier is of the form *databaseName.tableName*. Do not also specify the **table.include.list** connector configuration property.
column.exclude.list	*empty string*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to exclude from change event record values. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*.
column.include.list	*empty string*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to include in change event record values. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*.

Property	Default	Description
column.truncate.to.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event record values if the field values are longer than the specified number of characters. You can configure multiple properties with different lengths in a single configuration. The length must be a positive integer. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*.
column.mask.with.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*) characters. You can configure multiple properties with different lengths in a single configuration. Each length must be a positive integer or zero. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*.

Property	Default	Description
column.mask.hash.*hashAlgorithm*.with.salt.*salt*; column.mask.hash.v2.*hashAlgorithm*.with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form ***<databaseName>.<tableName>.<columnName>***. In the resulting change event record, the values for the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. In the following example, **CzQMA0cB5K** is a randomly selected salt. `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts. Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting data set might not be completely masked. Hashing strategy version 2 should be used to ensure fidelity if the value is being hashed in different places or systems.

Property	Default	Description
column.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change event records. These schema parameters: **__Debezium.source.column.type** **__Debezium.source.column.length** **__Debezium.source.column.scale** are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of one of these forms: *databaseName.tableName.columnName* *databaseName.schemaName.tableName.columnName*
datatype.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change event records. These schema parameters: **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of one of these forms: *databaseName.tableName.typeName* *databaseName.schemaName.tableName.typeName* See how MySQL connectors map data types for the list of MySQL-specific data type names.

Property	Default	Description
time.precision.mode	**adaptive_time_microseconds**	Time, date, and timestamps can be represented with different kinds of precision, including: **adaptive_time_microseconds** (the default) captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type, with the exception of TIME type fields, which are always captured as microseconds. **connect** always represents time and timestamp values using Kafka Connect's built-in representations for Time, Date, and Timestamp, which use millisecond precision regardless of the database columns' precision.
decimal.handling.mode	**precise**	Specifies how the connector should handle values for **DECIMAL** and **NUMERIC** columns: **precise** (the default) represents them precisely using **java.math.BigDecimal** values represented in change events in a binary form. **double** represents them using **double** values, which may result in a loss of precision but is easier to use. **string** encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.
bigint.unsigned.handling.mode	**long**	Specifies how BIGINT UNSIGNED columns should be represented in change events. Possible settings are: **long** represents values by using Java's **long**, which might not offer the precision but which is easy to use in consumers. **long** is usually the preferred setting. **precise** uses **java.math.BigDecimal** to represent values, which are encoded in the change events by using a binary representation and Kafka Connect's **org.apache.kafka.connect.data.Decimal** type. Use this setting when working with values larger than 2^63, because these values cannot be conveyed by using **long**.

Property	Default	Description
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded by using a key that contains the database name and whose value includes the DDL statement(s). This is independent of how the connector internally records database history.
include.schema.comments	**false**	Boolean value that specifies whether the connector should parse and publish table and column comments on metadata objects. Enabling this option will bring the implications on memory usage. The number and size of logical schema objects is what largely impacts how much memory is consumed by the Debezium connectors, and adding potentially large string data to each of them can potentially be quite expensive.
include.query	**false**	Boolean value that specifies whether the connector should include the original SQL query that generated the change event. If you set this option to **true** then you must also configure MySQL with the **binlog_rows_query_log_events** option set to **ON**. When **include.query** is **true**, the query is not present for events that the snapshot process generates. Setting **include.query** to **true** might expose tables or fields that are explicitly excluded or masked by including the original SQL statement in the change event. For this reason, the default setting is **false**.
event.deserialization.failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during deserialization of binlog events. **fail** propagates the exception, which indicates the problematic event and its binlog offset, and causes the connector to stop. **warn** logs the problematic event and its binlog offset and then skips the event. **ignore** passes over the problematic event and does not log anything.

Property	Default	Description
inconsistent.schema.handling.mode	**fail**	Specifies how the connector should react to binlog events that relate to tables that are not present in internal schema representation. That is, the internal representation is not consistent with the database. **fail** throws an exception that indicates the problematic event and its binlog offset, and causes the connector to stop. **warn** logs the problematic event and its binlog offset and skips the event. **skip** passes over the problematic event and does not log anything.
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.
max.queue.size	**8192**	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.
max.queue.size.in.bytes	**0**	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.

Property	Default	Description
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait for new change events to appear before it starts processing a batch of events. Defaults to 1000 milliseconds, or 1 second.
connect.timeout.ms	**30000**	A positive integer value that specifies the maximum time in milliseconds this connector should wait after trying to connect to the MySQL database server before timing out. Defaults to 30 seconds.
gtid.source.includes	No default	A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources that match one of these include patterns are used. Do not also specify a setting for **gtid.source.excludes**.
gtid.source.excludes	No default	A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources that do not match any of these exclude patterns are used. Do not also specify a value for **gtid.source.includes**.
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. **true** - a delete operation is represented by a*delete* event and a subsequent tombstone event. **false** - only a *delete* event is emitted. After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case log compaction is enabled for the topic.

Property	Default	Description
message.key.columns	*n/a*	A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables. By default, Debezium uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns. To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format: ***<fully-qualified_tableName>:<keyColumn>,<keyColumn>*** To base a table key on multiple column names, insert commas between the column names. Each fully-qualified table name is a regular expression in the following format: ***<databaseName>.<tableName>*** The property can include entries for multiple tables. Use a semicolon to separate table entries in the list. The following example sets the message key for the tables **inventory.customers** and **purchase.orders**: **inventory.customers:pk1,pk2; (.*).purchaseorders:pk3,pk4** For the table **inventory.customer**, the columns **pk1** and **pk2** are specified as the message key. For the **purchaseorders** tables in any database, the columns **pk3** and **pk4** server as the message key. There is no limit to the number of columns that you use to create custom message keys. However, it's best to use the minimum number that are required to specify a unique key.

Property	Default	Description
binary.handling.mode	bytes	Specifies how binary columns, for example, **blob**, **binary**, **varbinary**, should be represented in change events. Possible settings: **bytes** represents binary data as a byte array. **base64** represents binary data as a base64-encoded String. **hex** represents binary data as a hex-encoded (base16) String.
schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.

Advanced MySQL connector configuration properties

The following table describes advanced MySQL connector properties. The default values for these properties rarely need to be changed. Therefore, you do not need to specify them in the connector configuration.

Table 5.26. Descriptions of MySQL connector advanced configuration properties

Property	Default	Description
connect.keep.alive	**true**	A Boolean value that specifies whether a separate thread should be used to ensure that the connection to the MySQL server/cluster is kept alive.

Property	Default	Description

converters	No default	Enumerates a comma-separated list of the symbolic names of the custom converter instances that the connector can use.
		For example, **boolean**.
		This property is required to enable the connector to use a custom converter.
		
		For each converter that you configure for a connector, you must also add a **.type** property, which specifies the fully-qualifed name of the class that implements the converter interface. The **.type** property uses the following format:
		
		***<converterSymbolicName>*.type**
		
		For example,
		
		`boolean.type: io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter`
		
		If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate these additional configuration parameter with a converter, prefix the paraemeter name with the symbolic name of the converter.
		
		For example, to define a **selector** parameter that specifies the subset of columns that the **boolean** converter processes, add the following property:
		
		`boolean.selector=db1.table1.*, db1.table2.column1`

table.ignore.builtin	**true**	A Boolean value that specifies whether built-in system tables should be ignored. This applies regardless of the table include and exclude lists. By default, system tables are excluded from having their changes captured, and no events are generated when changes are made to any system tables.

Property	Default	Description
database.ssl.mode	**disabled**	Specifies whether to use an encrypted connection. Possible settings are: **disabled** specifies the use of an unencrypted connection. **preferred** establishes an encrypted connection if the server supports secure connections. If the server does not support secure connections, falls back to an unencrypted connection. **required** establishes an encrypted connection or fails if one cannot be made for any reason. **verify_ca** behaves like **required** but additionally it verifies the server TLS certificate against the configured Certificate Authority (CA) certificates and fails if the server TLS certificate does not match any valid CA certificates. **verify_identity** behaves like **verify_ca** but additionally verifies that the server certificate matches the host of the remote connection.

Property	Default	Description
snapshot.mode	**initial**	Specifies the criteria for running a snapshot when the connector starts. Possible settings are: **initial** - the connector runs a snapshot only when no offsets have been recorded for the logical server name. **initial_only** - the connector runs a snapshot only when no offsets have been recorded for the logical server name and then stops; i.e. it will not read change events from the binlog. **when_needed** - the connector runs a snapshot upon startup whenever it deems it necessary. That is, when no offsets are available, or when a previously recorded offset specifies a binlog location or GTID that is not available in the server. **never** - the connector never uses snapshots. Upon first startup with a logical server name, the connector reads from the beginning of the binlog. Configure this behavior with care. It is valid only when the binlog is guaranteed to contain the entire history of the database. **schema_only** - the connector runs a snapshot of the schemas and not the data. This setting is useful when you do not need the topics to contain a consistent snapshot of the data but need them to have only the changes since the connector was started. **schema_only_recovery** - this is a recovery setting for a connector that has already been capturing changes. When you restart the connector, this setting enables recovery of a corrupted or lost database history topic. You might set it periodically to "clean up" a database history topic that has been growing unexpectedly. Database history topics require infinite retention.

Property	Default	Description
snapshot.locking.mode	**minimal**	Controls whether and how long the connector holds the global MySQL read lock, which prevents any updates to the database, while the connector is performing a snapshot. Possible settings are: **minimal** - the connector holds the global read lock for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global read lock is no longer held and other MySQL clients are updating the database. **minimal_percona** - the connector holds the global backup lock for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global backup lock is no longer held and other MySQL clients are updating the database. This mode does not flush tables to disk, is not blocked by long-running reads, and is available only in Percona Server. **extended** - blocks all writes for the duration of the snapshot. Use this setting if there are clients that are submitting operations that MySQL excludes from REPEATABLE READ semantics. **none** - prevents the connector from acquiring any table locks during the snapshot. While this setting is allowed with all snapshot modes, it is safe to use if and *only* if no schema changes are happening while the snapshot is running. For tables defined with MyISAM engine, the tables would still be locked despite this property being set as MyISAM acquires a table lock. This behavior is unlike InnoDB engine, which acquires row level locks.

Property	Default	Description
snapshot.include.collection.list	All tables specified in **table.include.list**	An optional, comma-separated list of regular expressions that match the fully-qualified names (***<databaseName>.<tableName>***) of the tables to include in a snapshot. The specified items must be named in the connector's **table.include.list** property. This property takes effect only if the connector's **snapshot.mode** property is set to a value other than **never**. This property does not affect the behavior of incremental snapshots.

Property	Default	Description
snapshot.select.statement.overrides	No default	Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log. The property contains a comma-separated list of fully-qualified table names in the form ***<databaseName>.<tableName>***. For example, **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** For each table in the list, add a further configuration property that specifies the **SELECT** statement for the connector to run on the table when it takes a snapshot. The specified **SELECT** statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this **SELECT** statement property: **snapshot.select.statement.overrides.*<databaseName>.<tableName>***. For example, **snapshot.select.statement.overrides.customers.orders**. Example: From a **customers.orders** table that includes the soft-delete column, **delete_flag**, add the following properties if you want a snapshot to include only those records that are not soft-deleted: `"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"` In the resulting snapshot, the connector includes only the records for which **delete_flag = 0**.

Property	Default	Description
min.row.count.to.stream.results	**1000**	During a snapshot, the connector queries each table for which the connector is configured to capture changes. The connector uses each query result to produce a read event that contains data for all rows in that table. This property determines whether the MySQL connector puts results for a table into memory, which is fast but requires large amounts of memory, or streams the results, which can be slower but work for very large tables. The setting of this property specifies the minimum number of rows a table must contain before the connector streams results. To skip all table size checks and always stream all results during a snapshot, set this property to **0**.
heartbeat.interval.ms	**0**	Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages. Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern: *heartbeat.topics.prefix.server.name* For example, if the database server name is **fulfillment**, the default topic name is **__debezium-heartbeat.fulfillment**.
heartbeat.action.query	No default	Specifies a query that the connector executes on the source database when the connector sends a heartbeat message. For example, this can be used to periodically capture the state of the executed GTID set in the source database. **INSERT INTO gtid_history_table (select * from mysql.gtid_executed)**

Property	Default	Description
database.initial.statements	No default	A semicolon separated list of SQL statements to be executed when a JDBC connection, not the connection that is reading the transaction log, to the database is established. To specify a semicolon as a character in a SQL statement and not as a delimiter, use two semicolons, (**;;**). The connector might establish JDBC connections at its own discretion, so this property is ony for configuring session parameters. It is not for executing DML statements.
snapshot.delay.ms	No default	An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
snapshot.fetch.size	No default	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
snapshot.lock.timeout.ms	**10000**	Positive integer that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this time interval, the snapshot fails. See how MySQL connectors perform database snapshots.
enable.time.adjuster	**true**	Boolean value that indicates whether the connector converts a 2-digit year specification to 4 digits. Set to **false** when conversion is fully delegated to the database. MySQL allows users to insert year values with either 2-digits or 4-digits. For 2-digit values, the value gets mapped to a year in the range 1970 - 2069. The default behavior is that the connector does the conversion.
sanitize.field.names	**true** if connector configuration sets the **key.converter** or **value.converter** property to the Avro converter. **false** if not.	Indicates whether field names are sanitized to adhere to Avro naming requirements.

Property	Default	Description
skipped.operations	No default	Comma-separated list of operation types to skip during streaming. The following values are possible: **c** for inserts/create, **u** for updates, **d** for deletes. By default, no operations are skipped.
signal.data.collection	No default value	Fully-qualified name of the data collection that is used to send signals to the connector. Use the following format to specify the collection name: ***<databaseName>.<tableName>***
incremental.snapshot.allow.schema.changes	**false**	Allow schema changes during an incremental snapshot. When enabled the connector will detect schema change during an incremental snapshot and re-select a current chunk to avoid locking DDLs. Note that changes to a primary key are not supported and can cause incorrect results if performed during an incremental snapshot. Another limitation is that if a schema change affects only columns' default values, then the change won't be detected until the DDL is processed from the binlog stream. This doesn't affect the snapshot events' values, but the schema of snapshot events may have outdated defaults.
incremental.snapshot.chunk.size	**1024**	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.
provide.transaction.metadata	**false**	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify **true** if you want the connector to do this. See Transaction metadata for details.
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name; defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**.

Debezium connector database history configuration properties

Debezium provides a set of **database.history.*** properties that control how the connector interacts with the schema history topic.

The following table describes the **database.history** properties for configuring the Debezium connector.

Table 5.27. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic	No default	The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.bootstrap.servers	No default	A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. This connection is used for retrieving the database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
database.history.kafka.recovery.poll.interval.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait during startup/recovery while polling for persisted data. The default is 100ms.
database.history.kafka.query.timeout.ms	**3000**	An integer value that specifies the maximum number of milliseconds the connector should wait while fetching cluster information using Kafka admin client.
database.history.kafka.recovery.attempts	**4**	The maximum number of times that the connector should try to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is **recovery.attempts** x **recovery.poll.interval.ms**.
database.history.skip.unparseable.ddl	**false**	A Boolean value that specifies whether the connector should ignore malformed or unknown database statements or stop processing so a human can fix the issue. The safe default is **false**. Skipping should be used only with care as it can lead to data loss or mangling when the binlog is being processed.
database.history.store.only.monitored.tables.ddl *Deprecated and scheduled for removal in a future release; use **database.history.store.only.captured.tables.ddl** instead.*	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Property	Default	Description
database.history.store.only.captured.tables.ddl	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Pass-through database history properties for configuring producer and consumer clients

Debezium relies on a Kafka producer to write schema changes to database history topics. Similarly, it relies on a Kafka consumer to read from database history topics when a connector starts. You define the configuration for the Kafka producer and consumer clients by assigning values to a set of pass-through configuration properties that begin with the **database.history.producer.*** and **database.history.consumer.*** prefixes. The pass-through producer and consumer database history properties control a range of behaviors, such as how these clients secure connections with the Kafka broker, as shown in the following example:

```
database.history.producer.security.protocol=SSL
database.history.producer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.producer.ssl.keystore.password=test1234
database.history.producer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.producer.ssl.truststore.password=test1234
database.history.producer.ssl.key.password=test1234

database.history.consumer.security.protocol=SSL
database.history.consumer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.consumer.ssl.keystore.password=test1234
database.history.consumer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.consumer.ssl.truststore.password=test1234
database.history.consumer.ssl.key.password=test1234
```

Debezium strips the prefix from the property name before it passes the property to the Kafka client.

See the Kafka documentation for more details about Kafka producer configuration properties and Kafka consumer configuration properties.

Debezium connector Kafka signals configuration properties

When the MySQL connector is configured as read-only, the alternative for the signaling table is the signals Kafka topic.

Debezium provides a set of **signal.*** properties that control how the connector interacts with the Kafka signals topic.

The following table describes the **signal** properties.

Table 5.28. Kafka signals configuration properties

Property	Default	Description
signal.kafka.topic	No default	The name of the Kafka topic that the connector monitors for ad hoc signals.
signal.kafka.bootstrap.servers	No default	A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
signal.kafka.poll.timeout.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait when polling signals. The default is 100ms.

Debezium connector pass-through signals Kafka consumer client configuration properties

The Debezium connector provides for pass-through configuration of the signals Kafka consumer. Pass-through signals properties begin with the prefix **signals.consumer.***. For example, the connector passes properties such as **signal.consumer.security.protocol=SSL** to the Kafka consumer.

As is the case with the pass-through properties for database history clients, Debezium strips the prefixes from the properties before it passes them to the Kafka signals consumer.

Debezium connector pass-through database driver configuration properties

The Debezium connector provides for pass-through configuration of the database driver. Pass-through database properties begin with the prefix **database.***. For example, the connector passes properties such as **database.foobar=false** to the JDBC URL.

As is the case with the pass-through properties for database history clients, Debezium strips the prefixes from the properties before it passes them to the database driver.

5.6. MONITORING DEBEZIUM MYSQL CONNECTOR PERFORMANCE

The Debezium MySQL connector provides three types of metrics that are in addition to the built-in support for JMX metrics that Zookeeper, Kafka, and Kafka Connect provide.

- Snapshot metrics provide information about connector operation while performing a snapshot.
- Streaming metrics provide information about connector operation when the connector is reading the binlog.
- Schema history metrics provide information about the status of the connector's schema history.

Debezium monitoring documentation provides details for how to expose these metrics by using JMX.

5.6.1. Monitoring Debezium during snapshots of MySQL databases

The **MBean** is **debezium.mysql:type=connector-metrics,context=snapshot,server=*<mysql.server.name>***. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.

Attributes	Type	Description
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

The connector also provides the following additional snapshot metrics when an incremental snapshot is executed:

Attributes	Type	Description
ChunkId	**string**	The identifier of the current snapshot chunk.
ChunkFrom	**string**	The lower bound of the primary key set defining the current chunk.
ChunkTo	**string**	The upper bound of the primary key set defining the current chunk.
TableFrom	**string**	The lower bound of the primary key set of the currently snapshotted table.

Attributes	Type	Description
TableTo	**string**	The upper bound of the primary key set of the currently snapshotted table.

The Debezium MySQL connector also provides the **HoldingGlobalLock** custom snapshot metric. This metric is set to a Boolean value that indicates whether the connector currently holds a global or table write lock.

5.6.2. Monitoring Debezium MySQL connector record streaming

Transaction-related attributes are available only if binlog event buffering is enabled. See **binlog.buffer.size** in the advanced connector configuration properties for more details.

The **MBean** is **debezium.mysql:type=connector-metrics,context=streaming,server=*<mysql.server.name>***. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since the last start or metrics reset.
TotalNumberOfCreateEventsSeen	**long**	The total number of create events that this connector has seen since the last start or metrics reset.
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.

Attributes	Type	Description
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

Attributes	Type	Description
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if max.queue.size.in.bytes is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The Debezium MySQL connector also provides the following additional streaming metrics:

Table 5.29. Descriptions of additional streaming metrics

Attribute	Type	Description
BinlogFilename	string	The name of the binlog file that the connector has most recently read.
BinlogPosition	long	The most recent position (in bytes) within the binlog that the connector has read.
IsGtidModeEnabled	boolean	Flag that denotes whether the connector is currently tracking GTIDs from MySQL server.
GtidSet	string	The string representation of the most recent GTID set processed by the connector when reading the binlog.
NumberOfSkippedEvents	long	The number of events that have been skipped by the MySQL connector. Typically events are skipped due to a malformed or unparseable event from MySQL's binlog.
NumberOfDisconnects	long	The number of disconnects by the MySQL connector.
NumberOfRolledBackTransactions	long	The number of processed transactions that were rolled back and not streamed.
NumberOfNotWellFormedTransactions	long	The number of transactions that have not conformed to the expected protocol of **BEGIN** + **COMMIT**/**ROLLBACK**. This value should be**0** under normal conditions.

Attribute	Type	Description
NumberOfLargeTransactions	**long**	The number of transactions that have not fit into the look-ahead buffer. For optimal performance, this value should be significantly smaller than **NumberOfCommittedTransactions** and **NumberOfRolledBackTransactions**.

5.6.3. Monitoring Debezium MySQL connector schema history

The MBean is **debezium.mysql:type=connector-metrics,context=schema-history,server=*<mysql.server.name>***.

The following table lists the schema history metrics that are available.

Attributes	Type	Description
Status	**string**	One of **STOPPED**, **RECOVERING** (recovering history from the storage), **RUNNING** describing the state of the database history.
RecoveryStartTime	**long**	The time in epoch seconds at what recovery has started.
ChangesRecovered	**long**	The number of changes that were read during recovery phase.
ChangesApplied	**long**	the total number of schema changes applied during recovery and runtime.
MilliSecondsSinceLastRecoveredChange	**long**	The number of milliseconds that elapsed since the last change was recovered from the history store.
MilliSecondsSinceLastAppliedChange	**long**	The number of milliseconds that elapsed since the last change was applied.
LastRecoveredChange	**string**	The string representation of the last change recovered from the history store.
LastAppliedChange	**string**	The string representation of the last applied change.

5.7. HOW DEBEZIUM MYSQL CONNECTORS HANDLE FAULTS AND PROBLEMS

Debezium is a distributed system that captures all changes in multiple upstream databases; it never misses or loses an event. When the system is operating normally or being managed carefully then Debezium provides *exactly once* delivery of every change event record.

If a fault does happen then the system does not lose any events. However, while it is recovering from the fault, it might repeat some change events. In these abnormal situations, Debezium, like Kafka, provides *at least once* delivery of change events.

Details are in the following sections:

- Configuration and startup errors
- MySQL becomes unavailable
- Kafka Connect stops gracefully
- Kafka Connect process crashes
- Kafka becomes unavailable
- MySQL purges binlog files

Configuration and startup errors

In the following situations, the connector fails when trying to start, reports an error or exception in the log, and stops running:

- The connector's configuration is invalid.
- The connector cannot successfully connect to the MySQL server by using the specified connection parameters.
- The connector is attempting to restart at a position in the binlog for which MySQL no longer has the history available.

In these cases, the error message has details about the problem and possibly a suggested workaround. After you correct the configuration or address the MySQL problem, restart the connector.

MySQL becomes unavailable

If your MySQL server becomes unavailable, the Debezium MySQL connector fails with an error and the connector stops. When the server is available again, restart the connector.

However, if GTIDs are enabled for a highly available MySQL cluster, you can restart the connector immediately. It will connect to a different MySQL server in the cluster, find the location in the server's binlog that represents the last transaction, and begin reading the new server's binlog from that specific location.

If GTIDs are not enabled, the connector records the binlog position of only the MySQL server to which it was connected. To restart from the correct binlog position, you must reconnect to that specific server.

Kafka Connect stops gracefully

When Kafka Connect stops gracefully, there is a short delay while the Debezium MySQL connector tasks are stopped and restarted on new Kafka Connect processes.

Kafka Connect process crashes

If Kafka Connect crashes, the process stops and any Debezium MySQL connector tasks terminate without their most recently-processed offsets being recorded. In distributed mode, Kafka Connect restarts the connector tasks on other processes. However, the MySQL connector resumes from the last offset recorded by the earlier processes. This means that the replacement tasks might generate some of the same events processed prior to the crash, creating duplicate events.

Each change event message includes source-specific information that you can use to identify duplicate events, for example:

- Event origin
- MySQL server's event time
- The binlog file name and position
- GTIDs (if used)

Kafka becomes unavailable

The Kafka Connect framework records Debezium change events in Kafka by using the Kafka producer API. If the Kafka brokers become unavailable, the Debezium MySQL connector pauses until the connection is reestablished and the connector resumes where it left off.

MySQL purges binlog files

If the Debezium MySQL connector stops for too long, the MySQL server purges older binlog files and the connector's last position may be lost. When the connector is restarted, the MySQL server no longer has the starting point and the connector performs another initial snapshot. If the snapshot is disabled, the connector fails with an error.

See snapshots for details about how MySQL connectors perform initial snapshots.

CHAPTER 6. DEBEZIUM CONNECTOR FOR ORACLE

Debezium's Oracle connector captures and records row-level changes that occur in databases on an Oracle server, including tables that are added while the connector is running. You can configure the connector to emit change events for specific subsets of schemas and tables, or to ignore, mask, or truncate values in specific columns.

For information about the Oracle Database versions that are compatible with this connector, see the Debezium Supported Configurations page .

Debezium ingests change events from Oracle by using the native LogMiner database package .

Information and procedures for using a Debezium Oracle connector are organized as follows:

- Section 6.1, "How Debezium Oracle connectors work"
- Section 6.2, "Descriptions of Debezium Oracle connector data change events"
- Section 6.3, "How Debezium Oracle connectors map data types"
- Section 6.4, "Setting up Oracle to work with Debezium"
- Section 6.5, "Deployment of Debezium Oracle connectors"
- Section 6.6, "Descriptions of Debezium Oracle connector configuration properties"
- Section 6.7, "Monitoring Debezium Oracle connector performance"
- Section 6.8, "How Debezium Oracle connectors handle faults and problems"

6.1. HOW DEBEZIUM ORACLE CONNECTORS WORK

To optimally configure and run a Debezium Oracle connector, it is helpful to understand how the connector performs snapshots, streams change events, determines Kafka topic names, uses metadata, and implements event buffering.

For more information, see the following topics:

- Section 6.1.1, "How Debezium Oracle connectors perform database snapshots"
- Section 6.1.2, "Default names of Kafka topics that receive Debezium Oracle change event records"
- Section 6.1.3, "How Debezium Oracle connectors expose database schema changes"
- Section 6.1.4, "Debezium Oracle connector-generated events that represent transaction boundaries"
- Section 6.1.5, "How the Debezium Oracle connector uses event buffering"

6.1.1. How Debezium Oracle connectors perform database snapshots

Typically, the redo logs on an Oracle server are configured to not retain the complete history of the database. As a result, the Debezium Oracle connector cannot retrieve the entire history of the database from the logs. To enable the connector to establish a baseline for the current state of the database, the first time that the connector starts, it performs an initial *consistent snapshot* of the database.

You can customize the way that the connector creates snapshots by setting the value of the **snapshot.mode** connector configuration property. By default, the connector's snapshot mode is set to **initial**.

Default connector workflow for creating an initial snapshot

When the snapshot mode is set to the default, the connector completes the following tasks to create a snapshot:

1. Determines the tables to be captured.
2. Obtains a **ROW SHARE MODE** lock on each of the monitored tables to prevent structural changes from occurring during creation of the snapshot. Debezium holds the locks for only a short time.
3. Reads the current system change number (SCN) position from the server's redo log.
4. Captures the structure of all relevant tables.
5. Releases the locks obtained in Step 2.
6. Scans all of the relevant database tables and schemas as valid at the SCN position that was read in Step 3 (**SELECT * FROM … AS OF SCN 123**), generates a **READ** event for each row, and then writes the event records to the table-specific Kafka topic.
7. Records the successful completion of the snapshot in the connector offsets.

After the snapshot process begins, if the process is interrupted due to connector failure, rebalancing, or other reasons, the process restarts after the connector restarts. After the connector completes the initial snapshot, it continues streaming from the position that it read in Step 3 so that it does not miss any updates. If the connector stops again for any reason, after it restarts, it resumes streaming changes from where it previously left off.

Table 6.1. Settings for snapshot.mode connector configuration property

Setting	Description
initial	The connector performs a database snapshot as described in the default workflow for creating an initial snapshot. After the snapshot completes, the connector begins to stream event records for subsequent database changes.
initial_only	The connector performs a database snapshot and stops before streaming any change event records, not allowing any subsequent change events to be captured.
schema_only	The connector captures the structure of all relevant tables, performing all of the steps described in the default snapshot workflow, except that it does not create **READ** events to represent the data set at the point of the connector's start-up (Step 6).

Setting	Description
schema_only_recovery	Set this option to restore a database history topic that is lost or corrupted. After a restart, the connector runs a snapshot that rebuilds the topic from the source tables. You can also set the property to periodically prune a database history topic that experiences unexpected growth. WARNING: Do not use this mode to perform a snapshot if schema changes were committed to the database after the last connector shutdown.

For more information, see **snapshot.mode** in the table of connector configuration properties.

6.1.1.1. Ad hoc snapshots

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing table data, Debezium includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of tables.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling tables. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the tables to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the tables in the database.

You specify the tables to capture by sending an **execute-snapshot** message to the signaling table. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the tables to include in the snapshot, as described in the following table:

Table 6.2. Example of an ad hoc execute-snapshot signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.

Field	Default	Value
data-collections	*N/A*	An array that contains the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling table. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each table. Based on the number of entries in the table, and the configured chunk size, Debezium divides the table into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

6.1.1.2. Incremental snapshots

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each table in phases, in a series of configurable chunks. You can specify the tables that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each table row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the other.
- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the table from the beginning.
- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a table to its **table.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each table by primary key and then splits the table into chunks based on the configured chunk size. Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying table records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a table row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same table row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified table chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the table's Kafka topic.

The connector repeats the process for each snapshot chunk.

Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling table on the source database. You submit signals to the table as SQL **INSERT** queries. After Debezium detects the change in the signaling table, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the tables to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.

NOTE

If the name of a table that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the table to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling table.

The following table describes the these parameters:

Table 6.3. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling table on the source database
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.

Value	Description
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of table names to include in the snapshot. The array lists tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

WARNING

The Debezium connector for Oracle does not support schema changes while an incremental snapshot is running.

6.1.2. Default names of Kafka topics that receive Debezium Oracle change event records

By default, the Oracle connector writes change events for all **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table. The connector uses the following convention to name change event topics:

serverName.schemaName.tableName

The following list provides definitions for the components of the default name:

serverName

The logical name of the server as specified by the **database.server.name** connector configuration property.

schemaName

The name of the schema in which the operation occurred.

tableName

The name of the table in which the operation occurred.

For example, if **fulfillment** is the server name, **inventory** is the schema name, and the database contains tables with the names **orders**, **customers**, and **products**, the Debezium Oracle connector emits events to the following Kafka topics, one for each table in the database:

```
fulfillment.inventory.orders
fulfillment.inventory.customers
fulfillment.inventory.products
```

The connector applies similar naming conventions to label its internal database history topics, schema change topics, and transaction metadata topics.

If the default topic name do not meet your requirements, you can configure custom topic names. To configure custom topic names, you specify regular expressions in the logical topic routing SMT. For more information about using the logical topic routing SMT to customize topic naming, see Topic routing.

6.1.3. How Debezium Oracle connectors expose database schema changes

You can configure a Debezium Oracle connector to produce schema change events that describe structural changes that are applied to captured tables in the database. The connector writes schema change events to a Kafka topic named ***<serverName>***, where ***serverName*** is the logical server name that is specified in the **database.server.name** configuration property.

Debezium emits a new message to this topic whenever it streams data from a new table.

Messages that the connector sends to the schema change topic contain a payload, and, optionally, also contain the schema of the change event message. The payload of a schema change event message includes the following elements:

ddl

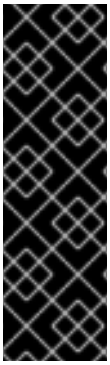
Provides the SQL **CREATE**, **ALTER**, or **DROP** statement that results in the schema change.

databaseName

The name of the database to which the statements are applied. The value of **databaseName** serves as the message key.

tableChanges

A structured representation of the entire table schema after the schema change. The **tableChanges** field contains an array that includes entries for each column of the table. Because the structured representation presents data in JSON or Avro format, consumers can easily read messages without first processing them through a DDL parser.

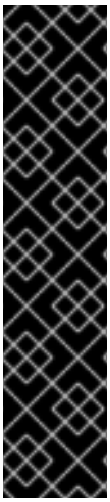


IMPORTANT

By default, the connector uses the **ALL_TABLES** database view to identify the table names to store in the schema history topic. Within that view, the connector can access data only from tables that are available to the user account through which it connects to the database. You can modify settings so that the schema history topic stores a different subset of tables. Use one of the following methods to alter the set of tables that the topic stores: * Change the permissions of the account that Debezium uses to access the database so that a different set of tables are visible in the **ALL_TABLES** view. * Set the connector property **database.history.store.only.captured.tables.ddl** to **true**.

IMPORTANT

When the connector is configured to capture a table, it stores the history of the table's schema changes not only in the schema change topic, but also in an internal database history topic. The internal database history topic is for connector use only and it is not intended for direct use by consuming applications. Ensure that applications that require notifications about schema changes consume that information only from the schema change topic.



IMPORTANT

Never partition the database history topic. For the database history topic to function correctly, it must maintain a consistent, global order of the event records that the connector emits to it.

To ensure that the topic is not split among partitions, set the partition count for the topic by using one of the following methods:

- If you create the database history topic manually, specify a partition count of **1**.
- If you use the Apache Kafka broker to create the database history topic automatically, the topic is created, set the value of the Kafka **num.partitions** configuration option to **1**.

Example: Message emitted to the Oracle connector schema change topic

The following example shows a typical schema change message in JSON format. The message contains a logical representation of the table schema.

```
{
  "schema": {
  ...
  },
  "payload": {
    "source": {
      "version": "1.9.7.Final",
      "connector": "oracle",
      "name": "server1",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "ORCLPDB1",
      "schema": "DEBEZIUM",
      "table": "CUSTOMERS",
      "txId" : null,
      "scn" : "1513734",
      "commit_scn": "1513754",
      "lcr_position" : null,
      "rs_id": "001234.00012345.0124",
      "ssn": 1,
      "redo_thread": 1
    },
    "databaseName": "ORCLPDB1", (1)
    "schemaName": "DEBEZIUM", //
    "ddl": "CREATE TABLE \"DEBEZIUM\".\"CUSTOMERS\" \n    (    \"ID\" NUMBER(9,0) NOT NULL
ENABLE, \n    \"FIRST_NAME\" VARCHAR2(255), \n    \"LAST_NAME" VARCHAR2(255), \n
\"EMAIL\" VARCHAR2(255), \n     PRIMARY KEY (\"ID\") ENABLE, \n     SUPPLEMENTAL LOG
DATA (ALL) COLUMNS\n   ) SEGMENT CREATION IMMEDIATE \n  PCTFREE 10 PCTUSED 40
INITRANS 1 MAXTRANS 255 \n NOCOMPRESS LOGGING\n  STORAGE(INITIAL 65536 NEXT
1048576 MINEXTENTS 1 MAXEXTENTS 2147483645\n  PCTINCREASE 0 FREELISTS 1
FREELIST GROUPS 1\n  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT
CELL_FLASH_CACHE DEFAULT)\n  TABLESPACE \"USERS\" ", (2)
    "tableChanges": [ (3)
      {
        "type": "CREATE", (4)
        "id": "\"ORCLPDB1\".\"DEBEZIUM\".\"CUSTOMERS\"", (5)
        "table": { (6)
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ (7)
            "ID"
          ],
          "columns": [ (8)
            {
              "name": "ID",
              "jdbcType": 2,
              "nativeType": null,
              "typeName": "NUMBER",
              "typeExpression": "NUMBER",
              "charsetName": null,
              "length": 9,
              "scale": 0,
              "position": 1,
              "optional": false,
              "autoIncremented": false,
              "generated": false
```

```
            },
            {
              "name": "FIRST_NAME",
              "jdbcType": 12,
              "nativeType": null,
              "typeName": "VARCHAR2",
              "typeExpression": "VARCHAR2",
              "charsetName": null,
              "length": 255,
              "scale": null,
              "position": 2,
              "optional": false,
              "autoIncremented": false,
              "generated": false
            },
            {
              "name": "LAST_NAME",
              "jdbcType": 12,
              "nativeType": null,
              "typeName": "VARCHAR2",
              "typeExpression": "VARCHAR2",
              "charsetName": null,
              "length": 255,
              "scale": null,
              "position": 3,
              "optional": false,
              "autoIncremented": false,
              "generated": false
            },
            {
              "name": "EMAIL",
              "jdbcType": 12,
              "nativeType": null,
              "typeName": "VARCHAR2",
              "typeExpression": "VARCHAR2",
              "charsetName": null,
              "length": 255,
              "scale": null,
              "position": 4,
              "optional": false,
              "autoIncremented": false,
              "generated": false
            }
          ]
        }
      }
    ]
  }
}
```


Table 6.4. Descriptions of fields in messages emitted to the schema change topic

Item	Field name	Description

Item	Field name	Description
1	**databaseName** **schemaName**	Identifies the database and the schema that contains the change.
2	**ddl**	This field contains the DDL that is responsible for the schema change.
3	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.
4	**type**	Describes the kind of change. The **type** can be set to one of the following values: **CREATE** Table created. **ALTER** Table modified. **DROP** Table deleted.
5	**id**	Full identifier of the table that was created, altered, or dropped. In the case of a table rename, this identifier is a concatenation of ***<old>,<new>*** table names.
6	**table**	Represents table metadata after the applied change.
7	**primaryKeyColumnNames**	List of columns that compose the table's primary key.
8	**columns**	Metadata for each column in the changed table.

In messages that the connector sends to the schema change topic, the message key is the name of the database that contains the schema change. In the following example, the **payload** field contains the **databaseName** key:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "databaseName"
      }
    ],
    "optional": false,
    "name": "io.debezium.connector.oracle.SchemaChangeKey"
```

```
  },
  "payload": {
    "databaseName": "ORCLPDB1"
  }
}
```

6.1.4. Debezium Oracle connector-generated events that represent transaction boundaries

Debezium can generate events that represent transaction metadata boundaries and that enrich data change event messages.

LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

Database transactions are represented by a statement block that is enclosed between the **BEGIN** and **END** keywords. Debezium generates transaction boundary events for the **BEGIN** and **END** delimiters in every transaction. Transaction boundary events contain the following fields:

status

BEGIN or **END**

id

String representation of unique transaction identifier.

event_count (for **END** events)

Total number of events emmitted by the transaction.

data_collections (for **END** events)

An array of pairs of **data_collection** and **event_count** elements that indicates number of events that the connector emits for changes that originate from a data collection.

The following example shows a typical transaction boundary message:

Example: Oracle connector transaction boundary event

```
{
  "status": "BEGIN",
  "id": "5.6.641",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "5.6.641",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.CUSTOMER",
      "event_count": 1
    },
```

```
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.ORDER",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, the connector emits transaction events to the ***<database.server.name>*.transaction** topic.

6.1.4.1. How the Debezium Oracle connector enriches change event messages with transaction metadata

When transaction metadata is enabled, the data message **Envelope** is enriched with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

id

String representation of unique transaction identifier.

total_order

The absolute position of the event among all events generated by the transaction.

data_collection_order

The per-data collection position of the event among all events that were emitted by the transaction.

The following example shows a typical transaction event message:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "5.6.641",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

6.1.5. How the Debezium Oracle connector uses event buffering

Oracle writes all changes to the redo logs in the order in which they occur, including changes that are later discarded by a rollback. As a result, concurrent changes from separate transactions are intertwined. When the connector first reads the stream of changes, because it cannot immediately determine which changes are committed or rolled back, it temporarily stores the change events in an internal buffer. After a change is committed, the connector writes the change event from the buffer to Kafka. The connector drops change events that are discarded by a rollback.

You can configure the buffering mechanism that the connector uses by setting the property **log.mining.buffer.type**.

Heap

The default buffer type is configured using **memory**. Under the default **memory** setting, the connector uses the heap memory of the JVM process to allocate and manage buffered event records. If you use the **memory** buffer setting, be sure that the amount of memory that you allocate to the Java process can accommodate long-running and large transactions in your environment.

6.1.6. How the Debezium Oracle connector detects gaps in SCN values

When the Debezium Oracle connector is configured to use LogMiner, it collects change events from Oracle by using a start and end range that is based on system change numbers (SCNs). The connector manages this range automatically, increasing or decreasing the range depending on whether the connector is able to stream changes in near real-time, or must process a backlog of changes due to the volume of large or bulk transactions in the database.

Under certain circumstances, the Oracle database advances the SCN by an unusually high amount, rather than increasing the SCN value at a constant rate. Such a jump in the SCN value can occur because of the way that a particular integration interacts with the database, or as a result of events such as hot backups.

The Debezium Oracle connector relies on the following configuration properties to detect the SCN gap and adjust the mining range.

log.mining.scn.gap.detection.gap.size.min

Specifies the minimum gap size.

log.mining.scn.gap.detection.time.interval.max.ms

Specifies the maximum time interval.

The connector first compares the difference in the number of changes between the current SCN and the highest SCN in the current mining range. If the difference between the current SCN value and the highest SCN value is greater than the minimum gap size, then the connector has potentially detected a SCN gap. To confirm whether a gap exists, the connector next compares the timestamps of the current SCN and the SCN at the end of the previous mining range. If the difference between the timestamps is less than the maximum time interval, then the existence of an SCN gap is confirmed.

When an SCN gap occurs, the Debezium connector automatically uses the current SCN as the end point for the range of the current mining session. This allows the connector to quickly catch up to the real-time events without mining smaller ranges in between that return no changes because the SCN value was increased by an unexpectedly large number. When the connector performs the preceding steps in response to an SCN gap, it ignores the value that is specified by the log.mining.batch.size.max property. After the connector finishes the mining session and catches back up to real-time events, it resumes enforcement of the maximum log mining batch size.

WARNING

SCN gap detection is available only if the large SCN increment occurs while the connector is running and processing near real-time events.

6.1.7. How Debezium manages offsets in databases that change infrequently

The Debezium Oracle connector tracks system change numbers in the connector offsets so that when the connector is restarted, it can begin where it left off. These offsets are part of each emitted change event; however, when the frequency of database changes are low (every few hours or days), the offsets can become stale and prevent the connector from successfully restarting if the system change number is no longer available in the transaction logs.

For connectors that use non-CDB mode to connect to Oracle, you can enable **heartbeat.interval.ms** to force the connector to emit a heartbeat event at regular intervals so that offsets remain synchronized.

For connectors that use CDB mode to connect to Oracle, maintaining synchronization is more complicated. Not only must you set **heartbeat.interval.ms**, but it's also necessary to set **heartbeat.action.query**. Specifying both properties is required, because in CDB mode, the connector specifically tracks changes inside the PDB only. A supplementary mechanism is needed to trigger change events from within the pluggable database. At regular intervals, the heartbeat action query causes the connector to insert a new table row, or update an existing row in the pluggable database. Debezium detects the table changes and emits change events for them, ensuring that offsets remain synchronized, even in pluggable databases that process changes infrequently.

NOTE

For the connector to use the **heartbeat.action.query** with tables that are not owned by the connector user account, you must grant the connector user permission to run the necessary **INSERT** or **UPDATE** queries on those tables.

6.2. DESCRIPTIONS OF DEBEZIUM ORACLE CONNECTOR DATA CHANGE EVENTS

Every data change event that the Oracle connector emits has a key and a value. The structures of the key and value depend on the table from which the change events originate. For information about how Debezium constructs topic names, see Topic names.

WARNING

The Debezium Oracle connector ensures that all Kafka Connect *schema names* are valid Avro schema names. This means that the logical server name must start with alphabetic characters or an underscore ([a-z,A-Z,_]), and the remaining characters in the logical server name and all characters in the schema and table names must be alphanumeric characters or an underscore ([a-z,A-Z,0-9,_]). The connector automatically replaces invalid characters with an underscore character.

Unexpected naming conflicts can result when the only distinguishing characters between multiple logical server names, schema names, or table names are not valid characters, and those characters are replaced with underscores.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events might change over time, which can be difficult for topic consumers to handle. To facilitate the processing of mutable event structures, each event in Kafka Connect is self-contained.

Every message key and value has two parts: a *schema* and *payload*. The schema describes the structure of the payload, while the payload contains the actual data.

WARNING

Changes that are performed by the **SYS** or **SYSTEM** user accounts are not captured by the connector.

The following topics contain more details about data change events:

- Section 6.2.1, "About keys in Debezium Oracle connector change events"
- Section 6.2.2, "About values in Debezium Oracle connector change events"

6.2.1. About keys in Debezium Oracle connector change events

For each changed table, the change event key is structured such that a field exists for each column in the primary key (or unique key constraint) of the table at the time when the event is created.

For example, a **customers** table that is defined in the **inventory** database schema, might have the following change event key:

```
CREATE TABLE customers (
  id NUMBER(9) GENERATED BY DEFAULT ON NULL AS IDENTITY (START WITH 1001) NOT
NULL PRIMARY KEY,
  first_name VARCHAR2(255) NOT NULL,
  last_name VARCHAR2(255) NOT NULL,
  email VARCHAR2(255) NOT NULL UNIQUE
);
```

If the value of the ***<database.server.name>*.transaction** configuration property is set to **server1**, the JSON representation for every change event that occurs in the **customers** table in the database features the following key structure:

```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "int32",
                "optional": false,
                "field": "ID"
            }
        ],
        "optional": false,
        "name": "server1.INVENTORY.CUSTOMERS.Key"
    },
    "payload": {
```

```
        "ID": 1004
    }
}
```

The **schema** portion of the key contains a Kafka Connect schema that describes the content of the key portion. In the preceding example, the **payload** value is not optional, the structure is defined by a schema named **server1.DEBEZIUM.CUSTOMERS.Key**, and there is one required field named **id** of type **int32**. The value of the key's **payload** field indicates that it is indeed a structure (which in JSON is just an object) with a single **id** field, whose value is **1004**.

Therefore, you can interpret this key as describing the row in the **inventory.customers** table (output from the connector named **server1**) whose **id** primary key column had a value of **1004**.

6.2.2. About values in Debezium Oracle connector change events

The structure of a value in a change event message mirrors the structure of the message key in the change event in the message, and contains both a *schema* section and a *payload* section.

Payload of a change event value

An *envelope* structure in the payload sections of a change event value contains the following fields:

op

A mandatory field that contains a string value describing the type of operation. The **op** field in the payload of an Oracle connector change event value contains one of the following values: **c** (create or insert), **u** (update), **d** (delete), or **r** (read, which indicates a snapshot).

before

An optional field that, if present, describes the state of the row *before* the event occurred. The structure is described by the **server1.INVENTORY.CUSTOMERS.Value** Kafka Connect schema, which the **server1** connector uses for all rows in the **inventory.customers** table.

after

An optional field that, if present, contains the state of a row *after* a change occurs. The structure is described by the same **server1.INVENTORY.CUSTOMERS.Value** Kafka Connect schema that is used for the **before** field.

source

A mandatory field that contains a structure that describes the source metadata for the event. In the case of the Oracle connector, the structure includes the following fields:

- The Debezium version.
- The connector name.
- Whether the event is part of an ongoing snapshot or not.
- The transaction id (not includes for snapshots).
- The SCN of the change.
- A timestamp that indicates when the record in the source database changed (for snapshots, the timestamp indicates when the snapshot occurred).

TIP

The **commit_scn** field is optional and describes the SCN of the transaction commit that the change event participates within.

ts_ms

An optional field that, if present, contains the time (based on the system clock in the JVM that runs the Kafka Connect task) at which the connector processed the event.

Schema of a change event value

The *schema* portion of the event message's value contains a schema that describes the envelope structure of the payload and the nested fields within it.

For more information about change event values, see the following topics:

- *create* events
- *update* events
- *delete* events
- *truncate* events

create events

The following example shows the value of a *create* event value from the **customers** table that is described in the change event keys example:

```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "struct",
                "fields": [
                    {
                        "type": "int32",
                        "optional": false,
                        "field": "ID"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "FIRST_NAME"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "LAST_NAME"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "EMAIL"
                    }
```

```
        ],
        "optional": true,
        "name": "server1.DEBEZIUM.CUSTOMERS.Value",
        "field": "before"
    },
    {
        "type": "struct",
        "fields": [
            {
                "type": "int32",
                "optional": false,
                "field": "ID"
            },
            {
                "type": "string",
                "optional": false,
                "field": "FIRST_NAME"
            },
            {
                "type": "string",
                "optional": false,
                "field": "LAST_NAME"
            },
            {
                "type": "string",
                "optional": false,
                "field": "EMAIL"
            }
        ],
        "optional": true,
        "name": "server1.DEBEZIUM.CUSTOMERS.Value",
        "field": "after"
    },
    {
        "type": "struct",
        "fields": [
            {
                "type": "string",
                "optional": true,
                "field": "version"
            },
            {
                "type": "string",
                "optional": false,
                "field": "name"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "ts_ms"
            },
            {
                "type": "string",
                "optional": true,
                "field": "txId"
            },
```

```
                {
                    "type": "string",
                    "optional": true,
                    "field": "scn"
                },
                {
                    "type": "string",
                    "optional": true,
                    "field": "commit_scn"
                },
                {
                    "type": "string",
                    "optional": true,
                    "field": "rs_id"
                },
                {
                    "type": "int32",
                    "optional": true,
                    "field": "ssn"
                },
                {
                    "type": "int32",
                    "optional": true,
                    "field": "redo_thread"
                },
                {
                    "type": "boolean",
                    "optional": true,
                    "field": "snapshot"
                }
            ],
            "optional": false,
            "name": "io.debezium.connector.oracle.Source",
            "field": "source"
        },
        {
            "type": "string",
            "optional": false,
            "field": "op"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
        }
    ],
    "optional": false,
    "name": "server1.DEBEZIUM.CUSTOMERS.Envelope"
},
"payload": {
    "before": null,
    "after": {
        "ID": 1004,
        "FIRST_NAME": "Anne",
        "LAST_NAME": "Kretchmar",
        "EMAIL": "annek@noanswer.org"
```

```
        },
        "source": {
            "version": "1.9.7.Final",
            "name": "server1",
            "ts_ms": 1520085154000,
            "txId": "6.28.807",
            "scn": "2122185",
            "commit_scn": "2122185",
            "rs_id": "001234.00012345.0124",
            "ssn": 1,
            "redo_thread": 1,
            "snapshot": false
        },
        "op": "c",
        "ts_ms": 1532592105975
    }
}
```

In the preceding example, notice how the event defines the following schema:

- The *envelope* (**server1.DEBEZIUM.CUSTOMERS.Envelope**).
- The **source** structure (**io.debezium.connector.oracle.Source**, which is specific to the Oracle connector and reused across all events).
- The table-specific schemas for the **before** and **after** fields.

TIP

The names of the schemas for the **before** and **after** fields are of the form ***<logicalName>.<schemaName>.<tableName>*.Value**, and thus are entirely independent from the schemas for all other tables. As a result, when you use the Avro converter, the Avro schemas for tables in each logical source have their own evolution and history.

The **payload** portion of this event's *value*, provides information about the event. It describes that a row was created (**op=c**), and shows that the **after** field value contains the values that were inserted into the **ID**, **FIRST_NAME**, **LAST_NAME**, and **EMAIL** columns of the row.

TIP

By default, the JSON representations of events are much larger than the rows that they describe. The larger size is due to the JSON representation including both the schema and payload portions of a message. You can use the Avro Converter to decrease the size of messages that the connector writes to Kafka topics.

update events

The following example shows an *update* change event that the connector captures from the same table as the preceding *create* event.

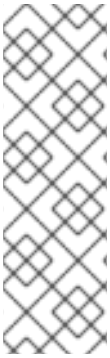
```
{
    "schema": { ... },
    "payload": {
        "before": {
            "ID": 1004,
```

```
            "FIRST_NAME": "Anne",
            "LAST_NAME": "Kretchmar",
            "EMAIL": "annek@noanswer.org"
        },
        "after": {
            "ID": 1004,
            "FIRST_NAME": "Anne",
            "LAST_NAME": "Kretchmar",
            "EMAIL": "anne@example.com"
        },
        "source": {
            "version": "1.9.7.Final",
            "name": "server1",
            "ts_ms": 1520085811000,
            "txId": "6.9.809",
            "scn": "2125544",
            "commit_scn": "2125544",
            "rs_id": "001234.00012345.0124",
            "ssn": 1,
            "redo_thread": 1,
            "snapshot": false
        },
        "op": "u",
        "ts_ms": 1532592713485
    }
}
```


The payload has the same structure as the payload of a *create* (insert) event, but the following values are different:

- The value of the **op** field is **u**, signifying that this row changed because of an update.
- The **before** field shows the former state of the row with the values that were present before the **update** database commit.
- The **after** field shows the updated state of the row, with the **EMAIL** value now set to **anne@example.com**.
- The structure of the **source** field includes the same fields as before, but the values are different, because the connector captured the event from a different position in the redo log.
- The **ts_ms** field shows the timestamp that indicates when Debezium processed the event.

The **payload** section reveals several other useful pieces of information. For example, by comparing the **before** and **after** structures, we can determine how a row changed as the result of a commit. The **source** structure provides information about Oracle's record of this change, providing traceability. It also gives us insight into when this event occurred in relation to other events in this topic and in other topics. Did it occur before, after, or as part of the same commit as another event?



NOTE

When the columns for a row's primary/unique key are updated, the value of the row's key changes. As a result, Debezium emits *three* events after such an update:

- A **DELETE** event.
- A tombstone event with the old key for the row.
- An **INSERT** event that provides the new key for the row.

delete events

The following example shows a *delete* event for the table that is shown in the preceding *create* and *update* event examples. The **schema** portion of the *delete* event is identical to the **schema** portion for those events.

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "ID": 1004,
            "FIRST_NAME": "Anne",
            "LAST_NAME": "Kretchmar",
            "EMAIL": "anne@example.com"
        },
        "after": null,
        "source": {
            "version": "1.9.7.Final",
            "name": "server1",
            "ts_ms": 1520085153000,
            "txId": "6.28.807",
            "scn": "2122184",
            "commit_scn": "2122184",
            "rs_id": "001234.00012345.0124",
            "ssn": 1,
            "redo_thread": 1,
            "snapshot": false
        },
        "op": "d",
        "ts_ms": 1532592105960
    }
}
```

The **payload** portion of the event reveals several differences when compared to the payload of a *create* or *update* event:

- The value of the **op** field is **d**, signifying that the row was deleted.
- The **before** field shows the former state of the row that was deleted with the database commit.
- The value of the **after** field is **null**, signifying that the row no longer exists.
- The structure of the **source** field includes many of the keys that exist in *create* or *update* events, but the values in the **ts_ms**, **scn**, and **txId** fields are different.

- The **ts_ms** shows a timestamp that indicates when Debezium processed this event.

The *delete* event provides consumers with the information that they require to process the removal of this row.

The Oracle connector's events are designed to work with Kafka log compaction, which allows for the removal of some older messages as long as at least the most recent message for every key is kept. This allows Kafka to reclaim storage space while ensuring the topic contains a complete dataset and can be used for reloading key-based state.

When a row is deleted, the *delete* event value shown in the preceding example still works with log compaction, because Kafka is able to remove all earlier messages that use the same key. The message value must be set to **null** to instruct Kafka to remove *all messages* that share the same key. To make this possible, by default, Debezium's Oracle connector always follows a *delete* event with a special *tombstone* event that has the same key but **null** value. You can change the default behavior by setting the connector property **tombstones.on.delete**.

truncate events

A *truncate* change event signals that a table has been truncated. The message key is **null** in this case, the message value looks like this:

```
{
    "schema": { ... },
    "payload": {
        "before": null,
        "after": null,
        "source": { (1)
            "version": "1.9.7.Final",
            "connector": "oracle",
            "name": "oracle_server",
            "ts_ms": 1638974535000,
            "snapshot": "false",
            "db": "ORCLPDB1",
            "sequence": null,
            "schema": "DEBEZIUM",
            "table": "TEST_TABLE",
            "txId": "02000a0037030000",
            "scn": "13234397",
            "commit_scn": "13271102",
            "lcr_position": null,
            "rs_id": "001234.00012345.0124",
            "ssn": 1,
            "redo_thread": 1
        },
        "op": "t", (2)
        "ts_ms": 1638974558961, (3)
        "transaction": null
    }
}
```

Table 6.5. Descriptions of *truncate* event value fields

Item	Field name	Description
1	**source**	Mandatory field that describes the source metadata for the event. In a *truncate* event value, the **source** field structure is the same as for *create*, *update*, and *delete* events for the same table, provides this metadata: • Debezium version • Connector type and name • Database and table that contains the new row • Schema name • If the event was part of a snapshot (always **false** for *truncate* events) • ID of the transaction in which the operation was performed • SCN of the operation • Timestamp for when the change was made in the database
2	**op**	Mandatory string that describes the type of operation. The **op** field value is **t**, signifying that this table was truncated.
3	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

Because *truncate* events represent changes made to an entire table, and have no message key, in topics with multiple partitions, there is no guarantee that consumers receive *truncate* events and change events (*create*, *update*, etc.) for to a table in order. For example, when a consumer reads events from different partitions, it might receive an *update* event for a table after it receives a *truncate* event for the same table. Ordering can be guaranteed only if a topic uses a single partition.

If you do not want to capture *truncate* events, use the **skipped.operations** option to filter them out.

6.3. HOW DEBEZIUM ORACLE CONNECTORS MAP DATA TYPES

When the Debezium Oracle connector detects a change in the value of a table row, it emits a change event that represents the change. Each change event record is structured in the same way as the original table, with the event record containing a field for each column value. The data type of a table column determines how the connector represents the column's values in change event fields, as shown in the tables in the following sections.

For each column in a table, Debezium maps the source data type to a *literal type* and, and in some cases, a *semantic type*, in the corresponding event field.

Literal types

Describe how the value is literally represented, using one of the following Kafka Connect schema types: **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**.

Semantic types

Describe how the Kafka Connect schema captures the *meaning* of the field, by using the name of the Kafka Connect schema for the field.

If the default data type conversions do not meet your needs, you can create a custom converter for the connector.

For some Oracle large object (CLOB, NCLOB, and BLOB) and numeric data types, you can manipulate the way that the connector performs the type mapping by changing default configuration property settings. For more information about how Debezium properties control mappings for these data types, see Binary and Character LOB types and Numeric types.

For more information about how the Debezium connector maps Oracle data types, see the following topics:

- Character types
- Binary and Character LOB types
- Numeric types
- Boolean types
- Temporal types
- ROWID types
- User-defined types
- Oracle-supplied types
- Default Values

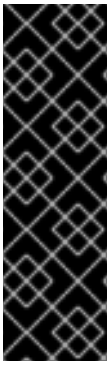
Character types

The following table describes how the connector maps basic character types.

Table 6.6. Mappings for Oracle basic character types

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
CHAR[(M)]	**STRING**	n/a
NCHAR[(M)]	**STRING**	n/a
NVARCHAR2[(M)]	**STRING**	n/a
VARCHAR[(M)]	**STRING**	n/a

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
VARCHAR2[(M)]	**STRING**	n/a



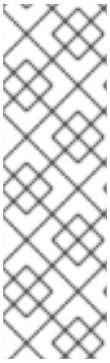
BINARY AND CHARACTER LOB TYPES

Use of the **BLOB**, **CLOB**, and **NCLOB** with the Debezium Oracle connector is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview.

The following table describes how the connector maps binary and character large object (LOB) data types.

Table 6.7. Mappings for Oracle binary and character LOB types

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
BFILE	n/a	*This data type is not supported*
BLOB	**BYTES**	Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the **binary.handling.mode** connector configuration property setting.
CLOB	**STRING**	n/a
LONG	n/a	*This data type is not supported.*
LONG RAW	n/a	*This data type is not supported.*
NCLOB	**STRING**	n/a
RAW	n/a	*This data type is not supported.*



NOTE

Oracle only supplies column values for **CLOB**, **NCLOB**, and **BLOB** data types if they're explicitly set or changed in a SQL statement. As a result, change events never contain the value of an unchanged **CLOB**, **NCLOB**, or **BLOB** column. Instead, they contain placeholders as defined by the connector property, **unavailable.value.placeholder**.

If the value of a **CLOB**, **NCLOB**, or **BLOB** column is updated, the new value is placed in the **after** element of the corresponding update change event. The **before** element contains the unavailable value placeholder.

Numeric types

The following table describes how the Debezium Oracle connector maps numeric types.

NOTE

You can modify the way that the connector maps the Oracle **DECIMAL**, **NUMBER**, **NUMERIC**, and **REAL** data types by changing the value of the connector's **decimal.handling.mode** configuration property. When the property is set to its default value of **precise**, the connector maps these Oracle data types to the Kafka Connect **org.apache.kafka.connect.data.Decimal** logical type, as indicated in the table. When the value of the property is set to **double** or **string**, the connector uses alternate mappings for some Oracle data types. For more information, see the *Semantic type and Notes* column in the following table.

Table 6.8. Mappings for Oracle numeric data types

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
BINARY_FLOAT	**FLOAT32**	n/a
BINARY_DOUBLE	**FLOAT64**	n/a
DECIMAL[(P, S)]	**BYTES** / **INT8** / **INT16** / **INT32** / **INT64**	**org.apache.kafka.connect.data.Decimal** if using **BYTES** Handled equivalently to **NUMBER** (note that S defaults to 0 for **DECIMAL**). When the **decimal.handling.mode** property is set to **double**, the connector represents **DECIMAL** values as Java **double** values with schema type **FLOAT64**. When the **decimal.handling.mode** property is set to **string**, the connector represents DECIMAL values as their formatted string representation with schema type **STRING**.
DOUBLE PRECISION	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
FLOAT[(P)]	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.
INTEGER, **INT**	**BYTES**	**org.apache.kafka.connect.data.Decimal** **INTEGER** is mapped in Oracle to NUMBER(38,0) and hence can hold values larger than any of the **INT** types could store
NUMBER[(P[, *])]	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form. When the **decimal.handling.mode** property is set to **double**, the connector represents **NUMBER** values as Java **double** values with schema type **FLOAT64**. When the **decimal.handling.mode** property is set to **string**, the connector represents **NUMBER** values as their formatted string representation with schema type **STRING**.
NUMBER(P, S <= 0)	**INT8** / **INT16** / **INT32** / **INT64**	**NUMBER** columns with a scale of 0 represent integer numbers. A negative scale indicates rounding in Oracle, for example, a scale of -2 causes rounding to hundreds. Depending on the precision and scale, one of the following matching Kafka Connect integer type is chosen: • P - S < 3, **INT8** • P - S < 5, **INT16** • P - S < 10, **INT32** • P - S < 19, **INT64** • P - S >= 19, **BYTES** (**org.apache.kafka.connect.data.Decimal**) When the **decimal.handling.mode** property is set to **double**, the connector represents **NUMBER** values as Java **double** values with schema type **FLOAT64**. When the **decimal.handling.mode** property is set to **string**, the connector represents **NUMBER** values as their formatted string representation with schema type **STRING**.
NUMBER(P, S > 0)	**BYTES**	**org.apache.kafka.connect.data.Decimal**

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
NUMERIC[(P, S)]	**BYTES** / **INT8** / **INT16** / **INT32** / **INT64**	**org.apache.kafka.connect.data.Decimal** if using **BYTES** Handled equivalently to **NUMBER** (note that S defaults to 0 for **NUMERIC**). When the **decimal.handling.mode** property is set to **double**, the connector represents **NUMERIC** values as Java **double** values with schema type **FLOAT64**. When the **decimal.handling.mode** property is set to **string**, the connector represents **NUMERIC** values as their formatted string representation with schema type **STRING**.
SMALLINT	**BYTES**	**org.apache.kafka.connect.data.Decimal** **SMALLINT** is mapped in Oracle to NUMBER(38,0) and hence can hold values larger than any of the **INT** types could store
REAL	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type**INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form. When the **decimal.handling.mode** property is set to **double**, the connector represents **REAL** values as Java **double** values with schema type **FLOAT64**. When the **decimal.handling.mode** property is set to **string**, the connector represents **REAL** values as their formatted string representation with schema type **STRING**.

Boolean types

Oracle does not provide native support for a **BOOLEAN** data type. However, it is common practice to use other data types with certain semantics to simulate the concept of a logical **BOOLEAN** data type.

To enable you to convert source columns to Boolean data types, Debezium provides a **NumberOneToBooleanConverter** custom converter that you can use in one of the following ways:

- Map all **NUMBER(1)** columns to a **BOOLEAN** type.
- Enumerate a subset of columns by using a comma-separated list of regular expressions. To use this type of conversion, you must set the **converters** configuration property with the **selector** parameter, as shown in the following example:

```
converters=boolean
boolean.type=io.debezium.connector.oracle.converters.NumberOneToBooleanConverter
boolean.selector=.*MYTABLE.FLAG,.*.IS_ARCHIVED
```

Temporal types

Other than the Oracle **INTERVAL**, **TIMESTAMP WITH TIME ZONE**, and **TIMESTAMP WITH LOCAL TIME ZONE** data types, the way that the connector converts temporal types depends on the value of the **time.precision.mode** configuration property.

When the **time.precision.mode** configuration property is set to **adaptive** (the default), then the connector determines the literal and semantic type for the temporal types based on the column's data type definition so that events *exactly* represent the values in the database:

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the UNIX epoch, and does not include timezone information.
INTERVAL DAY[(M)] TO SECOND	**FLOAT64**	**io.debezium.time.MicroDuration** The number of micro seconds for a time interval using the **365.25** / **12.0** formula for days per month average. **io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.
INTERVAL YEAR[(M)] TO MONTH	**FLOAT64**	**io.debezium.time.MicroDuration** The number of micro seconds for a time interval using the **365.25** / **12.0** formula for days per month average. **io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.
TIMESTAMP(0 - 3)	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the UNIX epoch, and does not include timezone information.
TIMESTAMP, TIMESTAMP(4 - 6)	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds since the UNIX epoch, and does not include timezone information.

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIMESTAMP(7 - 9)	**INT64**	**io.debezium.time.NanoTimestamp** Represents the number of nanoseconds since the UNIX epoch, and does not include timezone information.
TIMESTAMP WITH TIME ZONE	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp with timezone information.
TIMESTAMP WITH LOCAL TIME ZONE	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp in UTC.

When the **time.precision.mode** configuration property is set to **connect**, then the connector uses the predefined Kafka Connect logical types. This can be useful when consumers only know about the built-in Kafka Connect logical types and are unable to handle variable-precision time values. Because the level of precision that Oracle supports exceeds the level that the logical types in Kafka Connect support, if you set **time.precision.mode** to **connect**, **a loss of precision** results when the *fractional second precision* value of a database column is greater than 3:

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the UNIX epoch.
INTERVAL DAY[(M)] TO SECOND	**FLOAT64**	**io.debezium.time.MicroDuration** The number of micro seconds for a time interval using the **365.25** / **12.0** formula for days per month average. **io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
INTERVAL YEAR[(M)] TO MONTH	**FLOAT64**	**io.debezium.time.MicroDuration** The number of micro seconds for a time interval using the **365.25** / **12.0** formula for days per month average. **io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.
TIMESTAMP(0 - 3)	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the UNIX epoch, and does not include timezone information.
TIMESTAMP(4 - 6)	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the UNIX epoch, and does not include timezone information.
TIMESTAMP(7 - 9)	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the UNIX epoch, and does not include timezone information.
TIMESTAMP WITH TIME ZONE	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp with timezone information.
TIMESTAMP WITH LOCAL TIME ZONE	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp in UTC.

ROWID types

The following table describes how the connector maps ROWID (row address) data types.

Table 6.9. Mappings for Oracle ROWID data types

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
ROWID	**STRING**	n/a

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
UROWID	n/a	*This data type is not supported.*

User-defined types

Oracle enables you to define custom data types to provide flexibility when the built-in data types do not satisfy your requirements. There are a several user-defined types such as Object types, REF data types, Varrays, and Nested Tables. At this time, you cannot use the Debezium Oracle connector with any of these user-defined types.

Oracle-supplied types

Oracle provides SQL-based interfaces that you can use to define new types when the built-in or ANSI-supported types are insufficient. Oracle offers several commonly used data types to serve a broad array of purposes such as **Any**, **XML**, or **Spatial** types. At this time, you cannot use the Debezium Oracle connector with any of these data types.

Default Values

If a default value is specified for a column in the database schema, the Oracle connector will attempt to propagate this value to the schema of the corresponding Kafka record field. Most common data types are supported, including:

- Character types (**CHAR**, **NCHAR**, **VARCHAR**, **VARCHAR2**, **NVARCHAR**, **NVARCHAR2**)
- Numeric types (**INTEGER**, **NUMERIC**, etc.)
- Temporal types (**DATE**, **TIMESTAMP**, **INTERVAL**, etc.)

If a temporal type uses a function call such as **TO_TIMESTAMP** or **TO_DATE** to represent the default value, the connector will resolve the default value by making an additional database call to evaluate the function. For example, if a **DATE** column is defined with the default value of **TO_DATE('2021-01-02', 'YYYY-MM-DD')**, the column's default value will be the number of days since the UNIX epoch for that date or **18629** in this case.

If a temporal type uses the **SYSDATE** constant to represent the default value, the connector will resolve this based on whether the column is defined as **NOT NULL** or **NULL**. If the column is nullable, no default value will be set; however, if the column isn't nullable then the default value will be resolved as either **0** (for **DATE** or **TIMESTAMP(n)** data types) or **1970-01-01T00:00:00Z** (for **TIMESTAMP WITH TIME ZONE** or **TIMESTAMP WITH LOCAL TIME ZONE** data types). The default value type will be numeric except if the column is a **TIMESTAMP WITH TIME ZONE** or **TIMESTAMP WITH LOCAL TIME ZONE** in which case its emitted as a string.

6.4. SETTING UP ORACLE TO WORK WITH DEBEZIUM

The following steps are necessary to set up Oracle for use with the Debezium Oracle connector. These steps assume the use of the multi-tenancy configuration with a container database and at least one pluggable database. If you do not intend to use a multi-tenant configuration, it might be necessary to adjust the following steps.

For details about setting up Oracle for use with the Debezium connector, see the following sections:

- Section 6.4.1, "Compatibility of the Debezium Oracle connector with Oracle installation types"

- Section 6.4.2, “Schemas that the Debezium Oracle connector excludes when capturing change events”
- Section 6.4.3, “Preparing Oracle databases for use with Debezium”
- Section 6.4.4, “Resizing Oracle redo logs to accommodate the data dictionary”
- Section 6.4.5, “Creating an Oracle user for the Debezium Oracle connector”
- Section 6.4.6, “Support for Oracle standby databases”

6.4.1. Compatibility of the Debezium Oracle connector with Oracle installation types

An Oracle database can be installed either as a standalone instance or using Oracle Real Application Cluster (RAC). The Debezium Oracle connector is compatible with both types of installation.

6.4.2. Schemas that the Debezium Oracle connector excludes when capturing change events

When the Debezium Oracle connector captures tables, it automatically excludes tables from the following schemas:

- **appqossys**
- **audsys**
- **ctxsys**
- **dvsys**
- **dbsfwuser**
- **dbsnmp**
- **qsmadmin_internal**
- **lbacsys**
- **mdsys**
- **ojvmsys**
- **olapsys**
- **orddata**
- **ordsys**
- **outln**
- **sys**
- **system**
- **wmsys**
- **xdb**

To enable the connector to capture changes from a table, the table must use a schema that is not named in the preceding list.

6.4.3. Preparing Oracle databases for use with Debezium

Configuration needed for Oracle LogMiner

```
ORACLE_SID=ORACLCDB dbz_oracle sqlplus /nolog

CONNECT sys/top_secret AS SYSDBA
alter system set db_recovery_file_dest_size = 10G;
alter system set db_recovery_file_dest = '/opt/oracle/oradata/recovery_area' scope=spfile;
shutdown immediate
startup mount
alter database archivelog;
alter database open;
-- Should now "Database log mode: Archive Mode"
archive log list

exit;
```

To enable Debezium to capture the *before* state of changed database rows, you must also enable supplemental logging for captured tables or for the entire database. The following example illustrates how to configure supplemental logging for all columns in a single **inventory.customers** table.

```
ALTER TABLE inventory.customers ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

Enabling supplemental logging for all table columns increases the volume of the Oracle redo logs. To prevent excessive growth in the size of the logs, apply the preceding configuration selectively.

Minimal supplemental logging must be enabled at the database level and can be configured as follows.

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

6.4.4. Resizing Oracle redo logs to accommodate the data dictionary

Depending on the database configuration, the size and number of redo logs might not be sufficient to achieve acceptable performance. Before you set up the Debezium Oracle connector, ensure that the capacity of the redo logs is sufficient to support the database.

The capacity of the redo logs for a database must be sufficient to store its data dictionary. In general, the size of the data dictionary increases with the number of tables and columns in the database. If the redo log lacks sufficient capacity, both the database and the Debezium connector might experience performance problems.

Consult with your database administrator to evaluate whether the database might require increased log capacity.

6.4.5. Creating an Oracle user for the Debezium Oracle connector

For the Debezium Oracle connector to capture change events, it must run as an Oracle LogMiner user that has specific permissions. The following example shows the SQL for creating an Oracle user account for the connector in a multi-tenant database model.

WARNING

The connector captures database changes that are made by its own Oracle user account. However, it does not capture changes that are made by the **SYS** or **SYSTEM** user accounts.

Creating the connector's LogMiner user

```
sqlplus sys/top_secret@//localhost:1521/ORCLCDB as sysdba
  CREATE TABLESPACE logminer_tbs DATAFILE '/opt/oracle/oradata/ORCLCDB/logminer_tbs.dbf'
    SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
  exit;

sqlplus sys/top_secret@//localhost:1521/ORCLPDB1 as sysdba
  CREATE TABLESPACE logminer_tbs DATAFILE
'/opt/oracle/oradata/ORCLCDB/ORCLPDB1/logminer_tbs.dbf'
    SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
  exit;

sqlplus sys/top_secret@//localhost:1521/ORCLCDB as sysdba

  CREATE USER c##dbzuser IDENTIFIED BY dbz
    DEFAULT TABLESPACE logminer_tbs
    QUOTA UNLIMITED ON logminer_tbs
    CONTAINER=ALL;

  GRANT CREATE SESSION TO c##dbzuser CONTAINER=ALL; (1)
  GRANT SET CONTAINER TO c##dbzuser CONTAINER=ALL; (2)
  GRANT SELECT ON V_$DATABASE to c##dbzuser CONTAINER=ALL; (3)
  GRANT FLASHBACK ANY TABLE TO c##dbzuser CONTAINER=ALL; (4)
  GRANT SELECT ANY TABLE TO c##dbzuser CONTAINER=ALL; (5)
  GRANT SELECT_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL; (6)
  GRANT EXECUTE_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL; (7)
  GRANT SELECT ANY TRANSACTION TO c##dbzuser CONTAINER=ALL; (8)
  GRANT LOGMINING TO c##dbzuser CONTAINER=ALL; (9)

  GRANT CREATE TABLE TO c##dbzuser CONTAINER=ALL; (10)
  GRANT LOCK ANY TABLE TO c##dbzuser CONTAINER=ALL; (11)
  GRANT CREATE SEQUENCE TO c##dbzuser CONTAINER=ALL; (12)

  GRANT EXECUTE ON DBMS_LOGMNR TO c##dbzuser CONTAINER=ALL; (13)
  GRANT EXECUTE ON DBMS_LOGMNR_D TO c##dbzuser CONTAINER=ALL; (14)

  GRANT SELECT ON V_$LOG TO c##dbzuser CONTAINER=ALL; (15)
  GRANT SELECT ON V_$LOG_HISTORY TO c##dbzuser CONTAINER=ALL; (16)
  GRANT SELECT ON V_$LOGMNR_LOGS TO c##dbzuser CONTAINER=ALL; (17)
  GRANT SELECT ON V_$LOGMNR_CONTENTS TO c##dbzuser CONTAINER=ALL; (18)
  GRANT SELECT ON V_$LOGMNR_PARAMETERS TO c##dbzuser CONTAINER=ALL; (19)
```

```
GRANT SELECT ON V_$LOGFILE TO c##dbzuser CONTAINER=ALL; 20
GRANT SELECT ON V_$ARCHIVED_LOG TO c##dbzuser CONTAINER=ALL; 21
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO c##dbzuser CONTAINER=ALL; 22
GRANT SELECT ON V_$TRANSACTION TO c##dbzuser CONTAINER=ALL; 23

exit;
```

Table 6.10. Descriptions of permissions / grants

Item	Role name	Description
1	CREATE SESSION	Enables the connector to connect to Oracle.
2	SET CONTAINER	Enables the connector to switch between pluggable databases. This is only required when the Oracle installation has container database support (CDB) enabled.
3	SELECT ON V_$DATABASE	Enables the connector to read the **V$DATABASE** table.
4	FLASHBACK ANY TABLE	Enables the connector to perform Flashback queries, which is how the connector performs the initial snapshot of data.
5	SELECT ANY TABLE	Enables the connector to read any table.
6	SELECT_CATALOG_ROLE	Enables the connector to read the data dictionary, which is needed by Oracle LogMiner sessions.
7	EXECUTE_CATALOG_ROLE	Enables the connector to write the data dictionary into the Oracle redo logs, which is needed to track schema changes.
8	SELECT ANY TRANSACTION	Enables the snapshot process to perform a Flashback snapshot query against any transaction. When **FLASHBACK ANY TABLE** is granted, this should also be granted.
9	LOGMINING	This role was added in newer versions of Oracle as a way to grant full access to Oracle LogMiner and its packages. On older versions of Oracle that don't have this role, you can ignore this grant.
10	CREATE TABLE	Enables the connector to create its flush table in its default tablespace. The flush table allows the connector to explicitly control flushing of the LGWR internal buffers to disk.

Item	Role name	Description
11	LOCK ANY TABLE	Enables the connector to lock tables during schema snapshot. If snapshot locks are explicitly disabled via configuration, this grant can be safely ignored.
12	CREATE ANY SEQUENCE	Enables the connector to create a sequence in its default tablespace.
13	EXECUTE ON DBMS_LOGMNR	Enables the connector to run methods in the **DBMS_LOGMNR** package. This is required to interact with Oracle LogMiner. On newer versions of Oracle this is granted via the **LOGMINING** role but on older versions, this must be explicitly granted.
14	EXECUTE ON DBMS_LOGMNR_D	Enables the connector to run methods in the **DBMS_LOGMNR_D** package. This is required to interact with Oracle LogMiner. On newer versions of Oracle this is granted via the **LOGMINING** role but on older versions, this must be explicitly granted.
15 to 23	SELECT ON V_$....	Enables the connector to read these tables. The connector must be able to read information about the Oracle redo and archive logs, and the current transaction state, to prepare the Oracle LogMiner session. Without these grants, the connector cannot operate.

6.4.6. Support for Oracle standby databases

The Debezium Oracle connector cannot be used with Oracle physical or logical standby databases.

6.5. DEPLOYMENT OF DEBEZIUM ORACLE CONNECTORS

You can use either of the following methods to deploy a Debezium Oracle connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .

IMPORTANT

Due to licensing requirements, the Debezium Oracle connector archive does not include the Oracle JDBC driver that the connector requires to connect to an Oracle database. To enable the connector to access the database, you must add the driver to your connector environment. For more information, see Obtaining the Oracle JDBC driver .

Additional resources

- Section 6.6, "Descriptions of Debezium Oracle connector configuration properties"

6.5.1. Obtaining the Oracle JDBC driver

Due to licensing requirements, the Oracle JDBC driver file that Debezium requires to connect to an Oracle database is not included in the Debezium Oracle connector archive. The driver is available for download from Maven Central. Depending on the deployment method that you use, you retrieve the driver by adding a command to the Kafka Connect custom resource or to the Dockerfile that you use to build the connector image.

- If you use AMQ Streams to add the connector to your Kafka Connect image, add the Maven Central location for the driver to **builds.plugins.artifact.url** in the **KafkaConnect** custom resource as shown in Section 6.5.3, "Using AMQ Streams to deploy a Debezium Oracle connector".
- If you use a Dockerfile to build a container image for the connector, insert a **curl** command in the Dockerfile to specify the URL for downloading the required driver file from Maven Central. For more information, see Deploying a Debezium Oracle connector by building a custom Kafka Connect container image from a Dockerfile.

6.5.2. Debezium Oracle connector deployment using AMQ Streams

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.

NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.

- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

6.5.3. Using AMQ Streams to deploy a Debezium Oracle connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.
- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

Example 6.1. A dbz-connect.yaml file that defines a KafkaConnect custom resource that includes a Debezium connector

In the example that follows, the custom resource is configured to download the following artifacts:

- The Debezium Oracle connector archive.
- The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
- The Debezium scripting SMT archive and the associated language dependencies that you want to use with the Debezium connector. The SMT archive and language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.
- The Oracle JDBC driver, which is required to connect to Oracle databases, but is not included in the connector archive.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-oracle
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-oracle/1.9.7.Final-redhat-<build_number>/debezium-connector-oracle-1.9.7.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip (8)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip (9)
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar (10)
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
          - type: jar
            url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json3.0.11/groovy-json-3.0.11.jar
```

```
        - type: jar                    (11)
          url:
https://repo1.maven.org/maven2/com/oracle/database/jdbc/ojdbc8/21.1.0.0/ojdbc8-
21.1.0.0.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 6.11. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry like Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. JDBC driver files are in **.jar** format. The **type** value must match the type of the file that is referenced in the **url** field.
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.

Item	Description
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) The Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.
11	Specifies the location of the Oracle JDBC driver in Maven Central. The required driver is not included in the Debezium Oracle connector archive.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **oracle-inventory-connector.yaml**

Example 6.2. oracle-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-oracle (1)
spec:
  class: io.debezium.connector.oracle.MySqlConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: oracle.debezium-oracle.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_oracle (10)
    database.include.list: public.inventory (11)
```

Table 6.12. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.
4	The connector's configuration.
5	The address of the host database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.
8	The password for the database user account.
9	The name of the database to capture changes from.

Item	Description
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium Oracle deployment.

6.5.4. Deploying a Debezium Oracle connector by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium Oracle connector, you must build a custom Kafka Connect container image that contains the Debezium connector archive, and then push this container image to a container registry. You then need to create the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.
- A **KafkaConnector** CR that defines your Debezium Oracle connector. Apply this CR to the same OpenShift instance where you apply the **KafkaConnect** CR.

Prerequisites

- Oracle Database is running and you completed the steps to set up Oracle to work with a Debezium connector.
- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift
- Podman or Docker is installed.

- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.
- The Kafka Connect server has access to Maven Central to download the required JDBC driver for Oracle. You can also use a local copy of the driver, or one that is available from a local Maven repository or other HTTP server.
  For more information, see Obtaining the Oracle JDBC driver .

Procedure

1. Create the Debezium Oracle container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-oracle.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-oracle/1.9.7.Final-redhat-<build_number>/debezium-connector-oracle-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-oracle-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-oracle-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://repo1.maven.org/maven2/com/oracle/ojdbc/ojdbc8/21.1.0.0/ojdbc8-21.1.0.0.jar
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

   The command creates a Dockerfile with the name **debezium-container-for-oracle.yaml** in the current directory.

   b. Build the container image from the **debezium-container-for-oracle.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-oracle:latest .
```

```
docker build -t debezium-container-for-oracle:latest .
```

The preceding commands build a container image with the name **debezium-container-for-oracle**.

c. Push your custom image to a container registry, such as quay.io or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-oracle:latest
```

```
docker push <myregistry.io>/debezium-container-for-oracle:latest
```

d. Create a new Debezium Oracle KafkaConnect custom resource (CR). For example, create a KafkaConnect CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  image: debezium-container-for-oracle 2
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply the **KafkaConnect** CR to the OpenShift Kafka Connect environment by entering the following command:

```
oc create -f dbz-connect.yaml
```

The command adds a Kafka Connect instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium Oracle connector instance.
You configure a Debezium Oracle connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce events for a subset of the schemas and tables, or it might set properties so that Debezium ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed.

The following example configures a Debezium connector that connects to an Oracle host IP address, on port **1521**. This host has a database named **ORCLCDB**, and **server1** is the server's logical name.

Oracle inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector 1
  labels:
    strimzi.io/cluster: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.oracle.OracleConnector 2
  config:
    database.hostname: <oracle_ip_address> 3
    database.port: 1521 4
    database.user: c##dbzuser 5
    database.password: dbz 6
    database.dbname: ORCLCDB 7
    database.pdb.name : ORCLPDB1, 8
    database.server.name: server1 9
    database.history.kafka.bootstrap.servers: kafka:9092 10
    database.history.kafka.topic: schema-changes.inventory 11
```

Table 6.13. Descriptions of connector configuration settings

Item	Description
1	The name of our connector when we register it with a Kafka Connect service.
2	The name of this Oracle connector class.
3	The address of the Oracle instance.
4	The port number of the Oracle instance.
5	The name of the Oracle user, as specified in Creating users for the connector.
6	The password for the Oracle user, as specified in Creating users for the connector.
7	The name of the database to capture changes from.
8	The name of the Oracle pluggable database that the connector captures changes from. Used in container database (CDB) installations only.
9	Logical name that identifies and provides a namespace for the Oracle database server from which the connector captures changes.

Item	Description
10	The list of Kafka brokers that this connector uses to write and recover DDL statements to the database history topic.
11	The name of the database history topic where the connector writes and recovers DDL statements. This topic is for internal use only and should not be used by consumers.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **inventory-connector.yaml** file, you would run the following command:

```
oc apply -f inventory-connector.yaml
```

The preceding command registers **inventory-connector** and the connector starts to run against the **server1** database as defined in the **KafkaConnector** CR.

For the complete list of the configuration properties that you can set for the Debezium Oracle connector, see Oracle connector properties.

Results

After the connector starts, it performs a consistent snapshot of the Oracle databases that the connector is configured for. The connector then starts generating data change events for row-level operations and streaming the change event records to Kafka topics.

6.5.5. Configuration of container databases and non-container-databases

Oracle Database supports the following deployment types:

Container database (CDB)

A database that can contain multiple pluggable databases (PDBs). Database clients connect to each PDB as if it were a standard, non-CDB database.

Non-container database (non-CDB)

A standard Oracle database, which does not support the creation of pluggable databases.

6.5.6. Verifying that the Debezium Oracle connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-oracle**.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
     a. Enter the following command:

```
oc describe KafkaConnector <connector-name> -n <project>
```

For example,

```
oc describe KafkaConnector inventory-connector-oracle -n debezium
```

The command returns status information that is similar to the following output:

Example 6.3. KafkaConnector resource status

```
Name:         inventory-connector-oracle
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:     RUNNING
```

```
    worker_id:  10.131.1.124:8083
  Name:         inventory-connector-oracle
  Tasks:
    Id:             0
    State:          RUNNING
    worker_id:      10.131.1.124:8083
  Type:             source
 Observed Generation:  1
 Tasks Max:            1
 Topics:
  inventory_connector_oracle
  inventory_connector_oracle.inventory.addresses
  inventory_connector_oracle.inventory.customers
  inventory_connector_oracle.inventory.geom
  inventory_connector_oracle.inventory.orders
  inventory_connector_oracle.inventory.products
  inventory_connector_oracle.inventory.products_on_hand
Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-oracle.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

   - From a terminal window:

     a. Enter the following command:

     ```
     oc get kafkatopics
     ```

     The command returns status information that is similar to the following output:

     Example 6.4. KafkaTopic resource status

     ```
     NAME                                                                CLUSTER
     PARTITIONS   REPLICATION FACTOR   READY
     connect-cluster-configs                                              debezium-
     kafka-cluster   1            1                 True
     connect-cluster-offsets                                              debezium-
     kafka-cluster   25           1                 True
     connect-cluster-status                                               debezium-
     kafka-cluster   5            1                 True
     consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
     debezium-kafka-cluster   50           1                 True
     ```

```
inventory-connector-oracle---a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster   1          1          True
inventory-connector-oracle.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480          debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1          True
inventory-connector-oracle.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5          debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d          debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1          True
inventory-connector-oracle.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1          True
schema-changes.inventory
debezium-kafka-cluster   1          1          True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1          1          True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1          True
```

3. Check topic content.

   - From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

   For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_oracle.inventory.products_on_hand
```

   The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_oracle.inventory.addresses**.

   For each event in the topic, the command returns information that is similar to the following output:

   Example 6.5. Content of a Debezium change event

{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_connector_oracle.inventory.products_on_hand.Key"},"payload":{"product_id":101}} {"schema":
{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector_oracle.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector_oracle.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.oracle.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transaction"}],"optional":false,"name":"inventory_connector_oracle.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.7.Final-redhat-
00001","connector":"oracle","name":"inventory_connector_oracle","ts_ms":1638985247805,
"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,
"gtid":null,"file":"oracle-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"transaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

6.6. DESCRIPTIONS OF DEBEZIUM ORACLE CONNECTOR CONFIGURATION PROPERTIES

The Debezium Oracle connector has numerous configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values. Information about the properties is organized as follows:

- Required Debezium Oracle connector configuration properties

- Database history connector configuration properties that control how Debezium processes events that it reads from the database history topic.
  - Pass-through database history properties
- Pass-through database driver properties that control the behavior of the database driver.

Required Debezium Oracle connector configuration properties

The following configuration properties are *required* unless a default value is available.

Property	Default	Description
name	No default	Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class	No default	The name of the Java class for the connector. Always use a value of **io.debezium.connector.oracle.OracleConnector** for the Oracle connector.

converters	No default	Enumerates a comma-separated list of the symbolic names of the custom converter instances that the connector can use.
		For example, **boolean**.
		This property is required to enable the connector to use a custom converter.
		
		For each converter that you configure for a connector, you must also add a **.type** property, which specifies the fully-qualifed name of the class that implements the converter interface.
		The **.type** property uses the following format:
		
		***<converterSymbolicName>*.type**
		
		For example,
		
		`boolean.type: io.debezium.connector.oracle.converters.NumberOneToBooleanConverter`
		
		If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate any additional configuration parameters with a converter, prefix the parameter names with the symbolic name of the converter.
		
		For example, to define a **selector** parameter that specifies the subset of columns that the **boolean** converter processes, add the following property:
		
		`boolean.selector: .*MYTABLE.FLAG,.*.IS_ARCHIVED`

tasks.max	**1**	The maximum number of tasks to create for this connector. The Oracle connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname	No default	IP address or hostname of the Oracle database server.
database.port	No default	Integer port number of the Oracle database server.
database.user	No default	Name of the Oracle user account that the connector uses to connect to the Oracle database server.

database.password	No default	Password to use when connecting to the Oracle database server.
database.dbname	No default	Name of the database to connect to. Must be the CDB name when working with the CDB + PDB model.
database.url	No default	Specifies the raw database JDBC URL. Use this property to provide flexibility in defining that database connection. Valid values include raw TNS names and RAC connection strings.
database.pdb.name	No default	Name of the Oracle pluggable database to connect to. Use this property with container database (CDB) installations only.
database.server.name	No default	Logical name that identifies and provides a namespace for the Oracle database server from which the connector captures changes. The value that you set is used as a prefix for all Kafka topic names that the connector emits. Specify a logical name that is unique among all connectors in your Debezium environment. The following characters are valid: alphanumeric characters, hyphens, dots, and underscores. **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value. The connector is also unable to recover its database history topic.

database.connection.adapter	**logminer**	The adapter implementation that the connector uses when it streams database changes. You can set the following values: **logminer**(default):: The connector uses the native Oracle LogMiner API.
snapshot.mode	*initial*	Specifies the mode that the connector uses to take snapshots of a captured table. You can set the following values: **initial** The snapshot includes the structure and data of the captured tables. Specify this value to populate topics with a complete representation of the data from the captured tables. **initial_only** The snapshot includes the structure and data of the captured tables. The connector performs an initial snapshot and then stops, without processing any subsequent changes. **schema_only** The snapshot includes only the structure of captured tables. Specify this value if you want the connector to capture data only for changes that occur after the snapshot. **schema_only_recovery** This is a recovery setting for a connector that has already been capturing changes. When you restart the connector, this setting enables recovery of a corrupted or lost database history topic. You might set it periodically to "clean up" a database history topic that has been growing unexpectedly. Database history topics require infinite retention. Note this mode is only safe to be used when it is guaranteed that no schema changes happened since the point in time the connector was shut down before and the point in time the snapshot is taken. After the snapshot is complete, the connector continues to read change events from the database's redo logs except when **snapshot.mode** is configured as **initial_only**. For more information, see the table of **snapshot.mode** options.

snapshot.locking.mode	*shared*	Controls whether and for how long the connector holds a table lock. Table locks prevent certain types of changes table operations from occurring while the connector performs a snapshot. You can set the following values: **shared** Enables concurrent access to the table, but prevents any session from acquiring an exclusive table lock. The connector acquires a **ROW SHARE** level lock while it captures table schema. **none** Prevents the connector from acquiring any table locks during the snapshot. Use this setting only if no schema changes might occur during the creation of the snapshot.
snapshot.include.collection.list	All tables specified in **table.include.list**	An optional, comma-separated list of regular expressions that match the fully-qualified names (***<schemaName>.<tableName>***) of the tables to include in a snapshot. The specified items must be named in the connector's **table.include.list** property. This property takes effect only if the connector's **snapshot.mode** property is set to a value other than **never**. This property does not affect the behavior of incremental snapshots.

snapshot.select.statement.overrides	No default	Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log. The property contains a comma-separated list of fully-qualified table names in the form ***<schemaName>.<tableName>***. For example, **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** For each table in the list, add a further configuration property that specifies the **SELECT** statement for the connector to run on the table when it takes a snapshot. The specified **SELECT** statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this **SELECT** statement property: **snapshot.select.statement.overrides.*<schemaName>.<tableName>*** For example, **snapshot.select.statement.overrides.customers.orders** Example: From a **customers.orders** table that includes the soft-delete column, **delete_flag**, add the following properties if you want a snapshot to include only those records that are not soft-deleted: `"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"` In the resulting snapshot, the connector includes only the records for which **delete_flag = 0**.

schema.include.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **want** to capture changes. Any schema name not included in **schema.include.list** is excluded from having its changes captured. By default, all non-system schemas have their changes captured. Do not also set the **schema.exclude.list** property. In environments that use the LogMiner implementation, you must use POSIX regular expressions only.
include.schema.comments	**false**	Boolean value that specifies whether the connector should parse and publish table and column comments on metadata objects. Enabling this option will bring the implications on memory usage. The number and size of logical schema objects is what largely impacts how much memory is consumed by the Debezium connectors, and adding potentially large string data to each of them can potentially be quite expensive.
schema.exclude.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **do not** want to capture changes. Any schema whose name is not included in **schema.exclude.list** has its changes captured, with the exception of system schemas. Do not also set the **schema.include.list** property. In environments that use the LogMiner implementation, you must use POSIX regular expressions only.
table.include.list	No default	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be monitored. Tables that are not included in the include list are excluded from monitoring. Each table identifier uses the following format: ***<schema_name>.<table_name>*** By default, the connector monitors every non-system table in each monitored database. Do not use this property in combination with **table.exclude.list**. If you use the LogMiner implementation, use only POSIX regular expressions with this property.

table.exclude.list	No default	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be excluded from monitoring. The connector captures change events from any table that is not specified in the exclude list. Specify the identifier for each table using the following format: ***<schemaName>.<tableName>***. Do not use this property in combination with **table.include.list**. If you use the LogMiner implementation, use only POSIX regular expressions with this property.
column.include.list	No default	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that want to include in the change event message values. Fully-qualified names for columns use the following format: ***<Schema_name>.<table_name>.<column_name>*** The primary key column is always included in an event's key, even if you do not use this property to explicitly include its value. If you include this property in the configuration, do not also set the **column.exclude.list** property.
column.exclude.list	No default	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that you want to exclude from change event message values. Fully-qualified column names use the following format: ***<schema_name>.<table_name>.<column_name>*** The primary key column is always included in an event's key, even if you use this property to explicitly exclude its value. If you include this property in the configuration, do not set the **column.include.list** property.

column.mask.hash.*hashAlgorithm*.with.salt.*salt*; **column.mask.hash.v2.*hashAlgorithm*.with.salt.*salt***	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form ***<schemaName>.<tableName>.<columnName>***. In the resulting change event record, the values for the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. In the following example, **CzQMA0cB5K** is a randomly selected salt. `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts. Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting data set might not be completely masked. Hashing strategy version 2 should be used to ensure fidelity if the value is being hashed in different places or systems.
binary.handling.mode	bytes	Specifies how binary (**blob**) columns should be represented in change events, including: **bytes** represents binary data as byte array (default), **base64** represents binary data as base64-encoded String, **hex** represents binary data as hex-encoded (base16) String

schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.
decimal.handling.mode	**precise**	Specifies how the connector should handle floating point values for **NUMBER**, **DECIMAL** and **NUMERIC** columns. You can set one of the following options: **precise** (default) Represents values precisely by using **java.math.BigDecimal** values represented in change events in a binary form. **double** Represents values by using **double** values. Using **double** values is easier, but can result in a loss of precision. **string** Encodes values as formatted strings. Using the **string** option is easier to consume, but results in a loss of semantic information about the real type. For more information, see Numeric types.
interval.handling.mode	**numeric**	Specifies how the connector should handle values for **interval** columns: **numeric** represents intervals using approximate number of microseconds. **string** represents intervals exactly by using the string pattern representation **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**. For example: **P1Y2M3DT4H5M6.78S**.

event.processing.failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events. You can set one of the following options: **fail** Propagates the exception (indicating the offset of the problematic event), causing the connector to stop. **warn** Causes the problematic event to be skipped. The offset of the problematic event is then logged. **skip** Causes the problematic event to be skipped.
max.batch.size	**2048**	A positive integer value that specifies the maximum size of each batch of events to process during each iteration of this connector.
max.queue.size	**8192**	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.
max.queue.size.in.bytes	**0** (disabled)	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.

poll.interval.ms	**1000** (1 second)	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear.
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. The following values are possible: **true** For each delete operation, the connector emits a *delete* event and a subsequent tombstone event. **false** For each delete operation, the connector emits only a *delete* event. After a source record is deleted, a tombstone event (the default behavior) enables Kafka to completely delete all events that share the key of the deleted row in topics that have log compaction enabled.

message.key.columns	No default	A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables. By default, Debezium uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns. To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format: ***<fullyQualifiedTableName>:<keyColumn>,<keyColumn>*** To base a table key on multiple column names, insert commas between the column names. Each fully-qualified table name is a regular expression in the following format: ***<schemaName>.<tableName>*** The property can include entries for multiple tables. Use a semicolon to separate table entries in the list. The following example sets the message key for the tables **inventory.customers** and **purchase.orders**: **inventory.customers:pk1,pk2; (.*).purchaseorders:pk3,pk4** For the table **inventory.customer**, the columns **pk1** and **pk2** are specified as the message key. For the **purchaseorders** tables in any schema, the columns **pk3** and **pk4** server as the message key. There is no limit to the number of columns that you use to create custom message keys. However, it's best to use the minimum number that are required to specify a unique key.

column.truncate.to.*length*.chars	No default	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns to be truncated in change event messages if their length exceeds the specified number of characters. Length is specified as a positive integer. A configuration can include multiple properties that specify different lengths. Specify the fully-qualified name for columns by using the following format: ***<schemaName>.<tableName>.<columnName>***.
column.mask.with.*length*.chars	No default	An optional comma-separated list of regular expressions for masking column names in change event messages by replacing characters with asterisks (*). Specify the number of characters to replace in the name of the property, for example, **column.mask.with.8.chars**. Specify length as a positive integer or zero. Then add regular expressions to the list for each character-based column name where you want to apply a mask. Use the following format to specify fully-qualified column names: ***<schemaName>.<tableName>.<columnName>***. The connector configuration can include multiple properties that specify different lengths.
column.propagate.source.type	No default	An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length**, and **__debezium.source.column.scale** are used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of the form ***<tableName>.<columnName>***, or ***<schemaName>.<tableName>.<columnName>***.

datatype.propagate.source.type	No default	An optional comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** are used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of the form ***<tableName>.<typeName>***, or ***<schemaName>.<tableName>.<typeName>***. See the list of Oracle-specific data type names
heartbeat.interval.ms	**0**	Specifies, in milliseconds, how frequently the connector sends messages to a heartbeat topic. Use this property to determine whether the connector continues to receive change events from the source database. It can also be useful to set the property in situations where no change events occur in captured tables for an extended period. In such a case, although the connector continues to read the redo log, it emits no change event messages, so that the offset in the Kafka topic remains unchanged. Because the connector does not flush the latest system change number (SCN) that it read from the database, the database might retain the redo log files for longer than necessary. If the connector restarts, the extended retention period could result in the connector redundantly sending some change events. The default value of **0** prevents the connector from sending any heartbeat messages.

heartbeat.action.query	No default	Specifies a query that the connector executes on the source database when the connector sends a heartbeat message. For example: **INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')** The connector runs the query after it emits a heartbeat message. Set this property and create a heartbeat table to receive the heartbeat messages to resolve situations in which Debezium fails to synchronize offsets on low-traffic databases that are on the same host as a high-traffic database. After the connector inserts records into the configured table, it is able to receive changes from the low-traffic database and acknowledge SCN changes in the database, so that offsets can be synchronized with the broker.
heartbeat.topics.prefix	**__debezium-heartbeat**	Specifies the string that prefixes the name of the topic to which the connector sends heartbeat messages. The topic is named according to the pattern ***<heartbeat.topics.prefix>.<serverName>***.
snapshot.delay.ms	No default	Specifies an interval in milliseconds that the connector waits after it starts before it takes a snapshot. Use this property to prevent snapshot interruptions when you start multiple connectors in a cluster, which might cause re-balancing of connectors.
snapshot.fetch.size	**2000**	Specifies the maximum number of rows that should be read in one go from each table while taking a snapshot. The connector reads table contents in multiple batches of the specified size.
sanitize.field.names	**true** when the connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Specifies whether field names are normalized to comply with Avro naming requirements. For more information, see Avro naming.

provide.transaction.metadata	**false**	Set the property to **true** if you want Debezium to generate events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name; defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**.
log.mining.strategy	**redo_log_catalog**	Specifies the mining strategy that controls how Oracle LogMiner builds and uses a given data dictionary for resolving table and column ids to names. **redo_log_catalog**:: Writes the data dictionary to the online redo logs causing more archive logs to be generated over time. This also enables tracking DDL changes against captured tables, so if the schema changes frequently this is the ideal choice. **online_catalog**:: Uses the database's current data dictionary to resolve object ids and does not write any extra information to the online redo logs. This allows LogMiner to mine substantially faster but at the expense that DDL changes cannot be tracked. If the captured table(s) schema changes infrequently or never, this is the ideal choice.
log.mining.buffer.type	**memory**	The buffer type controls how the connector manages buffering transaction data. **memory** - Uses the JVM process' heap to buffer all transaction data. Choose this option if you don't expect the connector to process a high number of long-running or large transactions. When this option is active, the buffer state is not persisted across restarts. Following a restart, recreate the buffer from the SCN value of the current offset.

log.mining.session.max.xs	0	The maximum number of milliseconds that a LogMiner session can be active before a new session is used. For low volume systems, a LogMiner session may consume too much PGA memory when the same session is used for a long period of time. The default behavior is to only use a new LogMiner session when a log switch is detected. By setting this value to something greater than **0**, this specifies the maximum number of milliseconds a LogMiner session can be active before it gets stopped and started to deallocate and reallocate PGA memory.
log.mining.batch.size.min	1000	The minimum SCN interval size that this connector attempts to read from redo/archive logs. Active batch size is also increased/decreased by this amount for tuning connector throughput when needed.
log.mining.batch.size.max	100000	The maximum SCN interval size that this connector uses when reading from redo/archive logs.
log.mining.batch.size.default	20000	The starting SCN interval size that the connector uses for reading data from redo/archive logs.
log.mining.sleep.time.min.ms	0	The minimum amount of time that the connector sleeps after reading data from redo/archive logs and before starting reading data again. Value is in milliseconds.
log.mining.sleep.time.max.ms	3000	The maximum amount of time that the connector ill sleeps after reading data from redo/archive logs and before starting reading data again. Value is in milliseconds.
log.mining.sleep.time.default.ms	1000	The starting amount of time that the connector sleeps after reading data from redo/archive logs and before starting reading data again. Value is in milliseconds.
log.mining.sleep.time.increment.ms	200	The maximum amount of time up or down that the connector uses to tune the optimal sleep time when reading data from logminer. Value is in milliseconds.
log.mining.view.fetch.size	10000	The number of content records that the connector fetches from the LogMiner content view.

log.mining.archive.log.hours	**0**	The number of hours in the past from SYSDATE to mine archive logs. When the default setting (**0**) is used, the connector mines all archive logs.
log.mining.archive.log.only.mode	**false**	Controls whether or not the connector mines changes from just archive logs or a combination of the online redo logs and archive logs (the default). Redo logs use a circular buffer that can be archived at any point. In environments where online redo logs are archived frequently, this can lead to LogMiner session failures. In contrast to redo logs, archive logs are guaranteed to be reliable. Set this option to **true** to force the connector to mine archive logs only. After you set the connector to mine only the archive logs, the latency between an operation being committed and the connector emitting an associated change event might increase. The degree of latency depends on how frequently the database is configured to archive online redo logs.
log.mining.archive.log.only.scn.poll.interval.ms	**10000**	The number of milliseconds the connector will sleep in between polling to determine if the starting system change number is in the archive logs. If **log.mining.archive.log.only.mode** is not enabled, this setting is not used.
log.mining.transaction.retention.hours	**0**	Positive integer value that specifies the number of hours to retain long running transactions between redo log switches. When set to **0**, transactions are retained until a commit or rollback is detected. The LogMiner adapter maintains an in-memory buffer of all running transactions. Because all of the DML operations that are part of a transaction are buffered until a commit or rollback is detected, long-running transactions should be avoided in order to not overflow that buffer. Any transaction that exceeds this configured value is discarded entirely, and the connector does not emit any messages for the operations that were part of the transaction.

log.mining.archive.destination.name	No default	Specifies the configured Oracle archive destination to use when mining archive logs with LogMiner. The default behavior automatically selects the first valid, local configured destination. However, you can use a specific destination can be used by providing the destination name, for example, **LOG_ARCHIVE_DEST_5**.
log.mining.username.exclude.list	No default	List of database users to exclude from the LogMiner query. It can be useful to set this property if you want the capturing process to always exclude the changes that specific users make.
log.mining.scn.gap.detection.gap.size.min	**1000000**	Specifies a value that the connector compares to the difference between the current and previous SCN values to determine whether an SCN gap exists. If the difference between the SCN values is greater than the specified value, and the time difference is smaller than **log.mining.scn.gap.detection.time.interval.max.ms** then an SCN gap is detected, and the connector uses a mining window larger than the configured maximum batch.
log.mining.scn.gap.detection.time.interval.max.ms	**20000**	Specifies a value, in milliseconds, that the connector compares to the difference between the current and previous SCN timestamps to determine whether an SCN gap exists. If the difference between the timestamps is less than the specified value, and the SCN delta is greater than **log.mining.scn.gap.detection.gap.size.min**, then an SCN gap is detected and the connector uses a mining window larger than the configured maximum batch.

lob.enabled	**false**	Controls whether or not large object (CLOB or BLOB) column values are emitted in change events. By default, change events have large object columns, but the columns contain no values. There is a certain amount of overhead in processing and managing large object column types and payloads. To capture large object values and serialized them in change events, set this option to **true**. **NOTE** Use of large object data types is a Technology Preview feature.
unavailable.value.placeholder	**__debezium_unavailable_value**	Specifies the constant that the connector provides to indicate that the original value is unchanged and not provided by the database.

rac.nodes	No default	A comma-separated list of Oracle Real Application Clusters (RAC) node host names or addresses. This field is required to enable compatibility with an Oracle RAC deployment.
		
		Specify the list of RAC nodes by using one of the following methods:
		
		• Specify a value for **database.port**, and use the specified port value for each address in the **rac.nodes** list. For example:
		
		`database.port=1521`
		`rac.nodes=192.168.1.100,192.168.1.101`
		
		• Specify a value for **database.port**, and override the default port for one or more entries in the list. The list can include entries that use the default **database.port** value, and entries that define their own unique port values. For example:
		
		`database.port=1521`
		`rac.nodes=192.168.1.100,192.168.1.101:1522`
		
		If you supply a raw JDBC URL for the database by using the **database.url** property, instead of defining a value for **database.port**, each RAC node entry must explicitly specify a port value.

skipped.operations	No default	A comma-separated list of the operation types that you want the connector to skip during streaming. You can configure the connector to skip the following types of operations: • **c** (insert/create) • **u** (update) • **d** (delete) • **t** (truncate) By default, no operations are skipped.
signal.data.collection	No default value	Fully-qualified name of the data collection that is used to send signals to the connector. When you use this property with an Oracle pluggable database (PDB), set its value to the name of the root database. Use the following format to specify the collection name: ***<databaseName>.<schemaName>.<tableName>***
incremental.snapshot.chunk.size	**1024**	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.

Debezium Oracle connector database history configuration properties

Debezium provides a set of **database.history.*** properties that control how the connector interacts with the schema history topic.

The following table describes the **database.history** properties for configuring the Debezium connector.

Table 6.14. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic	No default	The full name of the Kafka topic where the connector stores the database schema history.

Property	Default	Description
database.history.kafka.bootstrap.servers	No default	A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. This connection is used for retrieving the database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
database.history.kafka.recovery.poll.interval.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait during startup/recovery while polling for persisted data. The default is 100ms.
database.history.kafka.query.timeout.ms	**3000**	An integer value that specifies the maximum number of milliseconds the connector should wait while fetching cluster information using Kafka admin client.
database.history.kafka.recovery.attempts	**4**	The maximum number of times that the connector should try to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is **recovery.attempts** x **recovery.poll.interval.ms**.
database.history.skip.unparseable.ddl	**false**	A Boolean value that specifies whether the connector should ignore malformed or unknown database statements or stop processing so a human can fix the issue. The safe default is **false**. Skipping should be used only with care as it can lead to data loss or mangling when the binlog is being processed.
database.history.store.only.monitored.tables.ddl *Deprecated and scheduled for removal in a future release; use **database.history.store.only.captured.tables.ddl** instead.*	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Property	Default	Description
database.history.store.only.captured.tables.ddl	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Pass-through database history properties for configuring producer and consumer clients

Debezium relies on a Kafka producer to write schema changes to database history topics. Similarly, it relies on a Kafka consumer to read from database history topics when a connector starts. You define the configuration for the Kafka producer and consumer clients by assigning values to a set of pass-through configuration properties that begin with the **database.history.producer.*** and **database.history.consumer.*** prefixes. The pass-through producer and consumer database history properties control a range of behaviors, such as how these clients secure connections with the Kafka broker, as shown in the following example:

```
database.history.producer.security.protocol=SSL
database.history.producer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.producer.ssl.keystore.password=test1234
database.history.producer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.producer.ssl.truststore.password=test1234
database.history.producer.ssl.key.password=test1234

database.history.consumer.security.protocol=SSL
database.history.consumer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.consumer.ssl.keystore.password=test1234
database.history.consumer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.consumer.ssl.truststore.password=test1234
database.history.consumer.ssl.key.password=test1234
```

Debezium strips the prefix from the property name before it passes the property to the Kafka client.

See the Kafka documentation for more details about Kafka producer configuration properties and Kafka consumer configuration properties.

Debezium Oracle connector pass-through database driver configuration properties

The Debezium connector provides for pass-through configuration of the database driver. Pass-through database properties begin with the prefix **database.***. For example, the connector passes properties such as **database.foobar=false** to the JDBC URL.

As is the case with the pass-through properties for database history clients, Debezium strips the prefixes from the properties before it passes them to the database driver.

6.7. MONITORING DEBEZIUM ORACLE CONNECTOR PERFORMANCE

The Debezium Oracle connector provides three metric types in addition to the built-in support for JMX metrics that Apache Zookeeper, Apache Kafka, and Kafka Connect have.

- snapshot metrics; for monitoring the connector when performing snapshots
- streaming metrics; for monitoring the connector when processing change events
- schema history metrics; for monitoring the status of the connector's schema history

Please refer to the monitoring documentation for details of how to expose these metrics via JMX.

6.7.1. Debezium Oracle connector snapshot metrics

The **MBean** is **debezium.oracle:type=connector-metrics,context=snapshot,server=*<oracle.server.name>***. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.

Attributes	Type	Description
QueueRemainingCapacity	int	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	int	The total number of tables that are being included in the snapshot.
RemainingTableCount	int	The number of tables that the snapshot has yet to copy.
SnapshotRunning	boolean	Whether the snapshot was started.
SnapshotAborted	boolean	Whether the snapshot was aborted.
SnapshotCompleted	boolean	Whether the snapshot completed.
SnapshotDurationInSeconds	long	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	Map<String, Long>	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The connector also provides the following additional snapshot metrics when an incremental snapshot is executed:

Attributes	Type	Description
ChunkId	string	The identifier of the current snapshot chunk.
ChunkFrom	string	The lower bound of the primary key set defining the current chunk.
ChunkTo	string	The upper bound of the primary key set defining the current chunk.
TableFrom	string	The lower bound of the primary key set of the currently snapshotted table.
TableTo	string	The upper bound of the primary key set of the currently snapshotted table.

6.7.2. Debezium Oracle connector streaming metrics

The **MBean** is **debezium.oracle:type=connector-metrics,context=streaming,server=*<oracle.server.name>***. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	string	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	long	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	long	The total number of events that this connector has seen since the last start or metrics reset.
TotalNumberOfCreateEventsSeen	long	The total number of create events that this connector has seen since the last start or metrics reset.

Attributes	Type	Description
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.

Attributes	Type	Description
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

The Debezium Oracle connector also provides the following additional streaming metrics:

Table 6.15. Descriptions of additional streaming metrics

Attributes	Type	Description
CurrentScn	**string**	The most recent system change number that has been processed.
OldestScn	**string**	The oldest system change number in the transaction buffer.
CommittedScn	**string**	The last committed system change number from the transaction buffer.
OffsetScn	**string**	The system change number currently written to the connector's offsets.
CurrentRedoLogFileName	**string[]**	Array of the log files that are currently mined.
MinimumMinedLogCount	**long**	The minimum number of logs specified for any LogMiner session.

Attributes	Type	Description
MaximumMinedLogCount	long	The maximum number of logs specified for any LogMiner session.
RedoLogStatus	string[]	Array of the current state for each mined logfile with the format ***filename|status***.
SwitchCounter	int	The number of times the database has performed a log switch for the last day.
LastCapturedDmlCount	long	The number of DML operations observed in the last LogMiner session query.
MaxCapturedDmlInBatch	long	The maximum number of DML operations observed while processing a single LogMiner session query.
TotalCapturedDmlCount	long	The total number of DML operations observed.
FetchingQueryCount	long	The total number of LogMiner session query (aka batches) performed.
LastDurationOfFetchQueryInMilliseconds	long	The duration of the last LogMiner session query's fetch in milliseconds.
MaxDurationOfFetchQueryInMilliseconds	long	The maximum duration of any LogMiner session query's fetch in milliseconds.
LastBatchProcessingTimeInMilliseconds	long	The duration for processing the last LogMiner query batch results in milliseconds.
TotalParseTimeInMilliseconds	long	The time in milliseconds spent parsing DML event SQL statements.
LastMiningSessionStartTimeInMilliseconds	long	The duration in milliseconds to start the last LogMiner session.

Attributes	Type	Description
MaxMiningSessionStartTimeInMilliseconds	long	The longest duration in milliseconds to start a LogMiner session.
TotalMiningSessionStartTimeInMilliseconds	long	The total duration in milliseconds spent by the connector starting LogMiner sessions.
MinBatchProcessingTimeInMilliseconds	long	The minimum duration in milliseconds spent processing results from a single LogMiner session.
MaxBatchProcessingTimeInMilliseconds	long	The maximum duration in milliseconds spent processing results from a single LogMiner session.
TotalProcessingTimeInMilliseconds	long	The total duration in milliseconds spent processing results from LogMiner sessions.
TotalResultSetNextTimeInMilliseconds	long	The total duration in milliseconds spent by the JDBC driver fetching the next row to be processed from the log mining view.
TotalProcessedRows	long	The total number of rows processed from the log mining view across all sessions.
BatchSize	int	The number of entries fetched by the log mining query per database round-trip.
MillisecondToSleepBetweenMiningQuery	long	The number of milliseconds the connector sleeps before fetching another batch of results from the log mining view.
MaxBatchProcessingThroughput	long	The maximum number of rows/second processed from the log mining view.

Attributes	Type	Description
AverageBatchProcessingThroughput	long	The average number of rows/second processed from the log mining.
LastBatchProcessingThroughput	long	The average number of rows/second processed from the log mining view for the last batch.
NetworkConnectionProblemsCounter	long	The number of connection problems detected.
HoursToKeepTransactionInBuffer	int	The number of hours that transactions are retained by the connector's in-memory buffer without being committed or rolled back before being discarded. See log.mining.transaction.retention for more details.
NumberOfActiveTransactions	long	The number of current active transactions in the transaction buffer.
NumberOfCommittedTransactions	long	The number of committed transactions in the transaction buffer.
NumberOfRolledBackTransactions	long	The number of rolled back transactions in the transaction buffer.
CommitThroughput	long	The average number of committed transactions per second in the transaction buffer.
RegisteredDmlCount	long	The number of registered DML operations in the transaction buffer.
LagFromSourceInMilliseconds	long	The time difference in milliseconds between when a change occurred in the transaction logs and when its added to the transaction buffer.

Attributes	Type	Description
MaxLagFromSourceInMilliseconds	long	The maximum time difference in milliseconds between when a change occurred in the transaction logs and when its added to the transaction buffer.
MinLagFromSourceInMilliseconds	long	The minimum time difference in milliseconds between when a change occurred in the transaction logs and when its added to the transaction buffer.
AbandonedTransactionIds	string[]	An array of the most recent abandoned transaction identifiers removed from the transaction buffer due to their age. See log.mining.transaction.retention.hours for details.
RolledBackTransactionIds	string[]	An array of the most recent transaction identifiers that have been mined and rolled back in the transaction buffer.
LastCommitDurationInMilliseconds	long	The duration of the last transaction buffer commit operation in milliseconds.
MaxCommitDurationInMilliseconds	long	The duration of the longest transaction buffer commit operation in milliseconds.
ErrorCount	int	The number of errors detected.
WarningCount	int	The number of warnings detected.

Attributes	Type	Description
ScnFreezeCount	int	The number of times that the system change number was checked for advancement and remains unchanged. A high value can indicate that a long-running transactions is ongoing and is preventing the connector from flushing the most recently processed system change number to the connector's offsets. When conditions are optimal, the value should be close to or equal to **0**.
UnparsableDdlCount	int	The number of DDL records that have been detected but could not be parsed by the DDL parser. This should always be **0**; however when allowing unparsable DDL to be skipped, this metric can be used to determine if any warnings have been written to the connector logs.
MiningSessionUserGlobalAreaMemoryInBytes	long	The current mining session's user global area (UGA) memory consumption in bytes.
MiningSessionUserGlobalAreaMaxMemoryInBytes	long	The maximum mining session's user global area (UGA) memory consumption in bytes across all mining sessions.
MiningSessionProcessGlobalAreaMemoryInBytes	long	The current mining session's process global area (PGA) memory consumption in bytes.
MiningSessionProcessGlobalAreaMaxMemoryInBytes	long	The maximum mining session's process global area (PGA) memory consumption in bytes across all mining sessions.

6.7.3. Debezium Oracle connector schema history metrics

The MBean is **debezium.oracle:type=connector-metrics,context=schema-history,server=*<oracle.server.name>***.

The following table lists the schema history metrics that are available.

Attributes	Type	Description
Status	**string**	One of **STOPPED**, **RECOVERING** (recovering history from the storage), **RUNNING** describing the state of the database history.
RecoveryStartTime	**long**	The time in epoch seconds at what recovery has started.
ChangesRecovered	**long**	The number of changes that were read during recovery phase.
ChangesApplied	**long**	the total number of schema changes applied during recovery and runtime.
MilliSecondsSinceLast RecoveredChange	**long**	The number of milliseconds that elapsed since the last change was recovered from the history store.
MilliSecondsSinceLastAppliedChange	**long**	The number of milliseconds that elapsed since the last change was applied.
LastRecoveredChange	**string**	The string representation of the last change recovered from the history store.
LastAppliedChange	**string**	The string representation of the last applied change.

6.8. HOW DEBEZIUM ORACLE CONNECTORS HANDLE FAULTS AND PROBLEMS

Debezium is a distributed system that captures all changes in multiple upstream databases; it never misses or loses an event. When the system is operating normally or being managed carefully then Debezium provides *exactly once* delivery of every change event record.

If a fault occurs, Debezium does not lose any events. However, while it is recovering from the fault, it might repeat some change events. In these abnormal situations, Debezium, like Kafka, provides *at least once* delivery of change events.

The rest of this section describes how Debezium handles various kinds of faults and problems.

Logs do not contain offset, perform a new snapshot

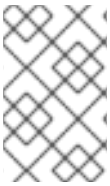
In some cases, after the Debezium Oracle connector restarts, it reports the following error:

```
Online REDO LOG files or archive logs do not contain the offset scn xxxxxxx. Please perform a new snapshot.
```

After the connector examines the redo and archive logs, if it cannot find the SCN that is recorded in the connector offsets, it returns the preceding error. Because the connector uses the SCN to determine where to resume processing, if the expected SCN if not found, a new snapshot must be completed.

You might find that the **V$ARCHIVED_LOG** table contains a record with an SCN that matches the expected range. However, the record might not be available for mining. To be available for mining, a record must include a filename in the **NAME** column, a value of **NO** in the **DELETED** column, and a value of **A** (available) in the **STATUS** column. If a record does not match any of these criteria, it is considered incomplete and cannot be mined.

At a minimum, archive logs must be retained for as long as the longest downtime window of the connector.



NOTE

Records that have no value in the **NAME** column no longer exist in the file system. In such records, the value of the **DELETED** field is set to **YES**, and the **STATUS** field is set to **D** to indicate that the log is deleted.

ORA-25191 - Cannot reference overflow table of an index-organized table

Oracle might issue this error during the snapshot phase when encountering an index-organized table (IOT). This error means that the connector has attempted to execute an operation that must be executed against the parent index-organized table that contains the specified overflow table.

To resolve this, the IOT name used in the SQL operation should be replaced with the parent index-organized table name. To determine the parent index-organized table name, use the following SQL:

```
SELECT IOT_NAME
  FROM DBA_TABLES
 WHERE OWNER='<tablespace-owner>'
   AND TABLE_NAME='<iot-table-name-that-failed>'
```

The connector's **table.include.list** or **table.exclude.list** configuration options should then be adjusted to explicitly include or exclude the appropriate tables to avoid the connector from attempting to capture changes from the child index-organized table.

ORA-04036: PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT

Oracle might report this error when Debezium connects to a database in which changes occur infrequently. The Debezium connector starts an Oracle LogMiner session and reuses this session until a log switch is detected. The reuse is both a performance and resource utilization optimization; however, a long-running mining session can cause high Program Global Area (PGA) memory usage.

If your redo log switches infrequently, you can avoid the ORA-04036 error by specifying how frequently Oracle switches logs. A log switch causes the connector to restart the mining session, thereby avoiding high PGA memory usage. The following configuration forces Oracle to switch log files every 20 minutes

if a log switch does not occur during that interval:

```
ALTER SYSTEM SET archive_lag_target=1200 scope=both;
```

Running the preceding query requires specific administrative privileges. Coordinate with your database administrator to implement the change.

If you are unable to adjust the Oracle database parameter, you can instead use the connector configuration option **log.mining.session.max.ms** to control the duration of an Oracle LogMiner session so that it is restarted regularly even if a database log switch has not yet happened.

LogMiner adapter does not capture changes made by SYS or SYSTEM

Oracle uses the **SYS** and **SYSTEM** accounts to carry out many internal changes. When the Debezium Oracle connector fetches changes from LogMiner, it automatically filters changes that originate from these administrator accounts. To ensure that the connector emit event records when you change a table, never use the **SYS** or **SYSTEM** user accounts to modify the table.

Connector stops capturing changes from Oracle on AWS

Due to the fixed idle timeout of 350 seconds on the AWS Gateway Load Balancer , JDBC calls that require more than 350 seconds to complete can hang indefinitely.

In situations where calls to the Oracle LogMiner API take more than 350 seconds to complete, a timeout can be triggered, causing the AWS Gateway Load Balancer to hang. For example, such timeouts can occur when a LogMiner session that processes large amounts of data runs concurrently with Oracle's periodic checkpointing task.

To prevent timeouts from occurring on the AWS Gateway Load Balancer, enable keep-alive packets from the Kafka Connect environment, by performing the following steps as root or a super-user:

1. From a terminal, run the following command:

```
sysctl -w net.ipv4.tcp_keepalive_time=60
```

2. Edit **/etc/sysctl.conf** and set the value of the following variable as shown:

```
net.ipv4.tcp_keepalive_time=60
```

3. Reconfigure the Debezium for Oracle connector to use the **database.url** property rather than **database.hostname** and add the **(ENABLE=broken)** Oracle connect string descriptor as shown in the following example:

```
database.url=jdbc:oracle:thin:username/password!@(DESCRIPTION=(ENABLE=broken)
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(Host=hostname)(Port=port)))
(CONNECT_DATA=(SERVICE_NAME=serviceName)))
```

The preceding steps configure the TCP network stack to send keep-alive packets every 60 seconds. As a result, the AWS Gateway Load Balancer does not timeout when JDBC calls to the LogMiner API take more than 350 seconds to complete, enabling the connector to continue to read changes from the database's transaction logs.

CHAPTER 7. DEBEZIUM CONNECTOR FOR POSTGRESQL

The Debezium PostgreSQL connector captures row-level changes in the schemas of a PostgreSQL database. For information about the PostgreSQL versions that are compatible with the connector, see the Debezium Supported Configurations page.

The first time it connects to a PostgreSQL server or cluster, the connector takes a consistent snapshot of all schemas. After that snapshot is complete, the connector continuously captures row-level changes that insert, update, and delete database content and that were committed to a PostgreSQL database. The connector generates data change event records and streams them to Kafka topics. For each table, the default behavior is that the connector streams all generated events to a separate Kafka topic for that table. Applications and services consume data change event records from that topic.

Information and procedures for using a Debezium PostgreSQL connector is organized as follows:

- Section 7.1, "Overview of Debezium PostgreSQL connector"
- Section 7.2, "How Debezium PostgreSQL connectors work"
- Section 7.3, "Descriptions of Debezium PostgreSQL connector data change events"
- Section 7.4, "How Debezium PostgreSQL connectors map data types"
- Section 7.5, "Setting up PostgreSQL to run a Debezium connector"
- Section 7.6, "Deployment of Debezium PostgreSQL connectors"
- Section 7.7, "Monitoring Debezium PostgreSQL connector performance"
- Section 7.8, "How Debezium PostgreSQL connectors handle faults and problems"

7.1. OVERVIEW OF DEBEZIUM POSTGRESQL CONNECTOR

PostgreSQL's *logical decoding* feature was introduced in version 9.4. It is a mechanism that allows the extraction of the changes that were committed to the transaction log and the processing of these changes in a user-friendly manner with the help of an *output plug-in*. The output plug-in enables clients to consume the changes.

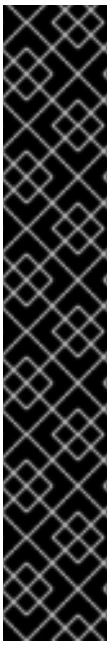
The PostgreSQL connector contains two main parts that work together to read and process database changes:

- **pgoutput** is the standard logical decoding output plug-in in PostgreSQL 10+. This is the only supported logical decoding output plug-in in this Debezium release. This plug-in is maintained by the PostgreSQL community, and used by PostgreSQL itself for logical replication. This plug-in is always present so no additional libraries need to be installed. The Debezium connector interprets the raw replication event stream directly into change events.
- Java code (the actual Kafka Connect connector) that reads the changes produced by the logical decoding output plug-in by using PostgreSQL's *streaming replication protocol* and the PostgreSQL *JDBC driver*.

The connector produces a *change event* for every row-level insert, update, and delete operation that was captured and sends change event records for each table in a separate Kafka topic. Client applications read the Kafka topics that correspond to the database tables of interest, and can react to every row-level event they receive from those topics.

PostgreSQL normally purges write-ahead log (WAL) segments after some period of time. This means that the connector does not have the complete history of all changes that have been made to the database. Therefore, when the PostgreSQL connector first connects to a particular PostgreSQL database, it starts by performing a *consistent snapshot* of each of the database schemas. After the connector completes the snapshot, it continues streaming changes from the exact point at which the snapshot was made. This way, the connector starts with a consistent view of all of the data, and does not omit any changes that were made while the snapshot was being taken.

The connector is tolerant of failures. As the connector reads changes and produces events, it records the WAL position for each event. If the connector stops for any reason (including communication failures, network problems, or crashes), upon restart the connector continues reading the WAL where it last left off. This includes snapshots. If the connector stops during a snapshot, the connector begins a new snapshot when it restarts.



IMPORTANT

The connector relies on and reflects the PostgreSQL logical decoding feature, which has the following limitations:

- Logical decoding does not support DDL changes. This means that the connector is unable to report DDL change events back to consumers.
- Logical decoding replication slots are supported on only **primary** servers. When there is a cluster of PostgreSQL servers, the connector can run on only the active **primary** server. It cannot run on **hot** or **warm** standby replicas. If the **primary** server fails or is demoted, the connector stops. After the **primary** server has recovered, you can restart the connector. If a different PostgreSQL server has been promoted to **primary**, adjust the connector configuration before restarting the connector.

Behavior when things go wrong describes what the connector does when there is a problem.

IMPORTANT

Debezium currently supports databases with UTF-8 character encoding only. With a single byte character encoding, it is not possible to correctly process strings that contain extended ASCII code characters.

7.2. HOW DEBEZIUM POSTGRESQL CONNECTORS WORK

To optimally configure and run a Debezium PostgreSQL connector, it is helpful to understand how the connector performs snapshots, streams change events, determines Kafka topic names, and uses metadata.

Details are in the following topics:

- Section 7.2.2, “How Debezium PostgreSQL connectors perform database snapshots”
- Section 7.2.3, “How Debezium PostgreSQL connectors stream change event records”
- Section 7.2.4, “Default names of Kafka topics that receive Debezium PostgreSQL change event records”
- Section 7.2.5, “Metadata in Debezium PostgreSQL change event records”

- Section 7.2.6, "Debezium PostgreSQL connector-generated events that represent transaction boundaries"

7.2.1. Security for PostgreSQL connector

To use the Debezium connector to stream changes from a PostgreSQL database, the connector must operate with specific privileges in the database. Although one way to grant the necessary privileges is to provide the user with **superuser** privileges, doing so potentially exposes your PostgreSQL data to unauthorized access. Rather than granting excessive privileges to the Debezium user, it is best to create a dedicated Debezium replication user to which you grant specific privileges.

For more information about configuring privileges for the Debezium PostgreSQL user, see Setting up permissions. For more information about PostgreSQL logical replication security, see the PostgreSQL documentation.

7.2.2. How Debezium PostgreSQL connectors perform database snapshots

Most PostgreSQL servers are configured to not retain the complete history of the database in the WAL segments. This means that the PostgreSQL connector would be unable to see the entire history of the database by reading only the WAL. Consequently, the first time that the connector starts, it performs an initial *consistent snapshot* of the database. The default behavior for performing a snapshot consists of the following steps. You can change this behavior by setting the **snapshot.mode** connector configuration property to a value other than **initial**.

1. Start a transaction with a SERIALIZABLE, READ ONLY, DEFERRABLE isolation level to ensure that subsequent reads in this transaction are against a single consistent version of the data. Any changes to the data due to subsequent **INSERT**, **UPDATE**, and **DELETE** operations by other clients are not visible to this transaction.

2. Read the current position in the server's transaction log.

3. Scan the database tables and schemas, generate a **READ** event for each row and write that event to the appropriate table-specific Kafka topic.

4. Commit the transaction.

5. Record the successful completion of the snapshot in the connector offsets.

If the connector fails, is rebalanced, or stops after Step 1 begins but before Step 5 completes, upon restart the connector begins a new snapshot. After the connector completes its initial snapshot, the PostgreSQL connector continues streaming from the position that it read in Step 2. This ensures that the connector does not miss any updates. If the connector stops again for any reason, upon restart, the connector continues streaming changes from where it previously left off.

Table 7.1. Options for the snapshot.mode connector configuration property

Option	Description

Option	Description
always	The connector always performs a snapshot when it starts. After the snapshot completes, the connector continues streaming changes from step 3 in the above sequence. This mode is useful in these situations: • It is known that some WAL segments have been deleted and are no longer available. • After a cluster failure, a new primary has been promoted. The **always** snapshot mode ensures that the connector does not miss any changes that were made after the new primary had been promoted but before the connector was restarted on the new primary.
never	The connector never performs snapshots. When a connector is configured this way, its behavior when it starts is as follows. If there is a previously stored LSN in the Kafka offsets topic, the connector continues streaming changes from that position. If no LSN has been stored, the connector starts streaming changes from the point in time when the PostgreSQL logical replication slot was created on the server. The **never** snapshot mode is useful only when you know all data of interest is still reflected in the WAL.
initial_only	The connector performs a database snapshot and stops before streaming any change event records. If the connector had started but did not complete a snapshot before stopping, the connector restarts the snapshot process and stops when the snapshot completes.
exported	Deprecated, all modes are lockless.

7.2.2.1. Ad hoc snapshots

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing table data, Debezium includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of tables.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling tables. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the tables to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the tables in the database.

You specify the tables to capture by sending an **execute-snapshot** message to the signaling table. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the tables to include in the snapshot, as described in the following table:

Table 7.2. Example of an ad hoc execute-snapshot signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.
data-collections	*N/A*	An array that contains the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling table. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each table. Based on the number of entries in the table, and the configured chunk size, Debezium divides the table into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

7.2.2.2. Incremental snapshots

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each table in phases, in a series of configurable chunks. You can specify the tables that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each table row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the

other.

- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the table from the beginning.
- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a table to its **table.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each table by primary key and then splits the table into chunks based on the configured chunk size. Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying table records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a table row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same table row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified table chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the table's Kafka topic.

The connector repeats the process for each snapshot chunk.

Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling table on the source database. You submit signals to the table as SQL **INSERT** queries. After Debezium detects the change in the signaling table, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the tables to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.

NOTE

If the name of a table that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the table to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling table.

The following table describes the these parameters:

Table 7.3. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling table on the source database
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of table names to include in the snapshot. The array lists tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

WARNING

The Debezium connector for PostgreSQL does not support schema changes while an incremental snapshot is running. If a schema change is performed *before* the incremental snapshot start but *after* sending the signal then passthrough config option **database.autosave** is set to **conservative** to correctly process the schema change.

7.2.3. How Debezium PostgreSQL connectors stream change event records

The PostgreSQL connector typically spends the vast majority of its time streaming changes from the PostgreSQL server to which it is connected. This mechanism relies on *PostgreSQL's replication protocol*. This protocol enables clients to receive changes from the server as they are committed in the server's transaction log at certain positions, which are referred to as Log Sequence Numbers (LSNs).

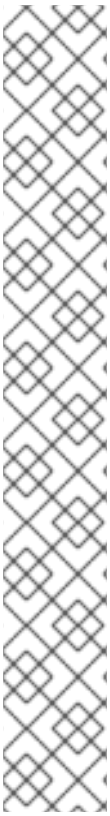
Whenever the server commits a transaction, a separate server process invokes a callback function from the logical decoding plug-in. This function processes the changes from the transaction, converts them to a specific format (Protobuf or JSON in the case of Debezium plug-in) and writes them on an output stream, which can then be consumed by clients.

The Debezium PostgreSQL connector acts as a PostgreSQL client. When the connector receives changes it transforms the events into Debezium *create*, *update*, or *delete* events that include the LSN of the event. The PostgreSQL connector forwards these change events in records to the Kafka Connect framework, which is running in the same process. The Kafka Connect process asynchronously writes the change event records in the same order in which they were generated to the appropriate Kafka topic.

Periodically, Kafka Connect records the most recent *offset* in another Kafka topic. The offset indicates source-specific position information that Debezium includes with each event. For the PostgreSQL connector, the LSN recorded in each change event is the offset.

When Kafka Connect gracefully shuts down, it stops the connectors, flushes all event records to Kafka, and records the last offset received from each connector. When Kafka Connect restarts, it reads the last recorded offset for each connector, and starts each connector at its last recorded offset. When the

connector restarts, it sends a request to the PostgreSQL server to send the events starting just after that position.



NOTE

The PostgreSQL connector retrieves schema information as part of the events sent by the logical decoding plug-in. However, the connector does not retrieve information about which columns compose the primary key. The connector obtains this information from the JDBC metadata (side channel). If the primary key definition of a table changes (by adding, removing or renaming primary key columns), there is a tiny period of time when the primary key information from JDBC is not synchronized with the change event that the logical decoding plug-in generates. During this tiny period, a message could be created with an inconsistent key structure. To prevent this inconsistency, update primary key structures as follows:

1. Put the database or an application into a read-only mode.
2. Let Debezium process all remaining events.
3. Stop Debezium.
4. Update the primary key definition in the relevant table.
5. Put the database or the application into read/write mode.
6. Restart Debezium.

PostgreSQL 10+ logical decoding support (pgoutput)

As of PostgreSQL 10+, there is a logical replication stream mode, called **pgoutput** that is natively supported by PostgreSQL. This means that a Debezium PostgreSQL connector can consume that replication stream without the need for additional plug-ins. This is particularly valuable for environments where installation of plug-ins is not supported or not allowed.

See Setting up PostgreSQL for more details.

7.2.4. Default names of Kafka topics that receive Debezium PostgreSQL change event records

By default, the PostgreSQL connector writes change events for all **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table. The connector uses the following convention to name change event topics:

serverName.schemaName.tableName

The following list provides definitions for the components of the default name:

serverName

The logical name of the connector, as specified by the **database.server.name** configuration property.

schemaName

The name of the database schema in which the change event occurred.

tableName

The name of the database table in which the change event occurred.

For example, suppose that **fulfillment** is the logical server name in the configuration for a connector that is capturing changes in a PostgreSQL installation that has a **postgres** database and an **inventory** schema that contains four tables: **products**, **products_on_hand**, **customers**, and **orders**. The connector would stream records to these four Kafka topics:

- **fulfillment.inventory.products**
- **fulfillment.inventory.products_on_hand**
- **fulfillment.inventory.customers**
- **fulfillment.inventory.orders**

Now suppose that the tables are not part of a specific schema but were created in the default **public** PostgreSQL schema. The names of the Kafka topics would be:

- **fulfillment.public.products**
- **fulfillment.public.products_on_hand**
- **fulfillment.public.customers**
- **fulfillment.public.orders**

The connector applies similar naming conventions to label its transaction metadata topics.

If the default topic name do not meet your requirements, you can configure custom topic names. To configure custom topic names, you specify regular expressions in the logical topic routing SMT. For more information about using the logical topic routing SMT to customize topic naming, see Topic routing.

7.2.5. Metadata in Debezium PostgreSQL change event records

In addition to a *database change event*, each record produced by a PostgreSQL connector contains some metadata. Metadata includes where the event occurred on the server, the name of the source partition and the name of the Kafka topic and partition where the event should go, for example:

```
"sourcePartition": {
    "server": "fulfillment"
},
"sourceOffset": {
    "lsn": "24023128",
    "txId": "555",
    "ts_ms": "1482918357011"
},
"kafkaPartition": null
```

- **sourcePartition** always defaults to the setting of the **database.server.name** connector configuration property.
- **sourceOffset** contains information about the location of the server where the event occurred:
  - **lsn** represents the PostgreSQL Log Sequence Number or **offset** in the transaction log.
  - **txId** represents the identifier of the server transaction that caused the event.

- **ts_ms** represents the server time at which the transaction was committed in the form of the number of milliseconds since the epoch.

- **kafkaPartition** with a setting of **null** means that the connector does not use a specific Kafka partition. The PostgreSQL connector uses only one Kafka Connect partition and it places the generated events into one Kafka partition.

7.2.6. Debezium PostgreSQL connector-generated events that represent transaction boundaries

Debezium can generate events that represent transaction boundaries and that enrich data change event messages.

LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

For every transaction **BEGIN** and **END**, Debezium generates an event that contains the following fields:

- **status** - **BEGIN** or **END**
- **id** - string representation of unique transaction identifier
- **event_count** (for **END** events) - total number of events emitted by the transaction
- **data_collections** (for **END** events) - an array of pairs of **data_collection** and **event_count** that provides the number of events emitted by changes originating from given data collection

Example

```
{
  "status": "BEGIN",
  "id": "571",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "571",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "s1.a",
      "event_count": 1
    },
    {
      "data_collection": "s2.a",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, transaction events are written to the topic named ***database.server.name*.transaction**.

Change data event enrichment

When transaction metadata is enabled the data message **Envelope** is enriched with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

- **id** - string representation of unique transaction identifier
- **total_order** - absolute position of the event among all events generated by the transaction
- **data_collection_order** - the per-data collection position of the event among all events that were emitted by the transaction

Following is an example of a message:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
    ...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "571",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

7.3. DESCRIPTIONS OF DEBEZIUM POSTGRESQL CONNECTOR DATA CHANGE EVENTS

The Debezium PostgreSQL connector generates a data change event for each row-level **INSERT**, **UPDATE**, and **DELETE** operation. Each event contains a key and a value. The structure of the key and the value depends on the table that was changed.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events may change over time, which can be difficult for consumers to handle. To address this, each event contains the schema for its content or, if you are using a schema registry, a schema ID that a consumer can use to obtain the schema from the registry. This makes each event self-contained.

The following skeleton JSON shows the basic four parts of a change event. However, how you configure the Kafka Connect converter that you choose to use in your application determines the representation of these four parts in change events. A **schema** field is in a change event only when you configure the converter to produce it. Likewise, the event key and event payload are in a change event only if you configure a converter to produce it. If you use the JSON converter and you configure it to produce all four basic change event parts, change events have this structure:

```
{
 "schema": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

Table 7.4. Overview of change event basic content

Item	Field name	Description
1	**schema**	The first **schema** field is part of the event key. It specifies a Kafka Connect schema that describes what is in the event key's **payload** portion. In other words, the first **schema** field describes the structure of the primary key, or the unique key if the table does not have a primary key, for the table that was changed. It is possible to override the table's primary key by setting the **message.key.columns** connector configuration property. In this case, the first schema field describes the structure of the key identified by that property.
2	**payload**	The first **payload** field is part of the event key. It has the structure described by the previous **schema** field and it contains the key for the row that was changed.
3	**schema**	The second **schema** field is part of the event value. It specifies the Kafka Connect schema that describes what is in the event value's **payload** portion. In other words, the second **schema** describes the structure of the row that was changed. Typically, this schema contains nested schemas.
4	**payload**	The second **payload** field is part of the event value. It has the structure described by the previous **schema** field and it contains the actual data for the row that was changed.

By default behavior is that the connector streams change event records to topics with names that are the same as the event's originating table.

NOTE

Starting with Kafka 0.10, Kafka can optionally record the event key and value with the *timestamp* at which the message was created (recorded by the producer) or written to the log by Kafka.

WARNING

The PostgreSQL connector ensures that all Kafka Connect schema names adhere to the Avro schema name format. This means that the logical server name must start with a Latin letter or an underscore, that is, a-z, A-Z, or _. Each remaining character in the logical server name and each character in the schema and table names must be a Latin letter, a digit, or an underscore, that is, a-z, A-Z, 0-9, or _. If there is an invalid character it is replaced with an underscore character.

This can lead to unexpected conflicts if the logical server name, a schema name, or a table name contains invalid characters, and the only characters that distinguish names from one another are invalid and thus replaced with underscores.

Details are in the following topics:

- Section 7.3.1, "About keys in Debezium PostgreSQL change events"
- Section 7.3.2, "About values in Debezium PostgreSQL change events"

7.3.1. About keys in Debezium PostgreSQL change events

For a given table, the change event's key has a structure that contains a field for each column in the primary key of the table at the time the event was created. Alternatively, if the table has **REPLICA IDENTITY** set to **FULL** or **USING INDEX** there is a field for each unique key constraint.

Consider a **customers** table defined in the **public** database schema and the example of a change event key for that table.

Example table

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

Example change event key

If the **database.server.name** connector configuration property has the value **PostgreSQL_server**, every change event for the **customers** table while it has this definition has the same key structure, which in JSON looks like this:

```
{
  "schema": { 1
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key", 2
    "optional": false, 3
    "fields": [ 4
```

```
        {
            "name": "id",
            "index": "0",
            "schema": {
                "type": "INT32",
                "optional": "false"
            }
        }
    ]
  },
  "payload": { 5
    "id": "1"
  },
}
```

Table 7.5. Description of change event key

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's **payload** portion.
2	**PostgreSQL_server.inventory.customers.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format *connector-name.database-name.table-name*.**Key**. In this example: • **PostgreSQL_server** is the name of the connector that generated this event. • **inventory** is the database that contains the table that was changed. • **customers** is the table that was updated.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a table does not have a primary key.
4	**fields**	Specifies each field that is expected in the **payload**, including each field's name, index, and schema.
5	**payload**	Contains the key for the row for which this change event was generated. In this example, the key, contains a single **id** field whose value is **1**.

NOTE

Although the **column.exclude.list** and **column.include.list** connector configuration properties allow you to capture only a subset of table columns, all columns in a primary or unique key are always included in the event's key.

WARNING

If the table does not have a primary or unique key, then the change event's key is null. The rows in a table without a primary or unique key constraint cannot be uniquely identified.

7.3.2. About values in Debezium PostgreSQL change events

The value in a change event is a bit more complicated than the key. Like the key, the value has a **schema** section and a **payload** section. The **schema** section contains the schema that describes the **Envelope** structure of the **payload** section, including its nested fields. Change events for operations that create, update or delete data all have a value payload with an envelope structure.

Consider the same sample table that was used to show an example of a change event key:

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

The value portion of a change event for a change to this table varies according to the **REPLICA IDENTITY** setting and the operation that the event is for.

Details follow in these sections:

- Replica identity
- *create* events
- *update* events
- Primary key updates
- *delete* events
- Tombstone events

Replica identity

REPLICA IDENTITY is a PostgreSQL-specific table-level setting that determines the amount of information that is available to the logical decoding plug-in for **UPDATE** and **DELETE** events. More specifically, the setting of **REPLICA IDENTITY** controls what (if any) information is available for the previous values of the table columns involved, whenever an **UPDATE** or **DELETE** event occurs.

There are 4 possible values for **REPLICA IDENTITY**:

- **DEFAULT** - The default behavior is that **UPDATE** and **DELETE** events contain the previous values for the primary key columns of a table if that table has a primary key. For an **UPDATE** event, only the primary key columns with changed values are present.

If a table does not have a primary key, the connector does not emit **UPDATE** or **DELETE** events for that table. For a table without a primary key, the connector emits only *create* events. Typically, a table without a primary key is used for appending messages to the end of the table, which means that **UPDATE** and **DELETE** events are not useful.

- **NOTHING** - Emitted events for **UPDATE** and **DELETE** operations do not contain any information about the previous value of any table column.
- **FULL** - Emitted events for **UPDATE** and **DELETE** operations contain the previous values of all columns in the table.
- **INDEX** *index-name* - Emitted events for **UPDATE** and **DELETE** operations contain the previous values of the columns contained in the specified index. **UPDATE** events also contain the indexed columns with the updated values.

create events

The following example shows the value portion of a change event that the connector generates for an operation that creates data in the **customers** table:

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "PostgreSQL_server.inventory.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
```

```
                "type": "int32",
                "optional": false,
                "field": "id"
              },
              {
                "type": "string",
                "optional": false,
                "field": "first_name"
              },
              {
                "type": "string",
                "optional": false,
                "field": "last_name"
              },
              {
                "type": "string",
                "optional": false,
                "field": "email"
              }
            ],
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "after"
          },
          {
            "type": "struct",
            "fields": [
              {
                "type": "string",
                "optional": false,
                "field": "version"
              },
              {
                "type": "string",
                "optional": false,
                "field": "connector"
              },
              {
                "type": "string",
                "optional": false,
                "field": "name"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "ts_ms"
              },
              {
                "type": "boolean",
                "optional": true,
                "default": false,
                "field": "snapshot"
              },
              {
                "type": "string",
                "optional": false,
```

```
                "field": "db"
            },
            {
                "type": "string",
                "optional": false,
                "field": "schema"
            },
            {
                "type": "string",
                "optional": false,
                "field": "table"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "txId"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "lsn"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "xmin"
            }
        ],
        "optional": false,
        "name": "io.debezium.connector.postgresql.Source", 3
        "field": "source"
    },
    {
        "type": "string",
        "optional": false,
        "field": "op"
    },
    {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
    }
  ],
  "optional": false,
  "name": "PostgreSQL_server.inventory.customers.Envelope" 4
},
"payload": { 5
  "before": null, 6
  "after": { 7
    "id": 1,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": { 8
```

```
            "version": "1.9.7.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": true,
            "db": "postgres",
            "sequence": "[\"24023119\",\"24023128\"]"
            "schema": "public",
            "table": "customers",
            "txId": 555,
            "lsn": 24023128,
            "xmin": null
        },
        "op": "c", 9
        "ts_ms": 1559033904863 10
    }
}
```

Table 7.6. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **PostgreSQL_server.inventory.customers.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.postgresql.Source** is the schema for the payload's **source** field. This schema is specific to the PostgreSQL connector. The connector uses it for all events that it generates.
4	**name**	**PostgreSQL_server.inventory.customers.Envelope** is the schema for the overall structure of the payload, where **PostgreSQL_server** is the connector name, **inventory** is the database, and **customers** is the table.

Item	Field name	Description
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content. **NOTE** Whether or not this field is available is dependent on the **REPLICA IDENTITY** setting for each table.
7	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **id**, **first_name**, **last_name**, and **email** columns.
8	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: ● Debezium version ● Connector type and name ● Database and table that contains the new row ● Stringified JSON array of additional offset information. The first value is always the last committed LSN, the second value is always the current LSN. Either value may be **null**. ● Schema name ● If the event was part of a snapshot ● ID of the transaction in which the operation was performed ● Offset of the operation in the database log ● Timestamp for when the change was made in the database

Item	Field name	Description
9	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots) • **t** = truncate • **m** = message
10	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

update events

The value of a change event for an update in the sample **customers** table has the same schema as a *create* event for that table. Likewise, the event value's payload has the same structure. However, the event value payload contains different values in an *update* event. Here is an example of a change event value in an event that the connector generates for an update in the **customers** table:

```
{
    "schema": { ... },
    "payload": {
      "before": { 1
          "id": 1
      },
      "after": { 2
          "id": 1,
          "first_name": "Anne Marie",
          "last_name": "Kretchmar",
          "email": "annek@noanswer.org"
      },
      "source": { 3
          "version": "1.9.7.Final",
          "connector": "postgresql",
          "name": "PostgreSQL_server",
          "ts_ms": 1559033904863,
          "snapshot": false,
          "db": "postgres",
          "schema": "public",
```

```
            "table": "customers",
            "txId": 556,
            "lsn": 24023128,
            "xmin": null
        },
        "op": "u", 4
        "ts_ms": 1465584025523 5
    }
}
```

Table 7.7. Descriptions of *update* event value fields

Item	Field name	Description
1	**before**	An optional field that contains values that were in the row before the database commit. In this example, only the primary key column, **id**, is present because the table's **REPLICA IDENTITY** setting is, by default, **DEFAULT**. + For an *update* event to contain the previous values of all columns in the row, you would have to change the **customers** table by running **ALTER TABLE customers REPLICA IDENTITY FULL**.
2	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **first_name** value is now **Anne Marie**.
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different. The source metadata includes: • Debezium version • Connector type and name • Database and table that contains the new row • Schema name • If the event was part of a snapshot (always **false** for *update* events) • ID of the transaction in which the operation was performed • Offset of the operation in the database log • Timestamp for when the change was made in the database
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.

Item	Field name	Description
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

Updating the columns for a row's primary/unique key changes the value of the row's key. When a key changes, Debezium outputs *three* events: a **DELETE** event and a tombstone event with the old key for the row, followed by an event with the new key for the row. Details are in the next section.

Primary key updates

An **UPDATE** operation that changes a row's primary key field(s) is known as a primary key change. For a primary key change, in place of sending an **UPDATE** event record, the connector sends a **DELETE** event record for the old key and a **CREATE** event record for the new (updated) key. These events have the usual structure and content, and in addition, each one has a message header related to the primary key change:

- The **DELETE** event record has **__debezium.newkey** as a message header. The value of this header is the new primary key for the updated row.
- The **CREATE** event record has **__debezium.oldkey** as a message header. The value of this header is the previous (old) primary key that the updated row had.

delete events

The value in a *delete* change event has the same **schema** portion as *create* and *update* events for the same table. The **payload** portion in a *delete* event for the sample **customers** table looks like this:

```
{
    "schema": { ... },
    "payload": {
        "before": { 1
            "id": 1
        },
        "after": null, 2
        "source": { 3
            "version": "1.9.7.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
```

```
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "d", 4
        "ts_ms": 1465581902461 5
    }
}
```

Table 7.8. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit. In this example, the **before** field contains only the primary key column because the table's **REPLICA IDENTITY** setting is **DEFAULT**.
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and **lsn** field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector type and name • Database and table that contained the deleted row • Schema name • If the event was part of a snapshot (always **false** for *delete* events) • ID of the transaction in which the operation was performed • Offset of the operation in the database log • Timestamp for when the change was made in the database
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.

Item	Field name	Description
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

A *delete* change event record provides a consumer with the information it needs to process the removal of this row.

WARNING

For a consumer to be able to process a *delete* event generated for a table that does not have a primary key, set the table's **REPLICA IDENTITY** to **FULL**. When a table does not have a primary key and the table's **REPLICA IDENTITY** is set to **DEFAULT** or **NOTHING**, a *delete* event has no **before** field.

PostgreSQL connector events are designed to work with Kafka log compaction. Log compaction enables removal of some older messages as long as at least the most recent message for every key is kept. This lets Kafka reclaim storage space while ensuring that the topic contains a complete data set and can be used for reloading key-based state.

Tombstone events

When a row is deleted, the *delete* event value still works with log compaction, because Kafka can remove all earlier messages that have that same key. However, for Kafka to remove all messages that have that same key, the message value must be **null**. To make this possible, the PostgreSQL connector follows a *delete* event with a special *tombstone* event that has the same key but a **null** value.

truncate events

A *truncate* change event signals that a table has been truncated. The message key is **null** in this case, the message value looks like this:

```
{
    "schema": { ... },
    "payload": {
        "source": { 1
            "version": "1.9.7.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "public",
```

```
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "t", 2
        "ts_ms": 1559033904961 3
    }
}
```

Table 7.9. Descriptions of *truncate* event value fields

Item	Field name	Description
1	**source**	Mandatory field that describes the source metadata for the event. In a *truncate* event value, the **source** field structure is the same as for *create*, *update*, and *delete* events for the same table, provides this metadata: • Debezium version • Connector type and name • Database and table that contains the new row • Schema name • If the event was part of a snapshot (always **false** for *delete* events) • ID of the transaction in which the operation was performed • Offset of the operation in the database log • Timestamp for when the change was made in the database
2	**op**	Mandatory string that describes the type of operation. The **op** field value is **t**, signifying that this table was truncated.
3	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

In case a single **TRUNCATE** statement applies to multiple tables, one *truncate* change event record for each truncated table will be emitted.

Note that since *truncate* events represent a change made to an entire table and don't have a message key, unless you're working with topics with a single partition, there are no ordering guarantees for the change events pertaining to a table (*create*, *update*, etc.) and *truncate* events for that table. For instance a consumer may receive an *update* event only after a *truncate* event for that table, when those events are read from different partitions.

MESSAGE EVENTS

This event type is only supported through the **pgoutput** plugin on Postgres 14+ (Postgres Documentation)

A *message* event signals that a generic logical decoding message has been inserted directly into the WAL typically with the **pg_logical_emit_message** function. The message key is a **Struct** with a single field named **prefix** in this case, carrying the prefix specified when inserting the message. The message value looks like this for transactional messages:

```
{
    "schema": { ... },
    "payload": {
        "source": { (1)
            "version": "1.9.7.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "",
            "table": "",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m", (2)
        "ts_ms": 1559033904961, (3)
        "message": { (4)
            "prefix": "foo",
            "content": "Ymfy"
        }
    }
}
```

Unlike other event types, non-transactional messages will not have any associated **BEGIN** or **END** transaction events. The message value looks like this for non-transactional messages:

```
{
    "schema": { ... },
    "payload": {
        "source": { (1)
            "version": "1.9.7.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": false,
            "db": "postgres",
            "schema": "",
            "table": "",
            "lsn": 46523128,
            "xmin": null
        },
        "op": "m", (2)
```

```
        "ts_ms": 1559033904961 3
        "message": { 4
            "prefix": "foo",
            "content": "Ymfy"
        }
    }
```

Table 7.10. Descriptions of *message* event value fields

Item	Field name	Description
1	**source**	Mandatory field that describes the source metadata for the event. In a *message* event value, the **source** field structure will not have **table** or **schema** information for any *message* events and will only have **txId** if the *message* event is transactional. • Debezium version • Connector type and name • Database name • Schema name (always **""** for *message* events) • Table name (always **""** for *message* events) • If the event was part of a snapshot (always **false** for *message* events) • ID of the transaction in which the operation was performed (**null** for non-transactional *message* events) • Offset of the operation in the database log • Transactional messages: Timestamp for when the message was inserted into the WAL • Non-Transactional messages; Timestamp for when the connector encounters the message
2	**op**	Mandatory string that describes the type of operation. The **op** field value is **m**, signifying that this is a*message* event.

Item	Field name	Description
3	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. For transactional *message* events, the **ts_ms** attribute of the **source** object indicates the time that the change was made in the database for transactional *message* events. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium. For non-transactional *message* events, the **source** object's **ts_ms** indicates time at which the connector encounters the *message* event, while the **payload.ts_ms** indicates the time at which the connector processed the event. This difference is due to the fact that the commit timestamp is not present in Postgres's generic logical message format and non-transactional logical messages are not preceded by a **BEGIN** event (which has timestamp information).
4	**message**	Field that contains the message metadata • Prefix (text) • Content (byte array that is encoded based on the binary handling mode setting)

7.4. HOW DEBEZIUM POSTGRESQL CONNECTORS MAP DATA TYPES

The PostgreSQL connector represents changes to rows with events that are structured like the table in which the row exists. The event contains a field for each column value. How that value is represented in the event depends on the PostgreSQL data type of the column. The following sections describe how the connector maps PostgreSQL data types to a *literal type* and a *semantic type* in event fields.

- *literal type* describes how the value is literally represented using Kafka Connect schema types: **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**.
- *semantic type* describes how the Kafka Connect schema captures the *meaning* of the field using the name of the Kafka Connect schema for the field.

If the default data type conversions do not meet your needs, you can create a custom converter for the connector.

Details are in the following sections:

- Basic types
- Temporal types
- TIMESTAMP type
- Decimal types

- HSTORE type
- Domain types
- Network address types
- PostGIS types
- Toasted values

Basic types

The following table describes how the connector maps basic types.

Table 7.11. Mappings for PostgreSQL basic data types

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
BOOLEAN	**BOOLEAN**	n/a
BIT(1)	**BOOLEAN**	n/a
BIT(> 1)	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits. The resulting **byte[]** contains the bits in little-endian form and is sized to contain the specified number of bits. For example, **numBytes = n/8 + (n % 8 == 0 ? 0 : 1)** where **n** is the number of bits.
BIT VARYING[(M)]	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits (2^31 - 1 in case no length is given for the column). The resulting **byte[]** contains the bits in little-endian form and is sized based on the content. The specified size **(M)** is stored in the length parameter of the **io.debezium.data.Bits** type.
SMALLINT, **SMALLSERIAL**	**INT16**	n/a
INTEGER, **SERIAL**	**INT32**	n/a
BIGINT, **BIGSERIAL**, **OID**	**INT64**	n/a
REAL	**FLOAT32**	n/a
DOUBLE PRECISION	**FLOAT64**	n/a

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
CHAR[(M)]	**STRING**	n/a
VARCHAR[(M)]	**STRING**	n/a
CHARACTER[(M)]	**STRING**	n/a
CHARACTER VARYING[(M)]	**STRING**	n/a
TIMESTAMPTZ, **TIMESTAMP WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp with timezone information, where the timezone is GMT.
TIMETZ, **TIME WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTime** A string representation of a time value with timezone information, where the timezone is GMT.
INTERVAL [P]	**INT64**	**io.debezium.time.MicroDuration** (default) The approximate number of microseconds for a time interval using the **365.25** / **12.0** formula for days per month average.
INTERVAL [P]	**STRING**	**io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.
BYTEA	**BYTES** or **STRING**	n/a Either the raw bytes (the default), a base64-encoded string, or a hex-encoded string, based on the connector's binary handling mode setting. Debezium only supports Postgres **bytea_output** configuration of value **hex**. See this documentation on Postgres binary data types.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
JSON, **JSONB**	**STRING**	**io.debezium.data.Json** Contains the string representation of a JSON document, array, or scalar.
XML	**STRING**	**io.debezium.data.Xml** Contains the string representation of an XML document.
UUID	**STRING**	**io.debezium.data.Uuid** Contains the string representation of a PostgreSQL UUID value.
POINT	**STRUCT**	**io.debezium.data.geometry.Point** Contains a structure with two **FLOAT64** fields, **(x,y)**. Each field represents the coordinates of a geometric point.
LTREE	**STRING**	**io.debezium.data.Ltree** Contains the string representation of a PostgreSQL LTREE value.
CITEXT	**STRING**	n/a
INET	**STRING**	n/a
INT4RANGE	**STRING**	n/a Range of integer.
INT8RANGE	**STRING**	n/a Range of **bigint**.
NUMRANGE	**STRING**	n/a Range of **numeric**.
TSRANGE	**STRING**	n/a Contains the string representation of a timestamp range without a time zone.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
TSTZRANGE	**STRING**	n/a Contains the string representation of a timestamp range with the local system time zone.
DATERANGE	**STRING**	n/a Contains the string representation of a date range. It always has an exclusive upper-bound.
ENUM	**STRING**	**io.debezium.data.Enum** Contains the string representation of the PostgreSQL **ENUM** value. The set of allowed values is maintained in the **allowed** schema parameter.

Temporal types

Other than PostgreSQL's **TIMESTAMPTZ** and **TIMETZ** data types, which contain time zone information, how temporal types are mapped depends on the value of the **time.precision.mode** connector configuration property. The following sections describe these mappings:

- **time.precision.mode=adaptive**
- **time.precision.mode=adaptive_time_microseconds**
- **time.precision.mode=connect**

time.precision.mode=adaptive

When the **time.precision.mode** property is set to **adaptive**, the default, the connector determines the literal type and semantic type based on the column's data type definition. This ensures that events *exactly* represent the values in the database.

Table 7.12. Mappings when time.precision.mode is adaptive

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**io.debezium.time.Date** Represents the number of days since the epoch.
TIME(1), **TIME(2)**, **TIME(3)**	**INT32**	**io.debezium.time.Time** Represents the number of milliseconds past midnight, and does not include timezone information.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIME(4), **TIME(5)**, **TIME(6)**	**INT64**	**io.debezium.time.MicroTime** Represents the number of microseconds past midnight, and does not include timezone information.
TIMESTAMP(1), **TIMESTAMP(2)**, **TIMESTAMP(3)**	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
TIMESTAMP(4), **TIMESTAMP(5)**, **TIMESTAMP(6)**, **TIMESTAMP**	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds since the epoch, and does not include timezone information.

time.precision.mode=adaptive_time_microseconds

When the **time.precision.mode** configuration property is set to **adaptive_time_microseconds**, the connector determines the literal type and semantic type for temporal types based on the column's data type definition. This ensures that events *exactly* represent the values in the database, except all **TIME** fields are captured as microseconds.

Table 7.13. Mappings when **time.precision.mode** is **adaptive_time_microseconds**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**io.debezium.time.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**io.debezium.time.MicroTime** Represents the time value in microseconds and does not include timezone information. PostgreSQL allows precision **P** to be in the range 0-6 to store up to microsecond precision.
TIMESTAMP(1) , **TIMESTAMP(2)**, **TIMESTAMP(3)**	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds past the epoch, and does not include timezone information.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIMESTAMP(4) , **TIMESTAMP(5)**, **TIMESTAMP(6)**, **TIMESTAMP**	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds past the epoch, and does not include timezone information.

time.precision.mode=connect

When the **time.precision.mode** configuration property is set to **connect**, the connector uses Kafka Connect logical types. This may be useful when consumers can handle only the built-in Kafka Connect logical types and are unable to handle variable-precision time values. However, since PostgreSQL supports microsecond precision, the events generated by a connector with the **connect** time precision mode **results in a loss of precision**when the database column has a *fractional second precision* value that is greater than 3.

Table 7.14. Mappings when **time.precision.mode** is **connect**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**org.apache.kafka.connect.data.Time** Represents the number of milliseconds since midnight, and does not include timezone information. PostgreSQL allows **P** to be in the range 0-6 to store up to microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.
TIMESTAMP([P])	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information. PostgreSQL allows **P** to be in the range 0-6 to store up to microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.

TIMESTAMP type

The **TIMESTAMP** type represents a timestamp without time zone information. Such columns are converted into an equivalent Kafka Connect value based on UTC. For example, the **TIMESTAMP** value "2018-06-20 15:13:16.945104" is represented by an **io.debezium.time.MicroTimestamp** with the value "1529507596945104" when **time.precision.mode** is not set to **connect**.

The timezone of the JVM running Kafka Connect and Debezium does not affect this conversion.

PostgreSQL supports using **+/-infinite** values in **TIMESTAMP** columns. These special values are

converted to timestamps with value **9223372036825200000** in case of positive infinity or **-9223372036832400000** in case of negative infinity. This behaviour mimics the standard behaviour of PostgreSQL JDBC driver - see **org.postgresql.PGStatement** interface for reference.

Decimal types

The setting of the PostgreSQL connector configuration property **decimal.handling.mode** determines how the connector maps decimal types.

When the **decimal.handling.mode** property is set to **precise**, the connector uses the Kafka Connect **org.apache.kafka.connect.data.Decimal** logical type for all **DECIMAL**, **NUMERIC** and **MONEY** columns. This is the default mode.

Table 7.15. Mappings when decimal.handling.mode is precise

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
NUMERIC[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted.
DECIMAL[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted.
MONEY[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted. The **scale** schema parameter is determined by the **money.fraction.digits** connector configuration property.

There is an exception to this rule. When the **NUMERIC** or **DECIMAL** types are used without scale constraints, the values coming from the database have a different (variable) scale for each value. In this case, the connector uses **io.debezium.data.VariableScaleDecimal**, which contains both the value and the scale of the transferred value.

Table 7.16. Mappings of DECIMAL and NUMERIC types when there are no scale constraints

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
NUMERIC	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.
DECIMAL	**STRUCT**	**io.debezium.data.VariableScaleDecimal** Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.

When the **decimal.handling.mode** property is set to **double**, the connector represents all **DECIMAL**, **NUMERIC** and **MONEY** values as Java double values and encodes them as shown in the following table.

Table 7.17. Mappings when **decimal.handling.mode** is **double**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
NUMERIC[(M[,D])]	**FLOAT64**	
DECIMAL[(M[,D])]	**FLOAT64**	
MONEY[(M[,D])]	**FLOAT64**	

The last possible setting for the **decimal.handling.mode** configuration property is **string**. In this case, the connector represents **DECIMAL**, **NUMERIC** and **MONEY** values as their formatted string representation, and encodes them as shown in the following table.

Table 7.18. Mappings when **decimal.handling.mode** is **string**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
NUMERIC[(M[,D])]	**STRING**	
DECIMAL[(M[,D])]	**STRING**	
MONEY[(M[,D])]	**STRING**	

PostgreSQL supports **NaN** (not a number) as a special value to be stored in **DECIMAL**/**NUMERIC** values when the setting of **decimal.handling.mode** is **string** or **double**. In this case, the connector encodes **NaN** as either **Double.NaN** or the string constant **NAN**.

HSTORE type

The setting of the PostgreSQL connector configuration property **hstore.handling.mode** determines how the connector maps **HSTORE** values.

When the **dhstore.handling.mode** property is set to **json** (the default), the connector represents **HSTORE** values as string representations of JSON values and encodes them as shown in the following table. When the **hstore.handling.mode** property is set to **map**, the connector uses the **MAP** schema type for **HSTORE** values.

Table 7.19. Mappings for HSTORE data type

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
HSTORE	**STRING**	**io.debezium.data.Json** Example: output representation using the JSON converter is **{"key" : "val"}**
HSTORE	**MAP**	n/a Example: output representation using the JSON converter is **{"key" : "val"}**

Domain types

PostgreSQL supports user-defined types that are based on other underlying types. When such column types are used, Debezium exposes the column's representation based on the full type hierarchy.

IMPORTANT

Capturing changes in columns that use PostgreSQL domain types requires special consideration. When a column is defined to contain a domain type that extends one of the default database types and the domain type defines a custom length or scale, the generated schema inherits that defined length or scale.

When a column is defined to contain a domain type that extends another domain type that defines a custom length or scale, the generated schema does **not** inherit the defined length or scale because that information is not available in the PostgreSQL driver's column metadata.

Network address types

PostgreSQL has data types that can store IPv4, IPv6, and MAC addresses. It is better to use these types instead of plain text types to store network addresses. Network address types offer input error checking and specialized operators and functions.

Table 7.20. Mappings for network address types

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
INET	**STRING**	n/a IPv4 and IPv6 networks
CIDR	**STRING**	n/a IPv4 and IPv6 hosts and networks
MACADDR	**STRING**	n/a MAC addresses
MACADDR8	**STRING**	n/a MAC addresses in EUI-64 format

PostGIS types

The PostgreSQL connector supports all PostGIS data types.

Table 7.21. Mappings of PostGIS data types

PostGIS data type	Literal type (schema type)	Semantic type (schema name) and Notes
GEOMETRY (planar)	**STRUCT**	**io.debezium.data.geometry.Geometry** Contains a structure with two fields: • **srid (INT32)** - Spatial Reference System Identifier that defines what type of geometry object is stored in the structure. • **wkb (BYTES)** - A binary representation of the geometry object encoded in the Well-Known-Binary format. For format details, see Open Geospatial Consortium Simple Features Access specification.

PostGIS data type	Literal type (schema type)	Semantic type (schema name) and Notes
GEOGRAPHY (spherical)	**STRUCT**	**io.debezium.data.geometry.Geography** Contains a structure with two fields: • **srid (INT32)** - Spatial Reference System Identifier that defines what type of geography object is stored in the structure. • **wkb (BYTES)** - A binary representation of the geometry object encoded in the Well-Known-Binary format. For format details, see Open Geospatial Consortium Simple Features Access specification.

Toasted values

PostgreSQL has a hard limit on the page size. This means that values that are larger than around 8 KBs need to be stored by using TOAST storage. This impacts replication messages that are coming from the database. Values that were stored by using the TOAST mechanism and that have not been changed are not included in the message, unless they are part of the table's replica identity. There is no safe way for Debezium to read the missing value out-of-bands directly from the database, as this would potentially lead to race conditions. Consequently, Debezium follows these rules to handle toasted values:

- Tables with **REPLICA IDENTITY FULL** - TOAST column values are part of the **before** and **after** fields in change events just like any other column.
- Tables with **REPLICA IDENTITY DEFAULT** - When receiving an **UPDATE** event from the database, any unchanged TOAST column value that is not part of the replica identity is not contained in the event. Similarly, when receiving a **DELETE** event, no TOAST columns, if any, are in the **before** field. As Debezium cannot safely provide the column value in this case, the connector returns a placeholder value as defined by the connector configuration property, **unavailable.value.placeholder**.

Default values

If a default value is specified for a column in the database schema, the PostgreSQL connector will attempt to propagate this value to the Kafka schema whenever possible. Most common data types are supported, including:

- **BOOLEAN**
- Numeric types (**INT**, **FLOAT**, **NUMERIC**, etc.)
- Text types (**CHAR**, **VARCHAR**, **TEXT**, etc.)
- Temporal types (**DATE**, **TIME**, **INTERVAL**, **TIMESTAMP**, **TIMESTAMPTZ**)
- **JSON**, **JSONB**, **XML**
- **UUID**

Note that for temporal types, parsing of the default value is provided by PostgreSQL libraries; therefore, any string representation which is normally supported by PostgreSQL should also be supported by the connector.

In the case that the default value is generated by a function rather than being directly specified in-line, the connector will instead export the equivalent of **0** for the given data type. These values include:

- **FALSE** for **BOOLEAN**
- **0** with appropriate precision, for numeric types
- Empty string for text/XML types
- **{}** for JSON types
- **1970-01-01** for **DATE**, **TIMESTAMP**, **TIMESTAMPTZ** types
- **00:00** for **TIME**
- **EPOCH** for **INTERVAL**
- **00000000-0000-0000-0000-000000000000** for **UUID**

This support currently extends only to explicit usage of functions. For example, **CURRENT_TIMESTAMP(6)** is supported with parentheses, but **CURRENT_TIMESTAMP** is not.

IMPORTANT

Support for the propagation of default values exists primarily to allow for safe schema evolution when using the PostgreSQL connector with a schema registry which enforces compatibility between schema versions. Due to this primary concern, as well as the refresh behaviours of the different plug-ins, the default value present in the Kafka schema is not guaranteed to always be in-sync with the default value in the database schema.

- Default values may appear 'late' in the Kafka schema, depending on when/how a given plugin triggers refresh of the in-memory schema. Values may never appear/be skipped in the Kafka schema if the default changes multiple times in-between refreshes
- Default values may appear 'early' in the Kafka schema, if a schema refresh is triggered while the connector has records waiting to be processed. This is due to the column metadata being read from the database at refresh time, rather than being present in the replication message. This may occur if the connector is behind and a refresh occurs, or on connector start if the connector was stopped for a time while updates continued to be written to the source database.

This behaviour may be unexpected, but it is still safe. Only the schema definition is affected, while the real values present in the message will remain consistent with what was written to the source database.

7.5. SETTING UP POSTGRESQL TO RUN A DEBEZIUM CONNECTOR

This release of Debezium supports only the native **pgoutput** logical replication stream. To set up PostgreSQL so that it uses the **pgoutput** plug-in, you must enable a replication slot, and configure a user with sufficient privileges to perform the replication.

Details are in the following topics:

- Section 7.5.1, "Configuring a replication slot for the Debezium **pgoutput** plug-in"
- Section 7.5.2, "Setting up PostgreSQL permissions for the Debezium connector"
- Section 7.5.3, "Setting privileges to enable Debezium to create PostgreSQL publications"
- Section 7.5.4, "Configuring PostgreSQL to allow replication with the Debezium connector host"
- Section 7.5.5, "Configuring PostgreSQL to manage Debezium WAL disk space consumption"

7.5.1. Configuring a replication slot for the Debezium pgoutput plug-in

PostgreSQL's logical decoding uses replication slots. To configure a replication slot, specify the following in the **postgresql.conf** file:

```
wal_level=logical
max_wal_senders=1
max_replication_slots=1
```

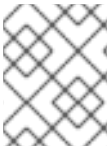
These settings instruct the PostgreSQL server as follows:

- **wal_level** - Use logical decoding with the write-ahead log.
- **max_wal_senders** - Use a maximum of one separate process for processing WAL changes.
- **max_replication_slots** - Allow a maximum of one replication slot to be created for streaming WAL changes.

Replication slots are guaranteed to retain all WAL entries that are required for Debezium even during Debezium outages. Consequently, it is important to closely monitor replication slots to avoid:

- Too much disk consumption
- Any conditions, such as catalog bloat, that can happen if a replication slot stays unused for too long

For more information, see the PostgreSQL documentation for replication slots.



NOTE

Familiarity with the mechanics and configuration of the PostgreSQL write-ahead log is helpful for using the Debezium PostgreSQL connector.

7.5.2. Setting up PostgreSQL permissions for the Debezium connector

Setting up a PostgreSQL server to run a Debezium connector requires a database user that can perform replications. Replication can be performed only by a database user that has appropriate permissions and only for a configured number of hosts.

Although, by default, superusers have the necessary **REPLICATION** and **LOGIN** roles, as mentioned in Security, it is best not to provide the Debezium replication user with elevated privileges. Instead, create a Debezium user that has the minimum required privileges.

Prerequisites

- PostgreSQL administrative permissions.

Procedure

1. To provide a user with replication permissions, define a PostgreSQL role that has *at least* the **REPLICATION** and **LOGIN** permissions, and then grant that role to the user. For example:

```
CREATE ROLE <name> REPLICATION LOGIN;
```

7.5.3. Setting privileges to enable Debezium to create PostgreSQL publications

Debezium streams change events for PostgreSQL source tables from *publications* that are created for the tables. Publications contain a filtered set of change events that are generated from one or more tables. The data in each publication is filtered based on the publication specification. The specification can be created by the PostgreSQL database administrator or by the Debezium connector. To permit the Debezium PostgreSQL connector to create publications and specify the data to replicate to them, the connector must operate with specific privileges in the database.

There are several options for determining how publications are created. In general, it is best to manually create publications for the tables that you want to capture, before you set up the connector. However, you can configure your environment in a way that permits Debezium to create publications automatically, and to specify the data that is added to them.

Debezium uses include list and exclude list properties to specify how data is inserted in the publication. For more information about the options for enabling Debezium to create publications, see **publication.autocreate.mode**.

For Debezium to create a PostgreSQL publication, it must run as a user that has the following privileges:

- Replication privileges in the database to add the table to a publication.
- **CREATE** privileges on the database to add publications.
- **SELECT** privileges on the tables to copy the initial table data. Table owners automatically have **SELECT** permission for the table.

To add tables to a publication, the user must be an owner of the table. But because the source table already exists, you need a mechanism to share ownership with the original owner. To enable shared ownership, you create a PostgreSQL replication group, and then add the existing table owner and the replication user to the group.

Procedure

1. Create a replication group.

```
CREATE ROLE <replication_group>;
```

2. Add the original owner of the table to the group.

```
GRANT REPLICATION_GROUP TO <original_owner>;
```

3. Add the Debezium replication user to the group.

```
GRANT REPLICATION_GROUP TO <replication_user>;
```

4. Transfer ownership of the table to **<replication_group>**.

```
ALTER TABLE <table_name> OWNER TO REPLICATION_GROUP;
```

For Debezium to specify the capture configuration, the value of **publication.autocreate.mode** must be set to **filtered**.

7.5.4. Configuring PostgreSQL to allow replication with the Debezium connector host

To enable Debezium to replicate PostgreSQL data, you must configure the database to permit replication with the host that runs the PostgreSQL connector. To specify the clients that are permitted to replicate with the database, add entries to the PostgreSQL host-based authentication file, **pg_hba.conf**. For more information about the **pg_hba.conf** file, see the PostgreSQL documentation.

Procedure

- Add entries to the **pg_hba.conf** file to specify the Debezium connector hosts that can replicate with the database host. For example,

  pg_hba.conf file example:

```
local   replication     <youruser>                    trust  (1)
host    replication     <youruser> 127.0.0.1/32       trust  (2)
host    replication     <youruser> ::1/128            trust  (3)
```

(1) Instructs the server to allow replication for **<youruser>** locally, that is, on the server machine.

(2) Instructs the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV4**.

(3) Instructs the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV6**.

NOTE

For more information about network masks, see the PostgreSQL documentation.

7.5.5. Configuring PostgreSQL to manage Debezium WAL disk space consumption

In certain cases, it is possible for PostgreSQL disk space consumed by WAL files to spike or increase out of usual proportions. There are several possible reasons for this situation:

- The LSN up to which the connector has received data is available in the **confirmed_flush_lsn** column of the server's **pg_replication_slots** view. Data that is older than this LSN is no longer available, and the database is responsible for reclaiming the disk space.

Also in the **pg_replication_slots** view, the **restart_lsn** column contains the LSN of the oldest WAL that the connector might require. If the value for **confirmed_flush_lsn** is regularly increasing and the value of **restart_lsn** lags then the database needs to reclaim the space.

The database typically reclaims disk space in batch blocks. This is expected behavior and no action by a user is necessary.

- There are many updates in a database that is being tracked but only a tiny number of updates are related to the table(s) and schema(s) for which the connector is capturing changes. This situation can be easily solved with periodic heartbeat events. Set the **heartbeat.interval.ms** connector configuration property.
- The PostgreSQL instance contains multiple databases and one of them is a high-traffic database. Debezium captures changes in another database that is low-traffic in comparison to the other database. Debezium then cannot confirm the LSN as replication slots work per-database and Debezium is not invoked. As WAL is shared by all databases, the amount used tends to grow until an event is emitted by the database for which Debezium is capturing changes. To overcome this, it is necessary to:
  - Enable periodic heartbeat record generation with the **heartbeat.interval.ms** connector configuration property.
  - Regularly emit change events from the database for which Debezium is capturing changes.

  A separate process would then periodically update the table by either inserting a new row or repeatedly updating the same row. PostgreSQL then invokes Debezium, which confirms the latest LSN and allows the database to reclaim the WAL space. This task can be automated by means of the **heartbeat.action.query** connector configuration property.

7.6. DEPLOYMENT OF DEBEZIUM POSTGRESQL CONNECTORS

You can use either of the following methods to deploy a Debezium PostgreSQL connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .

Additional resources

- Section 7.6.5, “Descriptions of Debezium PostgreSQL connector configuration properties”

7.6.1. PostgreSQL connector deployment using AMQ Streams

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.

NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

7.6.2. Using AMQ Streams to deploy a Debezium PostgreSQL connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.

- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

   Example 7.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

   In the example that follows, the custom resource is configured to download the following artifacts:

   - The Debezium PostgreSQL connector archive.
   - The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
   - The Debezium scripting SMT archive and the associated scripting engine that you want to use with the Debezium connector. The SMT archive and scripting language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  version: 3.00
  build: 2
    output: 3
      type: imagestream 4
      image: debezium-streams-connect:latest
    plugins: 5
      - name: debezium-connector-postgres
        artifacts:
          - type: zip 6
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
```

```
postgres/1.9.7.Final-redhat-<build_number>/debezium-connector-postgres-1.9.7.Final-redhat-<build_number>-plugin.zip 7
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip 8
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip 9
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar 10
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json3.0.11/groovy-json-3.0.11.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 7.22. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry such as Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. The **type** value must match the type of the file that is referenced in the **url** field.

Item	Description
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) As an alternative, the Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or

ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **postgresql-inventory-connector.yaml**

Example 7.2. postgresql-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-postgresql (1)
spec:
  class: io.debezium.connector.postgresql.PostgresConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: postgresql.debezium-postgresql.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_postgresql (10)
    database.include.list: public.inventory (11)
```

Table 7.23. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.
4	The connector’s configuration.
5	The address of the host database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.

Item	Description
8	The password for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium PostgreSQL deployment.

7.6.3. Deploying a Debezium PostgreSQL connector by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium PostgreSQL connector, you need to build a custom Kafka Connect container image that contains the Debezium connector archive and push this container image to a container registry. You then need to create two custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.
- A **KafkaConnector** CR that defines your Debezium Db2 connector. Apply this CR to the same OpenShift instance where you applied the **KafkaConnect** CR.

Prerequisites

- PostgreSQL is running and you performed the steps to set up PostgreSQL to run a Debezium connector.

- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift .
- Podman or Docker is installed.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium PostgreSQL container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-postgresql.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-postgresql/1.9.7.Final-redhat-<build_number>/debezium-connector-postgresql-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-postgresql-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-postgresql-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

The command creates a Dockerfile with the name **debezium-container-for-postgresql.yaml** in the current directory.

   b. Build the container image from the **debezium-container-for-postgresql.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-postgresql:latest .
```

```
docker build -t debezium-container-for-postgresql:latest .
```

The **build** command builds a container image with the name **debezium-container-for-postgresql**.

c. Push your custom image to a container registry such as **quay.io** or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-postgresql:latest
```

```
docker push <myregistry.io>/debezium-container-for-postgresql:latest
```

d. Create a new Debezium PostgreSQL **KafkaConnect** custom resource (CR). For example, create a **KafkaConnect** CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  image: debezium-container-for-postgresql 2
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply your **KafkaConnect** CR to the OpenShift Kafka instance by running the following command:

```
oc create -f dbz-connect.yaml
```

This updates your Kafka Connect environment in OpenShift to add a Kafka Connector instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium PostgreSQL connector instance.
You configure a Debezium PostgreSQL connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce events for a subset of the schemas and tables, or it might set properties so that Debezium ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed. For the complete list of the configuration properties that you can set for the Debezium PostgreSQL connector, see PostgreSQL connector properties.

The following example configures a Debezium connector that connects to a PostgreSQL server host, **192.168.99.100**, on port **5432**. This host has a database named **sampledb**, a schema named **public**, and **fulfillment** is the server's logical name.

fulfillment-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
  name: fulfillment-connector 1
  labels:
   strimzi.io/cluster: my-connect-cluster
 spec:
  class: io.debezium.connector.postgresql.PostgresConnector
  tasksMax: 1 2
  config: 3
   database.hostname: 192.168.99.100 4
   database.port: 5432
   database.user: debezium
   database.password: dbz
   database.dbname: sampledb
   database.server.name: fulfillment 5
   schema.include.list: public 6
   plugin.name: pgoutput 7
```

1 1 1 1 1 The name of the connector.

2 2 2 2 2 Only one task should operate at any one time. Because the PostgreSQL connector reads the PostgreSQL server's **binlog**, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.

3 3 3 The connector's configuration.

4 4 4 The name of the database host that is running the PostgreSQL server. In this example, the database host name is **192.168.99.100**.

5 5 5 A unique server name. The server name is the logical identifier for the PostgreSQL server or cluster of servers. This name is used as the prefix for all Kafka topics that receive change event records.

6 6 6 The connector captures changes in only the **public** schema. It is possible to configure the connector to capture changes in only the tables that you choose. See **table.include.list** connector configuration property .

7 7 7 The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server. While the only supported value for PostgreSQL 10 and later is **pgoutput**, you must explicitly set **plugin.name** to **pgoutput**.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **fulfillment-connector.yaml** file, you would run the following command:

```
oc apply -f fulfillment-connector.yaml
```

This registers **fulfillment-connector** and the connector starts to run against the **sampledb** database as defined in the **KafkaConnector** CR.

Results

After the connector starts, it performs a consistent snapshot of the PostgreSQL server databases that the connector is configured for. The connector then starts generating data change events for row-level operations and streaming change event records to Kafka topics.

7.6.4. Verifying that the Debezium PostgreSQL connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-postgresql**.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
     a. Enter the following command:

     ```
     oc describe KafkaConnector <connector-name> -n <project>
     ```

     For example,

```
oc describe KafkaConnector inventory-connector-postgresql -n debezium
```

The command returns status information that is similar to the following output:

Example 7.3. KafkaConnector resource status

```
Name:         inventory-connector-postgresql
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  10.131.1.124:8083
    Name:         inventory-connector-postgresql
    Tasks:
      Id:              0
      State:           RUNNING
      worker_id:       10.131.1.124:8083
    Type:              source
  Observed Generation:  1
  Tasks Max:            1
  Topics:
    inventory_connector_postgresql
    inventory_connector_postgresql.inventory.addresses
    inventory_connector_postgresql.inventory.customers
    inventory_connector_postgresql.inventory.geom
    inventory_connector_postgresql.inventory.orders
    inventory_connector_postgresql.inventory.products
    inventory_connector_postgresql.inventory.products_on_hand
Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-postgresql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

- From a terminal window:

a. Enter the following command:

```
oc get kafkatopics
```

The command returns status information that is similar to the following output:

Example 7.4. KafkaTopic resource status

```
NAME                                                                                                CLUSTER
PARTITIONS   REPLICATION FACTOR   READY
connect-cluster-configs                                                                             debezium-
kafka-cluster   1            1                 True
connect-cluster-offsets                                                                             debezium-
kafka-cluster   25           1                 True
connect-cluster-status                                                                              debezium-
kafka-cluster   5            1                 True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
debezium-kafka-cluster   50           1                 True
inventory-connector-postgresql---a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster   1            1                 True
inventory-connector-postgresql.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480          debezium-kafka-cluster
1            1                 True
inventory-connector-postgresql.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1                 True
inventory-connector-postgresql.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5            debezium-kafka-cluster
1            1                 True
inventory-connector-postgresql.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d            debezium-kafka-cluster
1            1                 True
inventory-connector-postgresql.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1                 True
inventory-connector-postgresql.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1                 True
schema-changes.inventory
debezium-kafka-cluster   1            1                 True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1            1                 True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1                 True
```

3. Check topic content.

- From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_postgresql.inventory.products_on_hand
```

The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_postgresql.inventory.addresses**.

For each event in the topic, the command returns information that is similar to the following output:

Example 7.5. Content of a Debezium change event


```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_connector_postgresql.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector_postgresql.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector_postgresql.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.postgresql.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transaction"}],"optional":false,"name":"inventory_connector_postgresql.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.7.Final-redhat-
```

00001","connector":"postgresql","name":"inventory_connector_postgresql","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"postgresql-bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"transaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

7.6.5. Descriptions of Debezium PostgreSQL connector configuration properties

The Debezium PostgreSQL connector has many configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values. Information about the properties is organized as follows:

- Required configuration properties
- Advanced configuration properties
- Pass-through configuration properties

The following configuration properties are *required* unless a default value is available.

Table 7.24. Required connector configuration properties

Property	Default	Description
name	No default	Unique name for the connector. Attempting to register again with the same name will fail. This property is required by all Kafka Connect connectors.
connector.class	No default	The name of the Java class for the connector. Always use a value of **io.debezium.connector.postgresql.PostgresConnector** for the PostgreSQL connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The PostgreSQL connector always uses a single task and therefore does not use this value, so the default is always acceptable.
plugin.name	**decoderbufs**	The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server. The only supported value is **pgoutput**. You must explicitly set **plugin.name** to **pgoutput**.

Property	Default	Description
slot.name	**debezium**	The name of the PostgreSQL logical decoding slot that was created for streaming changes from a particular plug-in for a particular database/schema. The server uses this slot to stream events to the Debezium connector that you are configuring. Slot names must conform to PostgreSQL replication slot naming rules, which state: *"Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character."*
slot.drop.on.stop	**false**	Whether or not to delete the logical replication slot when the connector stops in a graceful, expected way. The default behavior is that the replication slot remains configured for the connector when the connector stops. When the connector restarts, having the same replication slot enables the connector to start processing where it left off. Set to **true** in only testing or development environments. Dropping the slot allows the database to discard WAL segments. When the connector restarts it performs a new snapshot or it can continue from a persistent offset in the Kafka Connect offsets topic.
publication.name	**dbz_publication**	The name of the PostgreSQL publication created for streaming changes when using **pgoutput**. This publication is created at start-up if it does not already exist and it includes *all tables*. Debezium then applies its own include/exclude list filtering, if configured, to limit the publication to change events for the specific tables of interest. The connector user must have superuser permissions to create this publication, so it is usually preferable to create the publication before starting the connector for the first time. If the publication already exists, either for all tables or configured with a subset of tables, Debezium uses the publication as it is defined.
database.hostname	No default	IP address or hostname of the PostgreSQL database server.

Property	Default	Description
database.port	**5432**	Integer port number of the PostgreSQL database server.
database.user	No default	Name of the PostgreSQL database user for connecting to the PostgreSQL database server.
database.password	No default	Password to use when connecting to the PostgreSQL database server.
database.dbname	No default	The name of the PostgreSQL database from which to stream the changes.
database.server.name	No default	Logical name that identifies and provides a namespace for the particular PostgreSQL database server or cluster in which Debezium is capturing changes. The logical name should be unique across all other connectors, since it is used as a topic name prefix for all Kafka topics that receive records from this connector. Only alphanumeric characters, hyphens, dots and underscores must be used in the database server logical name. + **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value.

Property	Default	Description
schema.include.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **want** to capture changes. Any schema name not included in **schema.include.list** is excluded from having its changes captured. By default, all non-system schemas have their changes captured. Do not also set the **schema.exclude.list** property.
schema.exclude.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **do not** want to capture changes. Any schema whose name is not included in **schema.exclude.list** has its changes captured, with the exception of system schemas. Do not also set the **schema.include.list** property.
table.include.list	No default	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you want to capture. Any table not included in **table.include.list** does not have its changes captured. Each identifier is of the form *schemaName.tableName*. By default, the connector captures changes in every non-system table in each schema whose changes are being captured. Do not also set the **table.exclude.list** property.
table.exclude.list	No default	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you **do not** want to capture. Any table not included in **table.exclude.list** has it changes captured. Each identifier is of the form *schemaName.tableName*. Do not also set the **table.include.list** property.
column.include.list	No default	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns that should be included in change event record values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Do not also set the **column.exclude.list** property.

Property	Default	Description
column.exclude.list	No default	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns that should be excluded from change event record values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Do not also set the **column.include.list** property.
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision: **adaptive** captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type. **adaptive_time_microseconds** captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type. An exception is **TIME** type fields, which are always captured as microseconds. **connect** always represents time and timestamp values by using Kafka Connect's built-in representations for **Time**, **Date**, and **Timestamp**, which use millisecond precision regardless of the database columns' precision. See temporal values.
decimal.handling.mode	**precise**	Specifies how the connector should handle values for **DECIMAL** and **NUMERIC** columns: **precise** represents values by using **java.math.BigDecimal** to represent values in binary form in change events. **double** represents values by using **double** values, which might result in a loss of precision but which is easier to use. **string** encodes values as formatted strings, which are easy to consume but semantic information about the real type is lost. See Decimal types.

Property	Default	Description
hstore.handling.mode	**map**	Specifies how the connector should handle values for **hstore** columns: **map** represents values by using **MAP**. **json** represents values by using **json string**. This setting encodes values as formatted strings such as **{"key" : "val"}**. See PostgreSQL **HSTORE** type.
interval.handling.mode	**numeric**	Specifies how the connector should handle values for **interval** columns: **numeric** represents intervals using approximate number of microseconds. **string** represents intervals exactly by using the string pattern representation **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**. For example: **P1Y2M3DT4H5M6.78S**. See PostgreSQL basic types.
database.sslmode	**disable**	Whether to use an encrypted connection to the PostgreSQL server. Options include: **disable** uses an unencrypted connection. **require** uses a secure (encrypted) connection, and fails if one cannot be established. **verify-ca** behaves like **require** but also verifies the server TLS certificate against the configured Certificate Authority (CA) certificates, or fails if no valid matching CA certificates are found. **verify-full** behaves like **verify-ca** but also verifies that the server certificate matches the host to which the connector is trying to connect. See the PostgreSQL documentation for more information.
database.sslcert	No default	The path to the file that contains the SSL certificate for the client. See the PostgreSQL documentation for more information.
database.sslkey	No default	The path to the file that contains the SSL private key of the client. See the PostgreSQL documentation for more information.

Property	Default	Description
database.sslpassword	No default	The password to access the client private key from the file specified by **database.sslkey**. See the PostgreSQL documentation for more information.
database.sslrootcert	No default	The path to the file that contains the root certificate(s) against which the server is validated. See the PostgreSQL documentation for more information.
database.tcpKeepAlive	**true**	Enable TCP keep-alive probe to verify that the database connection is still alive. See the PostgreSQL documentation for more information.
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. **true** - a delete operation is represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is emitted. After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case log compaction is enabled for the topic.
column.truncate.to.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event records, values in these columns are truncated if they are longer than the number of characters specified by *length* in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, **+column.truncate.to.20.chars**.

Property	Default	Description
column.mask.with.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event values, the values in the specified table columns are replaced with *length* number of asterisk (*) characters. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer or zero. When you specify zero, the connector replaces a value with an empty string.

Property	Default	Description
column.mask.hash.*hashAlgorithm*.with.salt.*salt*; column.mask.hash.v2.*hashAlgorithm*.with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *<schemaName>.<tableName>.<columnName>*. In the resulting change event record, the values for the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. In the following example, **CzQMA0cB5K** is a randomly selected salt. `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts. Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting data set might not be completely masked. Hashing strategy version 2 should be used to ensure fidelity if the value is being hashed in different places or systems.

Property	Default	Description
column.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*, or *databaseName.schemaName.tableName.columnName*. For each specified column, the connector adds the column's original type and original length as parameters to the corresponding field schemas in the emitted change records. The following added schema parameters propagate the original type name and also the original length for variable-width types: **__debezium.source.column.type** + **__debezium.source.column.length** + **__debezium.source.column.scale** This property is useful for properly sizing corresponding columns in sink databases.
datatype.propagate.source.type	*n/a*	An optional, comma-separated list of regular expressions that match the database-specific data type name for some columns. Fully-qualified data type names are of the form *databaseName.tableName.typeName*, or *databaseName.schemaName.tableName.typeName*. For these data types, the connector adds parameters to the corresponding field schemas in emitted change records. The added parameters specify the original type and length of the column: **__debezium.source.column.type** + **__debezium.source.column.length** + **__debezium.source.column.scale** These parameters propagate a column's original type name and length, for variable-width types, respectively. This property is useful for properly sizing corresponding columns in sink databases. See the list of PostgreSQL-specific data type names.

Property	Default	Description
message.key.columns	*empty string*	A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables. By default, Debezium uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns. To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format: ***<fully-qualified_tableName>:<keyColumn>,<keyColumn>*** To base a table key on multiple column names, insert commas between the column names. Each fully-qualified table name is a regular expression in the following format: ***<schemaName>.<tableName>*** The property can include entries for multiple tables. Use a semicolon to separate table entries in the list. The following example sets the message key for the tables **inventory.customers** and **purchase.orders**: **inventory.customers:pk1,pk2;(.*).purchaseorders:pk3,pk4** For the table **inventory.customer**, the columns **pk1** and **pk2** are specified as the message key. For the **purchaseorders** tables in any schema, the columns **pk3** and **pk4** server as the message key. There is no limit to the number of columns that you use to create custom message keys. However, it's best to use the minimum number that are required to specify a unique key.

Property	Default	Description
publication.autocreate.mode	*all_tables*	Applies only when streaming changes by using the **pgoutput** plug-in. The setting determines how creation of a publication should work. Possible settings are: **all_tables** - If a publication exists, the connector uses it. If a publication does not exist, the connector creates a publication for all tables in the database for which the connector is capturing changes. This requires that the database user that has permission to perform replications also has permission to create a publication. This is granted with **CREATE PUBLICATION <publication_name> FOR ALL TABLES;**. **disabled** - The connector does not attempt to create a publication. A database administrator or the user configured to perform replications must have created the publication before running the connector. If the connector cannot find the publication, the connector throws an exception and stops. **filtered** - If a publication exists, the connector uses it. If no publication exists, the connector creates a new publication for tables that match the current filter configuration as specified by the **database.exclude.list**, **schema.include.list**, **schema.exclude.list**, and **table.include.list** connector configuration properties. For example: **CREATE PUBLICATION <publication_name> FOR TABLE <tbl1, tbl2, tbl3>**.
binary.handling.mode	bytes	Specifies how binary (**bytea**) columns should be represented in change events: **bytes** represents binary data as byte array. **base64** represents binary data as base64-encoded strings. **hex** represents binary data as hex-encoded (base16) strings.

Property	Default	Description
schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.
truncate.handling.mode	skip	Specifies how whether **TRUNCATE** events should be propagated or not (only available when using the **pgoutput** plug-in with Postgres 11 or later): **skip** causes those event to be omitted (the default). **include** causes those events to be included. For information about the structure of *truncate* events and about their ordering semantics, see]. This option is deprecated. Use xref:postgresql-property-skipped-operations[**skipped.operations** in its place.
money.fraction.digits	**2**	Specifies how many decimal digits should be used when converting Postgres **money** type to **java.math.BigDecimal**, which represents the values in change events. Applicable only when **decimal.handling.mode** is set to **precise**.
message.prefix.include.list	No default	An optional, comma-separated list of regular expressions that match names of logical decoding message prefixes for which you **want** to capture. Any logical decoding message with a prefix not included in **message.prefix.include.list** is excluded By default, all logical decoding messages are captured. Do not also set the **message.prefix.exclude.list** property. For information about the structure of *message* events and about their ordering semantics, see *message* events.

Property	Default	Description
message.prefix.exclude.list	No default	An optional, comma-separated list of regular expressions that match names of logical decoding message prefixes for which you **do not** to capture. Any logical decoding message with a prefix that is not included in **message.prefix.exclude.list** is included. Do not also set the **message.prefix.include.list** property. To exclude all logical decoding messages pass **.*** into this config. For information about the structure of *message* events and about their ordering semantics, see *message* events.

The following *advanced* configuration properties have defaults that work in most situations and therefore rarely need to be specified in the connector's configuration.

Table 7.25. Advanced connector configuration properties

Property	Default	Description

Property	Default	Description

converters	No default	Enumerates a comma-separated list of the symbolic names of the custom converter instances that the connector can use. For example,
		
		isbn
		
		You must set the **converters** property to enable the connector to use a custom converter.
		
		For each converter that you configure for a connector, you must also add a **.type** property, which specifies the fully-qualifed name of the class that implements the converter interface. The **.type** property uses the following format:
		
		***<converterSymbolicName>*.type**
		
		For example,
		
		`isbn.type: io.debezium.test.IsbnConverter`
		
		If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate any additional configuration parameter with a converter, prefix the parameter names with the symbolic name of the converter.
		For example,
		
		`isbn.schema.name: io.debezium.postgresql.type.Isbn`

Property	Default	Description
snapshot.mode	**initial**	Specifies the criteria for performing a snapshot when the connector starts: **initial** - The connector performs a snapshot only when no offsets have been recorded for the logical server name. **always** - The connector performs a snapshot each time the connector starts. **never** - The connector never performs snapshots. When a connector is configured this way, its behavior when it starts is as follows. If there is a previously stored LSN in the Kafka offsets topic, the connector continues streaming changes from that position. If no LSN has been stored, the connector starts streaming changes from the point in time when the PostgreSQL logical replication slot was created on the server. The **never** snapshot mode is useful only when you know all data of interest is still reflected in the WAL. **initial_only** - The connector performs an initial snapshot and then stops, without processing any subsequent changes. **exported** - deprecated For more information, see the table of **snapshot.mode** options.
snapshot.include.collection.list	All tables specified in **table.include.list**	An optional, comma-separated list of regular expressions that match the fully-qualified names (***<schemaName>.<tableName>***) of the tables to include in a snapshot. The specified items must be named in the connector's **table.include.list** property. This property takes effect only if the connector's **snapshot.mode** property is set to a value other than **never**. This property does not affect the behavior of incremental snapshots.

Property	Default	Description
snapshot.lock.timeout.ms	**10000**	Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this time interval, the snapshot fails. How the connector performs snapshots provides details.

Property	Default	Description
snapshot.select.statement.overrides	No default	Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log. The property contains a comma-separated list of fully-qualified table names in the form ***<schemaName>.<tableName>***. For example, **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** For each table in the list, add a further configuration property that specifies the **SELECT** statement for the connector to run on the table when it takes a snapshot. The specified **SELECT** statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this **SELECT** statement property: **snapshot.select.statement.overrides.*<schemaName>.<tableName>***. For example, **snapshot.select.statement.overrides.customers.orders**. Example: From a **customers.orders** table that includes the soft-delete column, **delete_flag**, add the following properties if you want a snapshot to include only those records that are not soft-deleted: "snapshot.select.statement.overrides" : "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" In the resulting snapshot, the connector includes only the records for which **delete_flag = 0**.

Property	Default	Description
event.processing.failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events: **fail** propagates the exception, indicates the offset of the problematic event, and causes the connector to stop. **warn** logs the offset of the problematic event, skips that event, and continues processing. **skip** skips the problematic event and continues processing.
max.batch.size	**10240**	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
max.queue.size	**20240**	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.

Property	Default	Description
max.queue.size.in.bytes	**0**	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait for new change events to appear before it starts processing a batch of events. Defaults to 1000 milliseconds, or 1 second.

Property	Default	Description
include.unknown.datatypes	**false**	Specifies connector behavior when the connector encounters a field whose data type is unknown. The default behavior is that the connector omits the field from the change event and logs a warning. Set this property to **true** if you want the change event to contain an opaque binary representation of the field. This lets consumers decode the field. You can control the exact representation by setting the **binary handling mode** property. **NOTE** Consumers risk backward compatibility issues when **include.unknown.datatypes** is set to **true**. Not only may the database-specific binary representation change between releases, but if the data type is eventually supported by Debezium, the data type will be sent downstream in a logical type, which would require adjustments by consumers. In general, when encountering unsupported data types, create a feature request so that support can be added.
database.initial.statements	No default	A semicolon separated list of SQL statements that the connector executes when it establishes a JDBC connection to the database. To use a semicolon as a character and not as a delimiter, specify two consecutive semicolons, **;;**. The connector may establish JDBC connections at its own discretion. Consequently, this property is useful for configuration of session parameters only, and not for executing DML statements. The connector does not execute these statements when it creates a connection for reading the transaction log.

Property	Default	Description
status.update.interval.ms	10000	Frequency for sending replication connection status updates to the server, given in milliseconds. The property also controls how frequently the database status is checked to detect a dead connection in case the database was shut down.
heartbeat.interval.ms	0	Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages. Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are needed when there are many updates in a database that is being tracked but only a tiny number of updates are related to the table(s) and schema(s) for which the connector is capturing changes. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that no offset updates are committed to Kafka and the connector does not have an opportunity to send the latest retrieved LSN to the database. The database retains WAL files that contain events that have already been processed by the connector. Sending heartbeat messages enables the connector to send the latest retrieved LSN to the database, which allows the database to reclaim disk space being used by no longer needed WAL files.

Property	Default	Description
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern: *<heartbeat.topics.prefix>.<server.name>* For example, if the database server name is **fulfillment**, the default topic name is **__debezium-heartbeat.fulfillment**.
heartbeat.action.query	No default	Specifies a query that the connector executes on the source database when the connector sends a heartbeat message. This is useful for resolving the situation described in WAL disk space consumption, where capturing changes from a low-traffic database on the same host as a high-traffic database prevents Debezium from processing WAL records and thus acknowledging WAL positions with the database. To address this situation, create a heartbeat table in the low-traffic database, and set this property to a statement that inserts records into that table, for example: **INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')** This allows the connector to receive changes from the low-traffic database and acknowledge their LSNs, which prevents unbounded WAL growth on the database host.

Property	Default	Description
schema.refresh.mode	**columns_diff**	Specify the conditions that trigger a refresh of the in-memory schema for a table. **columns_diff** is the safest mode. It ensures that the in-memory schema stays in sync with the database table's schema at all times. **columns_diff_exclude_unchanged_toast** instructs the connector to refresh the in-memory schema cache if there is a discrepancy with the schema derived from the incoming message, unless unchanged TOASTable data fully accounts for the discrepancy. This setting can significantly improve connector performance if there are frequently-updated tables that have TOASTed data that are rarely part of updates. However, it is possible for the in-memory schema to become outdated if TOASTable columns are dropped from the table.
snapshot.delay.ms	No default	An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
snapshot.fetch.size	**10240**	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
slot.stream.params	No default	Semicolon separated list of parameters to pass to the configured logical decoding plug-in. For example, **add-tables=public.table,public.table2;include-lsn=true**.

Property	Default	Description
sanitize.field.names	**true** if connector configuration sets the **key.converter** or **value.converter** property to the Avro converter. **false** if not.	Indicates whether field names are sanitized to adhere to Avro naming requirements.
slot.max.retries	**6**	If connecting to a replication slot fails, this is the maximum number of consecutive attempts to connect.
slot.retry.delay.ms	**10000** (10 seconds)	The number of milliseconds to wait between retry attempts when the connector fails to connect to a replication slot.
toasted.value.placeholder	**__debezium_unavailable_value**	Specifies the constant that the connector provides to indicate that the original value is a toasted value that is not provided by the database. If the setting of **toasted.value.placeholder** starts with the **hex:** prefix it is expected that the rest of the string represents hexadecimally encoded octets. See toasted values for additional details. *This option is deprecated, please use **unavailable.value.placeholder** instead.*
unavailable.value.placeholder	**__debezium_unavailable_value**	Specifies the constant that the connector provides to indicate that the original value is a toasted value that is not provided by the database. If the setting of **unavailable.value.placeholder** starts with the **hex:** prefix it is expected that the rest of the string represents hexadecimally encoded octets. See toasted values for additional details.
provide.transaction.metadata	**false**	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify **true** if you want the connector to do this. See Transaction metadata for details.

Property	Default	Description
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name; defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**.
retriable.restart.connector.wait.ms	10000 (10 seconds)	The number of milliseconds to wait before restarting a connector after a retriable error occurs.
skipped.operations	**t**	A comma-separated list of operation types that will be skipped during streaming. The operations include: **c** for inserts/create, **u** for updates, **d** for deletes, **t** for truncates, and **none** to not skip any operations. By default, truncate operations are skipped.
signal.data.collection	No default value	Fully-qualified name of the data collection that is used to send signals to the connector. Use the following format to specify the collection name: ***<schemaName>.<tableName>***
incremental.snapshot.chunk.size	1024	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.
xmin.fetch.interval.ms	**0**	How often, in milliseconds, the XMIN will be read from the replication slot. The XMIN value provides the lower bounds of where a new replication slot could start from. The default value of **0** disables tracking XMIN tracking.

Pass-through connector configuration properties

The connector also supports *pass-through* configuration properties that are used when creating the Kafka producer and consumer.

Be sure to consult the Kafka documentation for all of the configuration properties for Kafka producers and consumers. The PostgreSQL connector does use the new consumer configuration properties.

7.7. MONITORING DEBEZIUM POSTGRESQL CONNECTOR PERFORMANCE

The Debezium PostgreSQL connector provides two types of metrics that are in addition to the built-in support for JMX metrics that Zookeeper, Kafka, and Kafka Connect provide.

- Snapshot metrics provide information about connector operation while performing a snapshot.
- Streaming metrics provide information about connector operation when the connector is capturing changes and streaming change event records.

Debezium monitoring documentation provides details for how to expose these metrics by using JMX.

7.7.1. Monitoring Debezium during snapshots of PostgreSQL databases

The **MBean** is **debezium.postgres:type=connector-metrics,context=snapshot,server=*<postgresql.server.name>***. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.

Attributes	Type	Description
QueueTotalCapacity	int	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	int	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	int	The total number of tables that are being included in the snapshot.
RemainingTableCount	int	The number of tables that the snapshot has yet to copy.
SnapshotRunning	boolean	Whether the snapshot was started.
SnapshotAborted	boolean	Whether the snapshot was aborted.
SnapshotCompleted	boolean	Whether the snapshot completed.
SnapshotDurationInSeconds	long	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	Map<String, Long>	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The connector also provides the following additional snapshot metrics when an incremental snapshot is executed:

Attributes	Type	Description
ChunkId	string	The identifier of the current snapshot chunk.
ChunkFrom	string	The lower bound of the primary key set defining the current chunk.
ChunkTo	string	The upper bound of the primary key set defining the current chunk.
TableFrom	string	The lower bound of the primary key set of the currently snapshotted table.
TableTo	string	The upper bound of the primary key set of the currently snapshotted table.

7.7.2. Monitoring Debezium PostgreSQL connector record streaming

The MBean is **debezium.postgres:type=connector-metrics,context=streaming,server=*<postgresql.server.name>***. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	string	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	long	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	long	The total number of events that this connector has seen since the last start or metrics reset.
TotalNumberOfCreateEventsSeen	long	The total number of create events that this connector has seen since the last start or metrics reset.

Attributes	Type	Description
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.

Attributes	Type	Description
NumberOfCommittedTransactions	long	The number of processed transactions that were committed.
SourceEventPosition	Map<String, String>	The coordinates of the last received event.
LastTransactionId	string	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

7.8. HOW DEBEZIUM POSTGRESQL CONNECTORS HANDLE FAULTS AND PROBLEMS

Debezium is a distributed system that captures all changes in multiple upstream databases; it never misses or loses an event. When the system is operating normally or being managed carefully then Debezium provides *exactly once* delivery of every change event record.

If a fault does happen then the system does not lose any events. However, while it is recovering from the fault, it might repeat some change events. In these abnormal situations, Debezium, like Kafka, provides *at least once* delivery of change events.

Details are in the following sections:

- Configuration and startup errors
- PostgreSQL becomes unavailable
- Cluster failures
- Kafka Connect process stops gracefully
- Kafka Connect process crashes
- Kafka becomes unavailable
- Connector is stopped for a duration

Configuration and startup errors

In the following situations, the connector fails when trying to start, reports an error/exception in the log, and stops running:

- The connector's configuration is invalid.
- The connector cannot successfully connect to PostgreSQL by using the specified connection parameters.
- The connector is restarting from a previously-recorded position in the PostgreSQL WAL (by using the LSN) and PostgreSQL no longer has that history available.

In these cases, the error message has details about the problem and possibly a suggested workaround. After you correct the configuration or address the PostgreSQL problem, restart the connector.

PostgreSQL becomes unavailable

When the connector is running, the PostgreSQL server that it is connected to could become unavailable for any number of reasons. If this happens, the connector fails with an error and stops. When the server is available again, restart the connector.

The PostgreSQL connector externally stores the last processed offset in the form of a PostgreSQL LSN. After a connector restarts and connects to a server instance, the connector communicates with the server to continue streaming from that particular offset. This offset is available as long as the Debezium replication slot remains intact. Never drop a replication slot on the primary server or you will lose data. See the next section for failure cases in which a slot has been removed.

Cluster failures

As of release 12, PostgreSQL allows logical replication slots *only on primary servers*. This means that you can point a Debezium PostgreSQL connector to only the active primary server of a database cluster. Also, replication slots themselves are not propagated to replicas. If the primary server goes down, a new primary must be promoted.

The new primary must have a replication slot that is configured for use by the **pgoutput** plug-in and the database in which you want to capture changes. Only then can you point the connector to the new server and restart the connector.

There are important caveats when failovers occur and you should pause Debezium until you can verify that you have an intact replication slot that has not lost data. After a failover:

- There must be a process that re-creates the Debezium replication slot before allowing the application to write to the **new** primary. This is crucial. Without this process, your application can miss change events.
- You might need to verify that Debezium was able to read all changes in the slot **before the old primary failed**.

One reliable method of recovering and verifying whether any changes were lost is to recover a backup of the failed primary to the point immediately before it failed. While this can be administratively difficult, it allows you to inspect the replication slot for any unconsumed changes.

Kafka Connect process stops gracefully

Suppose that Kafka Connect is being run in distributed mode and a Kafka Connect process is stopped gracefully. Prior to shutting down that process, Kafka Connect migrates the process's connector tasks to another Kafka Connect process in that group. The new connector tasks start processing exactly where the prior tasks stopped. There is a short delay in processing while the connector tasks are stopped gracefully and restarted on the new processes.

Kafka Connect process crashes

If the Kafka Connector process stops unexpectedly, any connector tasks it was running terminate without recording their most recently processed offsets. When Kafka Connect is being run in distributed mode, Kafka Connect restarts those connector tasks on other processes. However, PostgreSQL connectors resume from the last offset that was *recorded* by the earlier processes. This means that the new replacement tasks might generate some of the same change events that were processed just prior to the crash. The number of duplicate events depends on the offset flush period and the volume of data changes just before the crash.

Because there is a chance that some events might be duplicated during a recovery from failure, consumers should always anticipate some duplicate events. Debezium changes are idempotent, so a sequence of events always results in the same state.

In each change event record, Debezium connectors insert source-specific information about the origin of the event, including the PostgreSQL server's time of the event, the ID of the server transaction, and the position in the write-ahead log where the transaction changes were written. Consumers can keep track of this information, especially the LSN, to determine whether an event is a duplicate.

Kafka becomes unavailable

As the connector generates change events, the Kafka Connect framework records those events in Kafka by using the Kafka producer API. Periodically, at a frequency that you specify in the Kafka Connect configuration, Kafka Connect records the latest offset that appears in those change events. If the Kafka brokers become unavailable, the Kafka Connect process that is running the connectors repeatedly tries to reconnect to the Kafka brokers. In other words, the connector tasks pause until a connection can be re-established, at which point the connectors resume exactly where they left off.

Connector is stopped for a duration

If the connector is gracefully stopped, the database can continue to be used. Any changes are recorded in the PostgreSQL WAL. When the connector restarts, it resumes streaming changes where it left off. That is, it generates change event records for all database changes that were made while the connector was stopped.

A properly configured Kafka cluster is able to handle massive throughput. Kafka Connect is written according to Kafka best practices, and given enough resources a Kafka Connect connector can also handle very large numbers of database change events. Because of this, after being stopped for a while, when a Debezium connector restarts, it is very likely to catch up with the database changes that were made while it was stopped. How quickly this happens depends on the capabilities and performance of Kafka and the volume of changes being made to the data in PostgreSQL.

CHAPTER 8. DEBEZIUM CONNECTOR FOR SQL SERVER

The Debezium SQL Server connector captures row-level changes that occur in the schemas of a SQL Server database.

For information about the SQL Server versions that are compatible with this connector, see the Debezium Supported Configurations page.

For details about the Debezium SQL Server connector and its use, see following topics:

- Section 8.1, "Overview of Debezium SQL Server connector"
- Section 8.2, "How Debezium SQL Server connectors work"
- Section 8.2.6, "Descriptions of Debezium SQL Server connector data change events"
- Section 8.2.8, "How Debezium SQL Server connectors map data types"
- Section 8.3, "Setting up SQL Server to run a Debezium connector"
- Section 8.4, "Deployment of Debezium SQL Server connectors"
- Section 8.5, "Refreshing capture tables after a schema change"
- Section 8.6, "Monitoring Debezium SQL Server connector performance"

The first time that the Debezium SQL Server connector connects to a SQL Server database or cluster, it takes a consistent snapshot of the schemas in the database. After the initial snapshot is complete, the connector continuously captures row-level changes for **INSERT**, **UPDATE**, or **DELETE** operations that are committed to the SQL Server databases that are enabled for CDC. The connector produces events for each data change operation, and streams them to Kafka topics. The connector streams all of the events for a table to a dedicated Kafka topic. Applications and services can then consume data change event records from that topic.

8.1. OVERVIEW OF DEBEZIUM SQL SERVER CONNECTOR

The Debezium SQL Server connector is based on the change data capture feature that is available in SQL Server 2016 Service Pack 1 (SP1) and later Standard edition or Enterprise edition. The SQL Server capture process monitors designated databases and tables, and stores the changes into specifically created *change tables* that have stored procedure facades.

To enable the Debezium SQL Server connector to capture change event records for database operations, you must first enable change data capture on the SQL Server database. CDC must be enabled on both the database and on each table that you want to capture. After you set up CDC on the source database, the connector can capture row-level **INSERT**, **UPDATE**, and **DELETE** operations that occur in the database. The connector writes event records for each source table to a Kafka topic especially dedicated to that table. One topic exists for each captured table. Client applications read the Kafka topics for the database tables that they follow, and can respond to the row-level events they consume from those topics.

The first time that the connector connects to a SQL Server database or cluster, it takes a consistent snapshot of the schemas for all tables for which it is configured to capture changes, and streams this state to Kafka. After the snapshot is complete, the connector continuously captures subsequent row-level changes that occur. By first establishing a consistent view of all of the data, the connector can continue reading without having lost any of the changes that were made while the snapshot was taking place.

The Debezium SQL Server connector is tolerant of failures. As the connector reads changes and produces events, it periodically records the position of events in the database log (*LSN / Log Sequence Number*). If the connector stops for any reason (including communication failures, network problems, or crashes), after a restart the connector resumes reading the SQL Server *CDC* tables from the last point that it read.

NOTE

Offsets are committed periodically. They are not committed at the time that a change event occurs. As a result, following an outage, duplicate events might be generated.

Fault tolerance also applies to snapshots. That is, if the connector stops during a snapshot, the connector begins a new snapshot when it restarts.

8.2. HOW DEBEZIUM SQL SERVER CONNECTORS WORK

To optimally configure and run a Debezium SQL Server connector, it is helpful to understand how the connector performs snapshots, streams change events, determines Kafka topic names, and uses metadata.

For details about how the connector works, see the following sections:

- Section 8.2.1, "How Debezium SQL Server connectors perform database snapshots"
- Section 8.2.2, "How Debezium SQL Server connectors read change data tables"
- Section 8.2.4, "Default names of Kafka topics that receive Debezium SQL Server change event records"
- Section 8.2.5, "How the Debezium SQL Server connector uses the schema change topic"
- Section 8.2.6, "Descriptions of Debezium SQL Server connector data change events"
- Section 8.2.7, "Debezium SQL Server connector-generated events that represent transaction boundaries"

8.2.1. How Debezium SQL Server connectors perform database snapshots

SQL Server CDC is not designed to store a complete history of database changes. For the Debezium SQL Server connector to establish a baseline for the current state of the database, it uses a process called *snapshotting*.

You can configure how the connector creates snapshots. By default, the connector's snapshot mode is set to **initial**. Based on this **initial** snapshot mode, the first time that the connector starts, it performs an initial *consistent snapshot* of the database. This initial snapshot captures the structure and data for any tables that match the criteria defined by the **include** and **exclude** properties that are configured for the connector (for example, **table.include.list**, **column.include.list**, **table.exclude.list**, and so forth).

When the connector creates a snapshot, it completes the following tasks:

1. Determines the tables to be captured.
2. Obtains a lock on the SQL Server tables for which CDC is enabled to prevent structural changes from occurring during creation of the snapshot. The level of the lock is determined by **snapshot.isolation.mode** configuration option.

3. Reads the maximum log sequence number (LSN) position in the server's transaction log.
4. Captures the structure of all relevant tables.
5. Releases the locks obtained in Step 2, if necessary. In most cases, locks are held for only a short period of time.
6. Scans the SQL Server source tables and schemas to be captured based on the LSN position that was read in Step 3, generates a **READ** event for each row in the table, and writes the events to the Kafka topic for the table.
7. Records the successful completion of the snapshot in the connector offsets.

The resulting initial snapshot captures the current state of each row in the tables that are enabled for CDC. From this baseline state, the connector captures subsequent changes as they occur.

8.2.1.1. Ad hoc snapshots

By default, a connector runs an initial snapshot operation only after it starts for the first time. Following this initial snapshot, under normal circumstances, the connector does not repeat the snapshot process. Any future change event data that the connector captures comes in through the streaming process only.

However, in some situations the data that the connector obtained during the initial snapshot might become stale, lost, or incomplete. To provide a mechanism for recapturing table data, Debezium includes an option to perform ad hoc snapshots. The following changes in a database might be cause for performing an ad hoc snapshot:

- The connector configuration is modified to capture a different set of tables.
- Kafka topics are deleted and must be rebuilt.
- Data corruption occurs due to a configuration error or some other problem.

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of signaling tables. You initiate an ad hoc snapshot by sending a signal request to the Debezium signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, Debezium can create a topic automatically if automatic topic creation is enabled.

Ad hoc snapshot signals specify the tables to include in the snapshot. The snapshot can capture the entire contents of the database, or capture only a subset of the tables in the database.

You specify the tables to capture by sending an **execute-snapshot** message to the signaling table. Set the type of the **execute-snapshot** signal to **incremental**, and provide the names of the tables to include in the snapshot, as described in the following table:

Table 8.1. Example of an ad hoc execute-snapshot signal record

Field	Default	Value
type	**incremental**	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only **incremental** snapshots.

Field	Default	Value
data-collections	*N/A*	An array that contains the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the **signal.data.collection** configuration option.

Triggering an ad hoc snapshot

You initiate an ad hoc snapshot by adding an entry with the **execute-snapshot** signal type to the signaling table. After the connector processes the message, it begins the snapshot operation. The snapshot process reads the first and last primary key values and uses those values as the start and end point for each table. Based on the number of entries in the table, and the configured chunk size, Debezium divides the table into chunks, and proceeds to snapshot each chunk, in succession, one at a time.

Currently, the **execute-snapshot** action type triggers incremental snapshots only. For more information, see Incremental snapshots.

8.2.1.2. Incremental snapshots

To provide flexibility in managing snapshots, Debezium includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the Debezium mechanism for sending signals to a Debezium connector.

In an incremental snapshot, instead of capturing the full state of a database all at once, as in an initial snapshot, Debezium captures each table in phases, in a series of configurable chunks. You can specify the tables that you want the snapshot to capture and the size of each chunk. The chunk size determines the number of rows that the snapshot collects during each fetch operation on the database. The default chunk size for incremental snapshots is 1 KB.

As an incremental snapshot proceeds, Debezium uses watermarks to track its progress, maintaining a record of each table row that it captures. This phased approach to capturing data provides the following advantages over the standard initial snapshot process:

- You can run incremental snapshots in parallel with streamed data capture, instead of postponing streaming until the snapshot completes. The connector continues to capture near real-time events from the change log throughout the snapshot process, and neither operation blocks the other.
- If the progress of an incremental snapshot is interrupted, you can resume it without losing any data. After the process resumes, the snapshot begins at the point where it stopped, rather than recapturing the table from the beginning.
- You can run an incremental snapshot on demand at any time, and repeat the process as needed to adapt to database updates. For example, you might re-run a snapshot after you modify the connector configuration to add a table to its **table.include.list** property.

Incremental snapshot process

When you run an incremental snapshot, Debezium sorts each table by primary key and then splits the table into chunks based on the configured chunk size. Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a **READ** event. That event represents the value of the row when the snapshot for the chunk began.

As a snapshot proceeds, it's likely that other processes continue to access the database, potentially modifying table records. To reflect such changes, **INSERT**, **UPDATE**, or **DELETE** operations are committed to the transaction log as per usual. Similarly, the ongoing Debezium streaming process continues to detect these change events and emits corresponding change event records to Kafka.

How Debezium resolves collisions among records with the same primary key

In some cases, the **UPDATE** or **DELETE** events that the streaming process emits are received out of sequence. That is, the streaming process might emit an event that modifies a table row before the snapshot captures the chunk that contains the **READ** event for that row. When the snapshot eventually emits the corresponding **READ** event for the row, its value is already superseded. To ensure that incremental snapshot events that arrive out of sequence are processed in the correct logical order, Debezium employs a buffering scheme for resolving collisions. Only after collisions between the snapshot events and the streamed events are resolved does Debezium emit an event record to Kafka.

Snapshot window

To assist in resolving collisions between late-arriving **READ** events and streamed events that modify the same table row, Debezium employs a so-called *snapshot window*. The snapshot windows demarcates the interval during which an incremental snapshot captures data for a specified table chunk. Before the snapshot window for a chunk opens, Debezium follows its usual behavior and emits events from the transaction log directly downstream to the target Kafka topic. But from the moment that the snapshot for a particular chunk opens, until it closes, Debezium performs a de-duplication step to resolve collisions between events that have the same primary key..

For each data collection, the Debezium emits two types of events, and stores the records for them both in a single destination Kafka topic. The snapshot records that it captures directly from a table are emitted as **READ** operations. Meanwhile, as users continue to update records in the data collection, and the transaction log is updated to reflect each commit, Debezium emits **UPDATE** or **DELETE** operations for each change.

As the snapshot window opens, and Debezium begins processing a snapshot chunk, it delivers snapshot records to a memory buffer. During the snapshot windows, the primary keys of the **READ** events in the buffer are compared to the primary keys of the incoming streamed events. If no match is found, the streamed event record is sent directly to Kafka. If Debezium detects a match, it discards the buffered **READ** event, and writes the streamed record to the destination topic, because the streamed event logically supersede the static snapshot event. After the snapshot window for the chunk closes, the buffer contains only **READ** events for which no related transaction log events exist. Debezium emits these remaining **READ** events to the table's Kafka topic.

The connector repeats the process for each snapshot chunk.

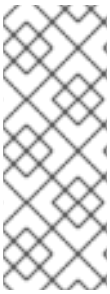
Triggering an incremental snapshot

Currently, the only way to initiate an incremental snapshot is to send an ad hoc snapshot signal to the signaling table on the source database. You submit signals to the table as SQL **INSERT** queries. After Debezium detects the change in the signaling table, it reads the signal, and runs the requested snapshot operation.

The query that you submit specifies the tables to include in the snapshot, and, optionally, specifies the kind of snapshot operation. Currently, the only valid option for snapshots operations is the default value, **incremental**.

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables, for example,
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}

The **data-collections** array for an incremental snapshot signal has no default value. If the **data-collections** array is empty, Debezium detects that no action is required and does not perform a snapshot.



NOTE

If the name of a table that you want to include in a snapshot contains a dot (**.**) in the name of the database, schema, or table, to add the table to the **data-collections** array, you must escape each part of the name in double quotes.

For example, to include a table that exists in the **public** schema and that has the name **My.Table**, use the following format: **"public"."My.Table"**.

Prerequisites

- Signaling is enabled.
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the **signal.data.collection** property.

Procedure

1. Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

```
INSERT INTO _<signalTable>_ (id, type, data) VALUES (_'<id>'_, _'<snapshotType>'_,
'{"data-collections": ["_<tableName>_","_<tableName>_"],"type":"_<snapshotType>_"}');
```

For example,

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-
snapshot', '{"data-collections": ["schema1.table1", "schema2.table2"],"type":"incremental"}');
```

The values of the **id**,**type**, and **data** parameters in the command correspond to the fields of the signaling table.

The following table describes the these parameters:

Table 8.2. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table

Value	Description
myschema.debezium_signal	Specifies the fully-qualified name of the signaling table on the source database
ad-hoc-1	The **id** parameter specifies an arbitrary string that is assigned as the**id** identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. Debezium does not use this string. Rather, during the snapshot, Debezium generates its own **id** string as a watermarking signal.

Value	Description
execute-snapshot	Specifies **type** parameter specifies the operation that the signal is intended to trigger.
data-collections	A required component of the **data** field of a signal that specifies an array of table names to include in the snapshot. The array lists tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the **signal.data.collection** configuration property.
incremental	An optional **type** component of the**data** field of a signal that specifies the kind of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.

The following example, shows the JSON for an incremental snapshot event that is captured by a connector.

Example: Incremental snapshot event message

```
{
    "before":null,
    "after": {
        "pk":"1",
        "value":"New data"
    },
    "source": {
        ...
        "snapshot":"incremental" 1
    },
    "op":"r", 2
    "ts_ms":"1620393591654",
    "transaction":null
}
```

Item	Field name	Description
1	**snapshot**	Specifies the type of snapshot operation to run. Currently, the only valid option is the default value, **incremental**. Specifying a **type** value in the SQL query that you submit to the signaling table is optional. If you do not specify a value, the connector runs an incremental snapshot.
2	**op**	Specifies the event type. The value for snapshot events is **r**, signifying a**READ** operation.

WARNING

The Debezium connector for SQL Server does not support schema changes while an incremental snapshot is running.

8.2.2. How Debezium SQL Server connectors read change data tables

When the connector first starts, it takes a structural snapshot of the structure of the captured tables and persists this information to its internal database history topic. The connector then identifies a change table for each source table, and completes the following steps.

1. For each change table, the connector read all of the changes that were created between the last stored maximum LSN and the current maximum LSN.
2. The connector sorts the changes that it reads in ascending order, based on the values of their commit LSN and change LSN. This sorting order ensures that the changes are replayed by Debezium in the same order in which they occurred in the database.
3. The connector passes the commit and change LSNs as offsets to Kafka Connect.
4. The connector stores the maximum LSN and restarts the process from Step 1.

After a restart, the connector resumes processing from the last offset (commit and change LSNs) that it read.

The connector is able to detect whether CDC is enabled or disabled for included source tables and adjust its behavior.

8.2.3. Limitations of Debezium SQL Server connector

SQL Server specifically requires the base object to be a table in order to create a change capture instance. As consequence, capturing changes from indexed views (aka. materialized views) is not supported by SQL Server and hence Debezium SQL Server connector.

8.2.4. Default names of Kafka topics that receive Debezium SQL Server change event records

By default, the SQL Server connector writes events for all **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table. The connector uses the following convention to name change event topics: ***<serverName>.<schemaName>.<tableName>***

The following list provides definitions for the components of the default name:

serverName

The logical name of the server, as specified by the **database.server.name** configuration property.

schemaName

The name of the database schema in which the change event occurred.

tableName

The name of the database table in which the change event occurred.

For example, if **fulfillment** is the server name, and **dbo** is the schema name, and the database contains tables with the names **products**, **products_on_hand**, **customers**, and **orders**, the connector would stream change event records to the following Kafka topics:

- **fulfillment.dbo.products**
- **fulfillment.dbo.products_on_hand**
- **fulfillment.dbo.customers**
- **fulfillment.dbo.orders**

The connector applies similar naming conventions to label its internal database history topics, schema change topics, and transaction metadata topics.

If the default topic name do not meet your requirements, you can configure custom topic names. To configure custom topic names, you specify regular expressions in the logical topic routing SMT. For more information about using the logical topic routing SMT to customize topic naming, see Topic routing.

8.2.5. How the Debezium SQL Server connector uses the schema change topic

For each table for which CDC is enabled, the Debezium SQL Server connector stores a history of the schema change events that are applied to captured tables in the database. The connector writes schema change events to a Kafka topic named ***<serverName>***, where ***serverName*** is the logical server name that is specified in the **database.server.name** configuration property.

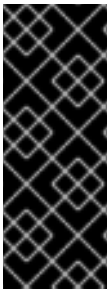
Messages that the connector sends to the schema change topic contain a payload, and, optionally, also contain the schema of the change event message. The payload of a schema change event message includes the following elements:

databaseName

The name of the database to which the statements are applied. The value of **databaseName** serves as the message key.

tableChanges

A structured representation of the entire table schema after the schema change. The **tableChanges** field contains an array that includes entries for each column of the table. Because the structured representation presents data in JSON or Avro format, consumers can easily read messages without first processing them through a DDL parser.



IMPORTANT

When the connector is configured to capture a table, it stores the history of the table's schema changes not only in the schema change topic, but also in an internal database history topic. The internal database history topic is for connector use only and it is not intended for direct use by consuming applications. Ensure that applications that require notifications about schema changes consume that information only from the schema change topic.

WARNING

The format of the messages that a connector emits to its schema change topic is in an incubating state and can change without notice.

Debezium emits a message to the schema change topic when the following events occur:

- You enable CDC for a table.
- You disable CDC for a table.
- You alter the structure of a table for which CDC is enabled by following the schema evolution procedure.

Example: Message emitted to the SQL Server connector schema change topic

The following example shows a message in the schema change topic. The message contains a logical representation of the table schema.

```
{
  "schema": {
  ...
  },
  "payload": {
    "source": {
      "version": "1.9.7.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "databaseName": "testDB", (1)
    "schemaName": "dbo",
    "ddl": null, (2)
    "tableChanges": [ (3)
      {
        "type": "CREATE", (4)
        "id": "\"testDB\".\"dbo\".\"customers\"", (5)
        "table": { (6)
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ (7)
            "id"
          ],
          "columns": [ (8)
            {
```

```
        "name": "id",
        "jdbcType": 4,
        "nativeType": null,
        "typeName": "int identity",
        "typeExpression": "int identity",
        "charsetName": null,
        "length": 10,
        "scale": 0,
        "position": 1,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "first_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 2,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "last_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 3,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "email",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 4,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      }
    ]
```

```
        }
      }
    ]
  }
}
```

Table 8.3. Descriptions of fields in messages emitted to the schema change topic

Item	Field name	Description
1	**databaseName** **schemaName**	Identifies the database and the schema that contain the change.
2	**ddl**	Always **null** for the SQL Server connector. For other connectors, this field contains the DDL responsible for the schema change. This DDL is not available to SQL Server connectors.
3	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.
4	**type**	Describes the kind of change. The value is one of the following: • **CREATE** - table created • **ALTER** - table modified • **DROP** - table deleted
5	**id**	Full identifier of the table that was created, altered, or dropped.
6	**table**	Represents table metadata after the applied change.
7	**primaryKeyColumnNames**	List of columns that compose the table's primary key.
8	**columns**	Metadata for each column in the changed table.

In messages that the connector sends to the schema change topic, the key is the name of the database that contains the schema change. In the following example, the **payload** field contains the key:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "databaseName"
      }
    ],
```

```
    "optional": false,
    "name": "io.debezium.connector.sqlserver.SchemaChangeKey"
  },
  "payload": {
    "databaseName": "testDB"
  }
}
```

8.2.6. Descriptions of Debezium SQL Server connector data change events

The Debezium SQL Server connector generates a data change event for each row-level **INSERT**, **UPDATE**, and **DELETE** operation. Each event contains a key and a value. The structure of the key and the value depends on the table that was changed.

Debezium and Kafka Connect are designed around *continuous streams of event messages*. However, the structure of these events may change over time, which can be difficult for consumers to handle. To address this, each event contains the schema for its content or, if you are using a schema registry, a schema ID that a consumer can use to obtain the schema from the registry. This makes each event self-contained.

The following skeleton JSON shows the basic four parts of a change event. However, how you configure the Kafka Connect converter that you choose to use in your application determines the representation of these four parts in change events. A **schema** field is in a change event only when you configure the converter to produce it. Likewise, the event key and event payload are in a change event only if you configure a converter to produce it. If you use the JSON converter and you configure it to produce all four basic change event parts, change events have this structure:

```
{
 "schema": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

Table 8.4. Overview of change event basic content

Item	Field name	Description

Item	Field name	Description
1	**schema**	The first **schema** field is part of the event key. It specifies a Kafka Connect schema that describes what is in the event key's **payload** portion. In other words, the first **schema** field describes the structure of the primary key, or the unique key if the table does not have a primary key, for the table that was changed. It is possible to override the table's primary key by setting the **message.key.columns** connector configuration property. In this case, the first schema field describes the structure of the key identified by that property.
2	**payload**	The first **payload** field is part of the event key. It has the structure described by the previous **schema** field and it contains the key for the row that was changed.
3	**schema**	The second **schema** field is part of the event value. It specifies the Kafka Connect schema that describes what is in the event value's **payload** portion. In other words, the second **schema** describes the structure of the row that was changed. Typically, this schema contains nested schemas.
4	**payload**	The second **payload** field is part of the event value. It has the structure described by the previous **schema** field and it contains the actual data for the row that was changed.

By default, the connector streams change event records to topics with names that are the same as the event's originating table. See topic names.

WARNING

The SQL Server connector ensures that all Kafka Connect schema names adhere to the Avro schema name format. This means that the logical server name must start with a Latin letter or an underscore, that is, a-z, A-Z, or _. Each remaining character in the logical server name and each character in the database and table names must be a Latin letter, a digit, or an underscore, that is, a-z, A-Z, 0-9, or _. If there is an invalid character it is replaced with an underscore character.

This can lead to unexpected conflicts if the logical server name, a database name, or a table name contains invalid characters, and the only characters that distinguish names from one another are invalid and thus replaced with underscores.

For details about change events, see the following topics:

- Section 8.2.6.1, "About keys in Debezium SQL Server change events"

- Section 8.2.6.2, “About values in Debezium SQL Server change events”

8.2.6.1. About keys in Debezium SQL Server change events

A change event’s key contains the schema for the changed table’s key and the changed row’s actual key. Both the schema and its corresponding payload contain a field for each column in the changed table’s primary key (or unique key constraint) at the time the connector created the event.

Consider the following **customers** table, which is followed by an example of a change event key for this table.

Example table

```
CREATE TABLE customers (
  id INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE
);
```

Example change event key

Every change event that captures a change to the **customers** table has the same event key schema. For as long as the **customers** table has the previous definition, every change event that captures a change to the **customers** table has the following key structure, which in JSON, looks like this:

```
{
   "schema": { 1
      "type": "struct",
      "fields": [ 2
         {
            "type": "int32",
            "optional": false,
            "field": "id"
         }
      ],
      "optional": false, 3
      "name": "server1.dbo.customers.Key" 4
   },
   "payload": { 5
      "id": 1004
   }
}
```

Table 8.5. Description of change event key

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s **payload** portion.

Item	Field name	Description
2	**fields**	Specifies each field that is expected in the **payload**, including each field's name, type, and whether it is required. In this example, there is one required field named **id** of type **int32**.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a table does not have a primary key.
4	**server1.dbo.customers.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format *connector-name.database-schema-name.table-name*.**Key**. In this example: • **server1** is the name of the connector that generated this event. • **dbo** is the database schema for the table that was changed. • **customers** is the table that was updated.
5	**payload**	Contains the key for the row for which this change event was generated. In this example, the key, contains a single **id** field whose value is **1004**.

8.2.6.2. About values in Debezium SQL Server change events

The value in a change event is a bit more complicated than the key. Like the key, the value has a **schema** section and a **payload** section. The **schema** section contains the schema that describes the **Envelope** structure of the **payload** section, including its nested fields. Change events for operations that create, update or delete data all have a value payload with an envelope structure.

Consider the same sample table that was used to show an example of a change event key:

```
CREATE TABLE customers (
  id INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE
);
```

The value portion of a change event for a change to this table is described for each event type.

- *create* events
- *update* events
- *delete* events

create events

The following example shows the value portion of a change event that the connector generates for an operation that creates data in the **customers** table:

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "server1.dbo.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
```

```
    "optional": true,
    "name": "server1.dbo.customers.Value",
    "field": "after"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "version"
      },
      {
        "type": "string",
        "optional": false,
        "field": "connector"
      },
      {
        "type": "string",
        "optional": false,
        "field": "name"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "ts_ms"
      },
      {
        "type": "boolean",
        "optional": true,
        "default": false,
        "field": "snapshot"
      },
      {
        "type": "string",
        "optional": false,
        "field": "db"
      },
      {
        "type": "string",
        "optional": false,
        "field": "schema"
      },
      {
        "type": "string",
        "optional": false,
        "field": "table"
      },
      {
        "type": "string",
        "optional": true,
        "field": "change_lsn"
      },
      {
        "type": "string",
        "optional": true,
```

```
            "field": "commit_lsn"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "event_serial_no"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.sqlserver.Source", 3
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "server1.dbo.customers.Envelope" 4
  },
  "payload": { 5
    "before": null, 6
    "after": { 7
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "source": { 8
      "version": "1.9.7.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
      "event_serial_no": "1"
    },
    "op": "c", 9
    "ts_ms": 1559729471739 10
  }
}
```


Table 8.6. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **server1.dbo.customers.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.database-schemaName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.sqlserver.Source** is the schema for the payload's **source** field. This schema is specific to the SQL Server connector. The connector uses it for all events that it generates.
4	**name**	**server1.dbo.customers.Envelope** is the schema for the overall structure of the payload, where **server1** is the connector name, **dbo** is the database schema name, and **customers** is the table.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content.
7	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **id**, **first_name**, **last_name**, and **email** columns.

Item	Field name	Description
8	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: • Debezium version • Connector type and name • Database and schema names • Timestamp for when the change was made in the database • If the event was part of a snapshot • Name of the table that contains the new row • Server log offsets
9	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots)
10	**ts_ms**	Optional field that displays the time at which the connector processed the event. In the event message envelope, the time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time when a change was committed in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

update events

The value of a change event for an update in the sample **customers** table has the same schema as a *create* event for that table. Likewise, the event value's payload has the same structure. However, the event value payload contains different values in an *update* event. Here is an example of a change event value in an event that the connector generates for an update in the **customers** table:

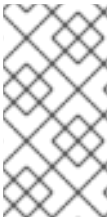
```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1005,
```

```
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "after": { (2)
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "source": { (3)
      "version": "1.9.7.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
      "event_serial_no": "2"
    },
    "op": "u", (4)
    "ts_ms": 1559729998706 (5)
  }
}
```

Table 8.7. Descriptions of *update* event value fields

Item	Field name	Description
1	**before**	An optional field that specifies the state of the row before the event occurred. In an *update* event value, the **before** field contains a field for each table column and the value that was in that column before the database commit. In this example, the **email** value is **john.doe@example.org.**
2	**after**	An optional field that specifies the state of the row after the event occurred. You can compare the **before** and **after** structures to determine what the update to this row was. In the example, the **email** value is now **noreply@example.org**.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different, for example, the sample *update* event has a different offset. The source metadata includes: • Debezium version • Connector type and name • Database and schema names • Timestamp for when the change was made in the database • If the event was part of a snapshot • Name of the table that contains the new row • Server log offsets The **event_serial_no** field differentiates events that have the same commit and change LSN. Typical situations for when this field has a value other than **1**: • *update* events have the value set to **2** because the update generates two events in the CDC change table of SQL Server (see the source documentation for details). The first event contains the old values and the second contains contains new values. The connector uses values in the first event to create the second event. The connector drops the first event. • When a primary key is updated SQL Server emits two evemts. A *delete* event for the removal of the record with the old primary key value and a *create* event for the addition of the record with the new primary key. Both operations share the same commit and change LSN and their event numbers are **1** and **2**, respectively.
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. In the event message envelope, the time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time when the change was committed to the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.



NOTE

Updating the columns for a row's primary/unique key changes the value of the row's key. When a key changes, Debezium outputs *three* events: a *delete* event and a tombstone event with the old key for the row, followed by a *create* event with the new key for the row.

delete events

The value in a *delete* change event has the same **schema** portion as *create* and *update* events for the same table. The **payload** portion in a *delete* event for the sample **customers** table looks like this:

```
{
  "schema": { ... },
  },
  "payload": {
    "before": { <>
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "after": null, (1)
    "source": { (2)
      "version": "1.9.7.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559730445243,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000db0:0005",
      "commit_lsn": "00000027:00000db0:0007",
      "event_serial_no": "1"
    },
    "op": "d", (3)
    "ts_ms": 1559730450205 (4)
  }
}
```

Table 8.8. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit.
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and **pos** field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector type and name • Database and schema names • Timestamp for when the change was made in the database • If the event was part of a snapshot • Name of the table that contains the new row • Server log offsets
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. In the event message envelope, the time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

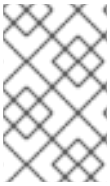
SQL Server connector events are designed to work with Kafka log compaction. Log compaction enables removal of some older messages as long as at least the most recent message for every key is kept. This lets Kafka reclaim storage space while ensuring that the topic contains a complete data set and can be used for reloading key-based state.

Tombstone events

When a row is deleted, the *delete* event value still works with log compaction, because Kafka can remove all earlier messages that have that same key. However, for Kafka to remove all messages that have that same key, the message value must be **null**. To make this possible, after Debezium's SQL Server connector emits a *delete* event, the connector emits a special tombstone event that has the same key but a **null** value.

8.2.7. Debezium SQL Server connector-generated events that represent transaction boundaries

Debezium can generate events that represent transaction boundaries and that enrich data change event messages.



LIMITS ON WHEN DEBEZIUM RECEIVES TRANSACTION METADATA

Debezium registers and receives metadata only for transactions that occur after you deploy the connector. Metadata for transactions that occur before you deploy the connector is not available.

Database transactions are represented by a statement block that is enclosed between the **BEGIN** and **END** keywords. Debezium generates transaction boundary events for the **BEGIN** and **END** delimiters in every transaction. Transaction boundary events contain the following fields:

status

BEGIN or **END**

id

String representation of unique transaction identifier.

event_count (for **END** events)

Total number of events emitted by the transaction.

data_collections (for **END** events)

An array of pairs of **data_collection** and **event_count** that provides the number of events emitted by changes originating from given data collection.

WARNING

There is no way for Debezium to reliably identify when a transaction has ended. The transaction **END** marker is thus emitted only after the first event of another transaction arrives. This can lead to the delayed delivery of **END** marker in case of a low-traffic system.

The following example shows a typical transaction boundary message:

Example: SQL Server connector transaction boundary event

```
{
  "status": "BEGIN",
  "id": "00000025:00000d08:0025",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "00000025:00000d08:0025",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "testDB.dbo.tablea",
      "event_count": 1
    },
    {
```

```
      "data_collection": "testDB.dbo.tableb",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **transaction.topic** option, transaction events are written to the topic named ***database.server.name*.transaction**.

8.2.7.1. Change data event enrichment

When transaction metadata is enabled, the data message **Envelope** is enriched with a new **transaction** field. This field provides information about every event in the form of a composite of fields:

id

String representation of unique transaction identifier

total_order

The absolute position of the event among all events generated by the transaction

data_collection_order

The per-data collection position of the event among all events that were emitted by the transaction

The following example shows what a typical message looks like:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "00000025:00000d08:0025",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

8.2.8. How Debezium SQL Server connectors map data types

The Debezium SQL Server connector represents changes to table row data by producing events that are structured like the table in which the row exists. Each event contains fields to represent the column values for the row. The way in which an event represents the column values for an operation depends on the SQL data type of the column. In the event, the connector maps the fields for each SQL Server data type to both a *literal type* and a *semantic type*.

The connector can map SQL Server data types to both *literal* and *semantic* types.

Literal type

Describes how the value is literally represented by using Kafka Connect schema types, namely **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**.

Semantic type

Describes how the Kafka Connect schema captures the *meaning* of the field using the name of the Kafka Connect schema for the field.

If the default data type conversions do not meet your needs, you can create a custom converter for the connector.

For more information about data type mappings, see the following sections:

- Basic types
- Temporal values
- Decimal values
- Timestamp values

Basic types

The following table shows how the connector maps basic SQL Server data types.

Table 8.9. Data type mappings used by the SQL Server connector

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
BIT	**BOOLEAN**	n/a
TINYINT	**INT16**	n/a
SMALLINT	**INT16**	n/a
INT	**INT32**	n/a
BIGINT	**INT64**	n/a
REAL	**FLOAT32**	n/a
FLOAT[(N)]	**FLOAT64**	n/a
CHAR[(N)]	**STRING**	n/a
VARCHAR[(N)]	**STRING**	n/a
TEXT	**STRING**	n/a
NCHAR[(N)]	**STRING**	n/a
NVARCHAR[(N)]	**STRING**	n/a

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
NTEXT	**STRING**	n/a
XML	**STRING**	**io.debezium.data.Xml** Contains the string representation of an XML document
DATETIMEOFFSET[(P)]	**STRING**	**io.debezium.time.ZonedTimestamp** A string representation of a timestamp with timezone information, where the timezone is GMT

Other data type mappings are described in the following sections.

If present, a column's default value is propagated to the corresponding field's Kafka Connect schema. Change messages will contain the field's default value (unless an explicit column value had been given), so there should rarely be the need to obtain the default value from the schema.

Temporal values

Other than SQL Server's **DATETIMEOFFSET** data type (which contain time zone information), the other temporal types depend on the value of the **time.precision.mode** configuration property. When the **time.precision.mode** configuration property is set to **adaptive** (the default), then the connector will determine the literal type and semantic type for the temporal types based on the column's data type definition so that events *exactly* represent the values in the database:

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**io.debezium.time.Date** Represents the number of days since the epoch.
TIME(0), **TIME(1)**, **TIME(2)**, **TIME(3)**	**INT32**	**io.debezium.time.Time** Represents the number of milliseconds past midnight, and does not include timezone information.
TIME(4), **TIME(5)**, **TIME(6)**	**INT64**	**io.debezium.time.MicroTime** Represents the number of microseconds past midnight, and does not include timezone information.

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIME(7)	**INT64**	**io.debezium.time.NanoTime** Represents the number of nanoseconds past midnight, and does not include timezone information.
DATETIME	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds past the epoch, and does not include timezone information.
SMALLDATETIME	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds past the epoch, and does not include timezone information.
DATETIME2(0), **DATETIME2(1)**, **DATETIME2(2)**, **DATETIME2(3)**	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds past the epoch, and does not include timezone information.
DATETIME2(4), **DATETIME2(5)**, **DATETIME2(6)**	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds past the epoch, and does not include timezone information.
DATETIME2(7)	**INT64**	**io.debezium.time.NanoTimestamp** Represents the number of nanoseconds past the epoch, and does not include timezone information.

When the **time.precision.mode** configuration property is set to **connect**, then the connector will use the predefined Kafka Connect logical types. This may be useful when consumers only know about the built-in Kafka Connect logical types and are unable to handle variable-precision time values. On the other hand, since SQL Server supports tenth of microsecond precision, the events generated by a connector with the **connect** time precision mode will **result in a loss of precision** when the database column has a *fractional second precision* value greater than 3:

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**org.apache.kafka.connect.data.Time** Represents the number of milliseconds since midnight, and does not include timezone information. SQL Server allows **P** to be in the range 0-7 to store up to tenth of a microsecond precision, though this mode results in a loss of precision when **P** > 3.
DATETIME	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information.
SMALLDATETIME	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds past the epoch, and does not include timezone information.
DATETIME2	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information. SQL Server allows **P** to be in the range 0-7 to store up to tenth of a microsecond precision, though this mode results in a loss of precision when **P** > 3.

Timestamp values

The **DATETIME**, **SMALLDATETIME** and **DATETIME2** types represent a timestamp without time zone information. Such columns are converted into an equivalent Kafka Connect value based on UTC. So for instance the **DATETIME2** value "2018-06-20 15:13:16.945104" is represented by a **io.debezium.time.MicroTimestamp** with the value "1529507596945104".

Note that the timezone of the JVM running Kafka Connect and Debezium does not affect this conversion.

Decimal values

Debezium connectors handle decimals according to the setting of the **decimal.handling.mode** connector configuration property.

decimal.handling.mode=precise

Table 8.10. Mappings when **decimal.handling.mode=precise**

SQL Server type	Literal type (schema type)	Semantic type (schema name)
NUMERIC[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
DECIMAL[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
SMALLMONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
MONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.

decimal.handling.mode=double

Table 8.11. Mappings when **decimal.handling.mode=double**

SQL Server type	Literal type	Semantic type
NUMERIC[(M[,D])]	**FLOAT64**	*n/a*
DECIMAL[(M[,D])]	**FLOAT64**	*n/a*
SMALLMONEY[(M[,D])]	**FLOAT64**	*n/a*
MONEY[(M[,D])]	**FLOAT64**	*n/a*

decimal.handling.mode=string

Table 8.12. Mappings when **decimal.handling.mode=string**

SQL Server type	Literal type	Semantic type
NUMERIC[(M[,D])]	**STRING**	*n/a*
DECIMAL[(M[,D])]	**STRING**	*n/a*
SMALLMONEY[(M[,D])]	**STRING**	*n/a*
MONEY[(M[,D])]	**STRING**	*n/a*

8.3. SETTING UP SQL SERVER TO RUN A DEBEZIUM CONNECTOR

For Debezium to capture change events from SQL Server tables, a SQL Server administrator with the necessary privileges must first run a query to enable CDC on the database. The administrator must then enable CDC for each table that you want Debezium to capture.

For details about setting up SQL Server for use with the Debezium connector, see the following sections:

- Section 8.3.1, "Enabling CDC on the SQL Server database"
- Section 8.3.2, "Enabling CDC on a SQL Server table"
- Section 8.3.3, "Verifying that the user has access to the CDC table"
- Section 8.3.4, "SQL Server on Azure"
- Section 8.3.5, "Effect of SQL Server capture job agent configuration on server load and latency"
- Section 8.3.6, "SQL Server capture job agent configuration parameters"

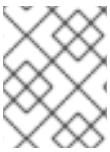
After CDC is applied, it captures all of the **INSERT**, **UPDATE**, and **DELETE** operations that are committed to the tables for which CDD is enabled. The Debezium connector can then capture these events and emit them to Kafka topics.

8.3.1. Enabling CDC on the SQL Server database

Before you can enable CDC for a table, you must enable it for the SQL Server database. A SQL Server administrator enables CDC by running a system stored procedure. System stored procedures can be run by using SQL Server Management Studio, or by using Transact-SQL.

Prerequisites

- You are a member of the *sysadmin* fixed server role for the SQL Server.
- You are a db_owner of the database.
- The SQL Server Agent is running.



NOTE

The SQL Server CDC feature processes changes that occur in user-created tables only. You cannot enable CDC on the SQL Server **master** database.

Procedure

1. From the **View** menu in SQL Server Management Studio, click **Template Explorer**.
2. In the **Template Browser**, expand **SQL Server Templates**.
3. Expand **Change Data Capture > Configuration**and then click **Enable Database for CDC**.
4. In the template, replace the database name in the **USE** statement with the name of the database that you want to enable for CDC.

5. Run the stored procedure **sys.sp_cdc_enable_db** to enable the database for CDC.
   After the database is enabled for CDC, a schema with the name **cdc** is created, along with a CDC user, metadata tables, and other system objects.

   The following example shows how to enable CDC for the database **MyDB**:

   Example: Enabling a SQL Server database for the CDC template

```
USE MyDB
GO
EXEC sys.sp_cdc_enable_db
GO
```

8.3.2. Enabling CDC on a SQL Server table

A SQL Server administrator must enable change data capture on the source tables that you want to Debezium to capture. The database must already be enabled for CDC. To enable CDC on a table, a SQL Server administrator runs the stored procedure **sys.sp_cdc_enable_table** for the table. The stored procedures can be run by using SQL Server Management Studio, or by using Transact-SQL. SQL Server CDC must be enabled for every table that you want to capture.

Prerequisites

- CDC is enabled on the SQL Server database.
- The SQL Server Agent is running.
- You are a member of the **db_owner** fixed database role for the database.

Procedure

1. From the **View** menu in SQL Server Management Studio, click **Template Explorer**.
2. In the **Template Browser**, expand **SQL Server Templates**.
3. Expand **Change Data Capture > Configuration**, and then click **Enable Table Specifying Filegroup Option**.
4. In the template, replace the table name in the **USE** statement with the name of the table that you want to capture.
5. Run the stored procedure **sys.sp_cdc_enable_table**.
   The following example shows how to enable CDC for the table **MyTable**:

   Example: Enabling CDC for a SQL Server table

```
USE MyDB
GO

EXEC sys.sp_cdc_enable_table
@source_schema = N'dbo',
@source_name   = N'MyTable', //<.>
@role_name     = N'MyRole',  //<.>
```

```
@filegroup_name = N'MyDB_CT',//<.>
@supports_net_changes = 0
GO
```

<.> Specifies the name of the table that you want to capture. <.> Specifies a role **MyRole** to which you can add users to whom you want to grant **SELECT** permission on the captured columns of the source table. Users in the **sysadmin** or **db_owner** role also have access to the specified change tables. Set the value of **@role_name** to **NULL**, to allow only members in the **sysadmin** or **db_owner** to have full access to captured information. <.> Specifies the **filegroup** where SQL Server places the change table for the captured table. The named **filegroup** must already exist. It is best not to locate change tables in the same **filegroup** that you use for source tables.

8.3.3. Verifying that the user has access to the CDC table

A SQL Server administrator can run a system stored procedure to query a database or table to retrieve its CDC configuration information. The stored procedures can be run by using SQL Server Management Studio, or by using Transact-SQL.

Prerequisites

- You have **SELECT** permission on all of the captured columns of the capture instance. Members of the **db_owner** database role can view information for all of the defined capture instances.
- You have membership in any gating roles that are defined for the table information that the query includes.

Procedure

1. From the **View** menu in SQL Server Management Studio, click **Object Explorer**.
2. From the Object Explorer, expand **Databases**, and then expand your database object, for example, **MyDB**.
3. Expand **Programmability > Stored Procedures > System Stored Procedures**
4. Run the **sys.sp_cdc_help_change_data_capture** stored procedure to query the table. Queries should not return empty results.

   The following example runs the stored procedure **sys.sp_cdc_help_change_data_capture** on the database **MyDB**:

   Example: Querying a table for CDC configuration information

   ```
   USE MyDB;
   GO
   EXEC sys.sp_cdc_help_change_data_capture
   GO
   ```

   The query returns configuration information for each table in the database that is enabled for CDC and that contains change data that the caller is authorized to access. If the result is empty, verify that the user has privileges to access both the capture instance and the CDC tables.

8.3.4. SQL Server on Azure

The Debezium SQL Server connector can be used with SQL Server on Azure. Refer to this example for configuring CDC for SQL Server on Azure and using it with Debezium.

8.3.5. Effect of SQL Server capture job agent configuration on server load and latency

When a database administrator enables change data capture for a source table, the capture job agent begins to run. The agent reads new change event records from the transaction log and replicates the event records to a change data table. Between the time that a change is committed in the source table, and the time that the change appears in the corresponding change table, there is always a small latency interval. This latency interval represents a gap between when changes occur in the source table and when they become available for Debezium to stream to Apache Kafka.

Ideally, for applications that must respond quickly to changes in data, you want to maintain close synchronization between the source and change tables. You might imagine that running the capture agent to continuously process change events as rapidly as possible might result in increased throughput and reduced latency – populating change tables with new event records as soon as possible after the events occur, in near real time. However, this is not necessarily the case. There is a performance penalty to pay in the pursuit of more immediate synchronization. Each time that the capture job agent queries the database for new event records, it increases the CPU load on the database host. The additional load on the server can have a negative effect on overall database performance, and potentially reduce transaction efficiency, especially during times of peak database use.

It's important to monitor database metrics so that you know if the database reaches the point where the server can no longer support the capture agent's level of activity. If you notice performance problems, there are SQL Server capture agent settings that you can modify to help balance the overall CPU load on the database host with a tolerable degree of latency.

8.3.6. SQL Server capture job agent configuration parameters

On SQL Server, parameters that control the behavior of the capture job agent are defined in the SQL Server table **msdb.dbo.cdc_jobs**. If you experience performance issues while running the capture job agent, adjust capture jobs settings to reduce CPU load by running the **sys.sp_cdc_change_job** stored procedure and supplying new values.

NOTE

Specific guidance about how to configure SQL Server capture job agent parameters is beyond the scope of this documentation.

The following parameters are the most significant for modifying capture agent behavior for use with the Debezium SQL Server connector:

pollinginterval

- Specifies the number of seconds that the capture agent waits between log scan cycles.
- A higher value reduces the load on the database host and increases latency.
- A value of **0** specifies no wait between scans.
- The default value is **5**.

maxtrans

- Specifies the maximum number of transactions to process during each log scan cycle. After the capture job processes the specified number of transactions, it pauses for the length of time that the **pollinginterval** specifies before the next scan begins.
- A lower value reduces the load on the database host and increases latency.
- The default value is **500**.

maxscans

- Specifies a limit on the number of scan cycles that the capture job can attempt in capturing the full contents of the database transaction log. If the **continuous** parameter is set to **1**, the job pauses for the length of time that the **pollinginterval** specifies before it resumes scanning.
- A lower values reduces the load on the database host and increases latency.
- The default value is **10**.

Additional resources

- For more information about capture agent parameters, see the SQL Server documentation.

8.4. DEPLOYMENT OF DEBEZIUM SQL SERVER CONNECTORS

You can use either of the following methods to deploy a Debezium SQL Server connector:

- Use AMQ Streams to automatically create an image that includes the connector plug-in . This is the preferred method.
- Build a custom Kafka Connect container image from a Dockerfile .

Additional resources

- Section 8.4.4, “Descriptions of Debezium SQL Server connector configuration properties”

8.4.1. SQL Server connector deployment using AMQ Streams

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in.

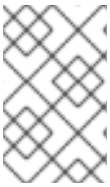
During the deployment process, you create and use the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance and includes information about the connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Apicurio Registry artifacts, or the Debezium

scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.



NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

8.4.2. Using AMQ Streams to deploy a Debezium SQL Server connector

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository or another configured HTTP server.

The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium license.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.
- Depending on how you intend to store the Kafka Connect build image, you need registry permissions or you must create an ImageStream resource:

To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

- An account and permissions to create and manage images in the registry.

To store the build image as a native OpenShift ImageStream

- An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a Debezium **KafkaConnect** custom resource (CR) for the connector, or modify an existing one. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

Example 8.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

In the example that follows, the custom resource is configured to download the following artifacts:

- The Debezium SQL Server connector archive.
- The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
- The Debezium scripting SMT archive and the associated scripting engine that you want to use with the Debezium connector. The SMT archive and scripting language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-sqlserver
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-sqlserver/1.9.7.Final-redhat-<build_number>/debezium-connector-sqlserver-1.9.7.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
```

```
        url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3-redhat-<build-number>.zip (8)
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.9.7.Final/debezium-scripting-1.9.7.Final.zip (9)
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar (10)
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json3.0.11/groovy-json-3.0.11.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093
```

Table 8.13. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry such as Docker Hub or Quay, or **imagestream** to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	The **plugins** configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in **name**, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.
6	The value of **artifacts.type** specifies the file type of the artifact specified in the **artifacts.url**. Valid types are **zip**, **tgz**, or **jar**. Debezium connector archives are provided in **.zip** file format. The **type** value must match the type of the file that is referenced in the **url** field.

Item	Description
7	The value of **artifacts.url** specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.
8	(Optional) Specifies the artifact **type** and **url** for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.
9	(Optional) Specifies the artifact **type** and **url** for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.
10	(Optional) Specifies the artifact **type** and **url** for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT. **IMPORTANT** If you use AMQ Streams to incorporate the connector plug-in into your Kafka Connect image, for each of the required scripting language components **artifacts.url** must specify the location of a JAR file, and the value of **artifacts.type** must also be set to **jar**. Invalid values cause the connector fails at runtime. To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries: • **groovy** • **groovy-jsr223** (scripting agent) • **groovy-json** (module for parsing JSON strings) As an alternative, the Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy.
After the build completes, the Operator pushes the image to the specified registry or

ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **sqlserver-inventory-connector.yaml**

Example 8.2. sqlserver-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-sqlserver (1)
spec:
  class: io.debezium.connector.sqlserver.SqlServerConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: sqlserver.debezium-sqlserver.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_sqlserver (10)
    database.include.list: public.inventory (11)
```

Table 8.14. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	The number of tasks that can operate concurrently.
4	The connector's configuration.
5	The address of the host database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.

Item	Description
8	The password for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

5. Create the connector resource by running the following command:

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by **spec.config.database.dbname** in the **KafkaConnector** CR. After the connector pod is ready, Debezium is running.

You are now ready to verify the Debezium SQL Server deployment.

8.4.3. Deploying a Debezium SQL Server connector by building a custom Kafka Connect container image from a Dockerfile

To deploy a Debezium SQL Server connector, you must build a custom Kafka Connect container image that contains the Debezium connector archive, and then push this container image to a container registry. You then need to create the following custom resources (CRs):

- A **KafkaConnect** CR that defines your Kafka Connect instance. The **image** property in the CR specifies the name of the container image that you create to run your Debezium connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed. AMQ Streams offers operators and images that bring Apache Kafka to OpenShift.
- A **KafkaConnector** CR that defines your Debezium SQL Server connector. Apply this CR to the same OpenShift instance where you apply the **KafkaConnect** CR.

Prerequisites

- SQL Server is running and you completed the steps to set up SQL Server to work with a Debezium connector.

- AMQ Streams is deployed on OpenShift and is running Apache Kafka and Kafka Connect. For more information, see Deploying and Upgrading AMQ Streams on OpenShift
- Podman or Docker is installed.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium SQL Server container for Kafka Connect:
   a. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. For example, from a terminal window, enter the following command:

```
cat <<EOF >debezium-container-for-sqlserver.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium (2)
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-sqlserver/1.9.7.Final-redhat-<build_number>/debezium-connector-sqserverl-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& unzip debezium-connector-sqlserver-1.9.7.Final-redhat-<build_number>-plugin.zip \
&& rm debezium-connector-sqlserver-1.9.7.Final-redhat-<build_number>-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

Item	Description
1	You can specify any file name that you want.
2	Specifies the path to your Kafka Connect plug-ins directory. If your Kafka Connect plug-ins directory is in a different location, replace this path with the actual path of your directory.

The command creates a Dockerfile with the name **debezium-container-for-sqlserver.yaml** in the current directory.

   b. Build the container image from the **debezium-container-for-sqlserver.yaml** Docker file that you created in the previous step. From the directory that contains the file, open a terminal window and enter one of the following commands:

```
podman build -t debezium-container-for-sqlserver:latest .
```

```
docker build -t debezium-container-for-sqlserver:latest .
```

The preceding commands build a container image with the name **debezium-container-for-sqlserver**.

c. Push your custom image to a container registry, such as quay.io or an internal container registry. The container registry must be available to the OpenShift instance where you want to deploy the image. Enter one of the following commands:

```
podman push <myregistry.io>/debezium-container-for-sqlserver:latest
```

```
docker push <myregistry.io>/debezium-container-for-sqlserver:latest
```

d. Create a new Debezium SQL Server KafkaConnect custom resource (CR). For example, create a KafkaConnect CR with the name **dbz-connect.yaml** that specifies **annotations** and **image** properties as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  #...
  image: debezium-container-for-sqlserver 2
```

Item	Description
1	**metadata.annotations** indicates to the Cluster Operator that**KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.
2	**spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

e. Apply the **KafkaConnect** CR to the OpenShift Kafka Connect environment by entering the following command:

```
oc create -f dbz-connect.yaml
```

The command adds a Kafka Connect instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium SQL Server connector instance.
You configure a Debezium SQL Server connector in a **.yaml** file that specifies the configuration properties for the connector. The connector configuration might instruct Debezium to produce events for a subset of the schemas and tables, or it might set properties so that Debezium ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed.

The following example configures a Debezium connector that connects to a SQL server host, **192.168.99.100**, on port **1433**. This host has a database named **testDB**, a table with the name **customers**, and **fulfillment** is the server's logical name.

SQL Server **fulfillment-connector.yaml**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: fulfillment-connector (1)
  labels:
    strimzi.io/cluster: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.sqlserver.SqlServerConnector (2)
  config:
    database.hostname: 192.168.99.100 (3)
    database.port: 1433 (4)
    database.user: debezium (5)
    database.password: dbz (6)
    database.dbname: testDB (7)
    database.server.name: fullfullment (8)
    database.include.list: dbo.customers (9)
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 (10)
    database.history.kafka.topic: dbhistory.fullfillment (11)
```

Table 8.15. Descriptions of connector configuration settings

Item	Description
1	The name of our connector when we register it with a Kafka Connect service.
2	The name of this SQL Server connector class.
3	The address of the SQL Server instance.
4	The port number of the SQL Server instance.
5	The name of the SQL Server user.
6	The password for the SQL Server user.
7	The name of the database to capture changes from.
8	The logical name of the SQL Server instance/cluster, which forms a namespace and is used in all the names of the Kafka topics to which the connector writes, the Kafka Connect schema names, and the namespaces of the corresponding Avro schema when the Avro converter is used.
9	A list of all tables whose changes Debezium should capture.

Item	Description
10	The list of Kafka brokers that this connector will use to write and recover DDL statements to the database history topic.
11	The name of the database history topic where the connector will write and recover DDL statements. This topic is for internal use only and should not be used by consumers.

3. Create your connector instance with Kafka Connect. For example, if you saved your **KafkaConnector** resource in the **fulfillment-connector.yaml** file, you would run the following command:

```
oc apply -f fulfillment-connector.yaml
```

The preceding command registers **fulfillment-connector** and the connector starts to run against the **testDB** database as defined in the **KafkaConnector** CR.

Verifying that the Debezium SQL Server connector is running

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

Prerequisites

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift **oc** CLI client is installed.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Check the status of the **KafkaConnector** resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home → Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaConnector**.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example **inventory-connector-sqlserver**.

d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.

- From a terminal window:

a. Enter the following command:

```
oc describe KafkaConnector <connector-name> -n <project>
```

For example,

```
oc describe KafkaConnector inventory-connector-sqlserver -n debezium
```

The command returns status information that is similar to the following output:

Example 8.3. KafkaConnector resource status

```
Name:         inventory-connector-sqlserver
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations:  <none>
API Version:  kafka.strimzi.io/v1beta2
Kind:         KafkaConnector

...

Status:
  Conditions:
    Last Transition Time:  2021-12-08T17:41:34.897153Z
    Status:                True
    Type:                  Ready
  Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  10.131.1.124:8083
    Name:         inventory-connector-sqlserver
    Tasks:
      Id:               0
      State:            RUNNING
      worker_id:        10.131.1.124:8083
    Type:               source
  Observed Generation:  1
  Tasks Max:            1
  Topics:
    inventory_connector_sqlserver
    inventory_connector_sqlserver.inventory.addresses
    inventory_connector_sqlserver.inventory.customers
    inventory_connector_sqlserver.inventory.geom
    inventory_connector_sqlserver.inventory.orders
    inventory_connector_sqlserver.inventory.products
    inventory_connector_sqlserver.inventory.products_on_hand
Events:  <none>
```

2. Verify that the connector created Kafka topics:

- From the OpenShift Container Platform web console.
  a. Navigate to **Home → Search**.
  b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.
  c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-sqlserver.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.
  d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
- From a terminal window:
  a. Enter the following command:

```
oc get kafkatopics
```

The command returns status information that is similar to the following output:

Example 8.4. KafkaTopic resource status

```
NAME                                                                    CLUSTER
PARTITIONS   REPLICATION FACTOR   READY
connect-cluster-configs                                                 debezium-
kafka-cluster   1           1                   True
connect-cluster-offsets                                                 debezium-
kafka-cluster   25          1                   True
connect-cluster-status                                                  debezium-
kafka-cluster   5           1                   True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
debezium-kafka-cluster   50          1                   True
inventory-connector-sqlserver---a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster   1           1                   True
inventory-connector-sqlserver.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480          debezium-kafka-cluster
1           1                   True
inventory-connector-sqlserver.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1                   True
inventory-connector-sqlserver.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5              debezium-kafka-cluster
1           1                   True
inventory-connector-sqlserver.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d              debezium-kafka-cluster
1           1                   True
inventory-connector-sqlserver.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1                   True
inventory-connector-sqlserver.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1                   True
schema-changes.inventory
debezium-kafka-cluster   1           1                   True
```

```
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1           1                   True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-cluster   1
1                   True
```

3. Check topic content.

- From a terminal window, enter the following command:

```
oc exec -n <project>  -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=<topic-name>
```

For example,

```
 oc exec -n debezium  -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_sqlserver.inventory.products_on_hand
```

The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **inventory_connector_sqlserver.inventory.addresses**.

For each event in the topic, the command returns information that is similar to the following output:

Example 8.5. Content of a Debezium change event

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_conne
ctor_sqlserver.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_sqlserver.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory_connector
_sqlserver.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"version"},
{"type":"string","optional":false,"field":"connector"},
{"type":"string","optional":false,"field":"name"},
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
```

{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connecto
r.sqlserver.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"}],"optional":true,"field":"transacti
on"}],"optional":false,"name":"inventory_connector_sqlserver.inventory.products_on_hand.E
nvelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.9.7.Final-redhat-
00001","connector":"sqlserver","name":"inventory_connector_sqlserver","ts_ms":163898524
7805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server
_id":0,"gtid":null,"file":"sqlserver-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

For the complete list of the configuration properties that you can set for the Debezium SQL Server connector, see SQL Server connector properties.

Results

When the connector starts, it performs a consistent snapshot of the SQL Server databases that the connector is configured for. The connector then starts generating data change events for row-level operations and streaming the change event records to Kafka topics.

8.4.4. Descriptions of Debezium SQL Server connector configuration properties

The Debezium SQL Server connector has numerous configuration properties that you can use to achieve the right connector behavior for your application. Many properties have default values.

Information about the properties is organized as follows:

- Required connector configuration properties
- Advanced connector configuration properties
- Database history connector configuration properties that control how Debezium processes events that it reads from the database history topic.
  - Pass-through database history properties
- Pass-through database driver properties that control the behavior of the database driver.

Required Debezium SQL Server connector configuration properties

The following configuration properties are *required* unless a default value is available.

Property	Default	Description
name	No default	Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class	No default	The name of the Java class for the connector. Always use a value of **io.debezium.connector.sqlserver.SqlServerConnector** for the SQL Server connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The SQL Server connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname	No default	IP address or hostname of the SQL Server database server.
database.port	**1433**	Integer port number of the SQL Server database server.
database.user	No default	Username to use when connecting to the SQL Server database server. Can be omitted when using Kerberos authentication, which can be configured using pass-through properties.
database.password	No default	Password to use when connecting to the SQL Server database server.
database.dbname	No default	The name of the SQL Server database from which to stream the changes.
database.instance	No default	Specifies the instance name of the SQL Server named instance.

Property	Default	Description
database.server.name	No default	Logical name that identifies and provides a namespace for the SQL Server database server that you want Debezium to capture. The logical name should be unique across all other connectors, since it is used as the prefix for all Kafka topic names that receive records from this connector. Only alphanumeric characters, hyphens, dots and underscores must be used in the database server logical name. + **WARNING** Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value. The connector is also unable to recover its database history topic.

Property	Default	Description
schema.include.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **want** to capture changes. Any schema name not included in **schema.include.list** is excluded from having its changes captured. By default, all non-system schemas have their changes captured. Do not also set the **schema.exclude.list** property.
schema.exclude.list	No default	An optional, comma-separated list of regular expressions that match names of schemas for which you **do not** want to capture changes. Any schema whose name is not included in **schema.exclude.list** has its changes captured, with the exception of system schemas. Do not also set the **schema.include.list** property.
table.include.list	No default	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables that you want Debezium to capture; any table that is not included in **table.include.list** is excluded from capture. Each identifier is of the form *schemaName.tableName*. By default, the connector captures all non-system tables for the designated schemas. Must not be used with **table.exclude.list**.
table.exclude.list	No default	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for the tables that you want to exclude from being captured; Debezium captures all tables that are not included in **table.exclude.list**. Each identifier is of the form *schemaName.tableName*. Must not be used with **table.include.list**.
column.include.list	*empty string*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that should be included in the change event message values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Note that primary key columns are always included in the event's key, even if not included in the value. Do not also set the **column.exclude.list** property.

Property	Default	Description
column.exclude.list	*empty string*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that should be excluded from change event message values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Note that primary key columns are always included in the event's key, also if excluded from the value. Do not also set the **column.include.list** property.
column.mask.hash.*hashAlgorithm*.with.salt.*salt*; column.mask.hash.v2.*hashAlgorithm*.with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form `<*schemaName*>.<*tableName*>._<columnName>`. In the resulting change event record, the values for the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. In the following example, **CzQMA0cB5K** is a randomly selected salt. column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts. Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting data set might not be completely masked. Hashing strategy version 2 should be used to ensure fidelity if the value is being hashed in different places or systems.

Property	Default	Description
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision, including: **adaptive** (the default) captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type; or **connect** always represents time and timestamp values using Kafka Connect's built-in representations for Time, Date, and Timestamp, which uses millisecond precision regardless of the database columns' precision. See temporal values.
decimal.handling.mode	**precise**	Specifies how the connector should handle values for **DECIMAL** and **NUMERIC** columns: **precise** (the default) represents them precisely using **java.math.BigDecimal** values represented in change events in a binary form. **double** represents them using **double** values, which may result in a loss of precision but is easier to use. **string** encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded with a key that contains the database name and a value that is a JSON structure that describes the schema update. This is independent of how the connector internally records database history. The default is **true**.

Property	Default	Description
tombstones.on.delete	**true**	Controls whether a *delete* event is followed by a tombstone event. **true** - a delete operation is represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is emitted. After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case log compaction is enabled for the topic.
column.truncate.to.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event message values if the field values are longer than the specified number of characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.
column.mask.with.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*) characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.

Property	Default	Description
column.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** is used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.
datatype.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** will be used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of the form *schemaName.tableName.typeName*. See SQL Server data types for the list of SQL Server-specific data type names.

Property	Default	Description
message.key.columns	*n/a*	A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables. By default, Debezium uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns. To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format: ***<fully-qualified_tableName>:<keyColumn>,<keyColumn>*** To base a table key on multiple column names, insert commas between the column names. Each fully-qualified table name is a regular expression in the following format: ***<schemaName>.<tableName>*** The property can include entries for multiple tables. Use a semicolon to separate table entries in the list. The following example sets the message key for the tables **inventory.customers** and **purchase.orders**: **inventory.customers:pk1,pk2;(.*).purchaseorders:pk3,pk4** For the table **inventory.customer**, the columns **pk1** and **pk2** are specified as the message key. For the **purchaseorders** tables in any schema, the columns **pk3** and **pk4** server as the message key. There is no limit to the number of columns that you use to create custom message keys. However, it's best to use the minimum number that are required to specify a unique key.

Property	Default	Description
binary.handling.mode	bytes	Specifies how binary (**binary**, **varbinary**) columns should be represented in change events, including: **bytes** represents binary data as byte array (default), **base64** represents binary data as base64-encoded String, **hex** represents binary data as hex-encoded (base16) String
schema.name.adjustment.mode	avro	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings: • **avro** replaces the characters that cannot be used in the Avro type name with underscore. • **none** does not apply any adjustment.

Advanced SQL Server connector configuration properties

The following *advanced* configuration properties have good defaults that will work in most situations and therefore rarely need to be specified in the connector's configuration.

Property	Default	Description

Property	Default	Description

converters	No default	Enumerates a comma-separated list of the symbolic names of the custom converter instances that the connector can use. For example,
		
		isbn
		
		You must set the **converters** property to enable the connector to use a custom converter.
		
		For each converter that you configure for a connector, you must also add a **.type** property, which specifies the fully-qualifed name of the class that implements the converter interface. The **.type** property uses the following format:
		
		***<converterSymbolicName>*.type**
		
		For example,
		
		`isbn.type: io.debezium.test.IsbnConverter`
		
		If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate any additional configuration parameter with a converter, prefix the parameter names with the symbolic name of the converter. For example,
		
		`isbn.schema.name: io.debezium.sqlserver.type.Isbn`

Property	Default	Description
snapshot.mode	*initial*	A mode for taking an initial snapshot of the structure and optionally data of captured tables. Once the snapshot is complete, the connector will continue reading change events from the database's redo logs. The following values are supported: • **initial**: Takes a snapshot of structure and data of captured tables; useful if topics should be populated with a complete representation of the data from the captured tables. • **initial_only**: Takes a snapshot of structure and data like **initial** but instead does not transition into streaming changes once the snapshot has completed. • **schema_only**: Takes a snapshot of the structure of captured tables only; useful if only changes happening from now onwards should be propagated to topics.
snapshot.include.collection.list	All tables specified in **table.include.list**	An optional, comma-separated list of regular expressions that match the fully-qualified names (***<dbName>.<schemaName>.<tableName>***) of the tables to include in a snapshot. The specified items must be named in the connector's **table.include.list** property. This property takes effect only if the connector's **snapshot.mode** property is set to a value other than **never**. This property does not affect the behavior of incremental snapshots.

Property	Default	Description
snapshot.isolation.mode	*repeatable_read*	Mode to control which transaction isolation level is used and how long the connector locks tables that are designated for capture. The following values are supported: • **read_uncommitted** • **read_committed** • **repeatable_read** • **snapshot** • **exclusive** (**exclusive** mode uses repeatable read isolation level, however, it takes the exclusive lock on all tables to be read). The **snapshot**, **read_committed** and **read_uncommitted** modes do not prevent other transactions from updating table rows during initial snapshot. The **exclusive** and **repeatable_read** modes do prevent concurrent updates. Mode choice also affects data consistency. Only **exclusive** and **snapshot** modes guarantee full consistency, that is, initial snapshot and streaming logs constitute a linear history. In case of **repeatable_read** and **read_committed** modes, it might happen that, for instance, a record added appears twice - once in initial snapshot and once in streaming phase. Nonetheless, that consistency level should do for data mirroring. For **read_uncommitted** there are no data consistency guarantees at all (some data might be lost or corrupted).
event.processing.failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events. **fail** will propagate the exception (indicating the offset of the problematic event), causing the connector to stop. **warn** will cause the problematic event to be skipped and the offset of the problematic event to be logged. **skip** will cause the problematic event to be skipped.

Property	Default	Description
poll.interval.ms	1000	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.
max.queue.size	8192	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When Debezium reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of **max.queue.size** to be larger than the value of **max.batch.size**.
max.queue.size.in.bytes	0	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value. If **max.queue.size** is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set **max.queue.size=1000**, and **max.queue.size.in.bytes=5000**, writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.
max.batch.size	2048	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector.

Property	Default	Description
heartbeat.interval.ms	**0**	Controls how frequently heartbeat messages are sent. This property contains an interval in milliseconds that defines how frequently the connector sends messages to a heartbeat topic. The property can be used to confirm whether the connector is still receiving change events from the database. You also should leverage heartbeat messages in cases where only records in non-captured tables are changed for a longer period of time. In such situation the connector would proceed to read the log from the database but never emit any change messages into Kafka, which in turn means that no offset updates are committed to Kafka. This may result in more change events to be re-sent after a connector restart. Set this parameter to **0** to not send heartbeat messages at all. Disabled by default.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.
snapshot.delay.ms	No default	An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size	**2000**	Specifies the maximum number of rows that should be read in one go from each table while taking a snapshot. The connector will read the table contents in multiple batches of this size. Defaults to 2000.
query.fetch.size	No default	Specifies the number of rows that will be fetched for each database round-trip of a given query. Defaults to the JDBC driver's default fetch size.

Property	Default	Description
snapshot.lock.timeout.ms	**10000**	An integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If table locks cannot be acquired in this time interval, the snapshot will fail (also see snapshots). When set to **0** the connector will fail immediately when it cannot obtain the lock. Value **-1** indicates infinite waiting.

Property	Default	Description
snapshot.select.statement.overrides	No default	Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log. The property contains a comma-separated list of fully-qualified table names in the form ***<schemaName>.<tableName>***. For example, **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** For each table in the list, add a further configuration property that specifies the **SELECT** statement for the connector to run on the table when it takes a snapshot. The specified **SELECT** statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this **SELECT** statement property: **snapshot.select.statement.overrides.*<schemaName>.<tableName>***. For example, **snapshot.select.statement.overrides.customers.orders**. Example: From a **customers.orders** table that includes the soft-delete column, **delete_flag**, add the following properties if you want a snapshot to include only those records that are not soft-deleted: "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" In the resulting snapshot, the connector includes only the records for which **delete_flag = 0**.

Property	Default	Description
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names are sanitized to adhere to Avro naming requirements. See Avro naming for more details.
provide.transaction.metadata	**false**	When set to **true** Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata.
transaction.topic	**${database.server.name}.transaction**	Controls the name of the topic to which the connector sends transaction metadata messages. The placeholder **${database.server.name}** can be used for referring to the connector's logical name; defaults to **${database.server.name}.transaction**, for example **dbserver1.transaction**. For more information, see Transaction Metadata.
retriable.restart.connector.wait.ms	10000 (10 seconds)	The number of milli-seconds to wait before restarting a connector after a retriable error occurs.
skipped.operations	No default	comma-separated list of operation types that will be skipped during streaming. The operations include: **c** for inserts/create, **u** for updates, and **d** for deletes. By default, no operations are skipped.
signal.data.collection	No default value	Fully-qualified name of the data collection that is used to send signals to the connector. Use the following format to specify the collection name: ***<databaseName>.<schemaName>.<tableName>***

Property	Default	Description
incremental.snapshot.allow.schema.changes	**false**	Allow schema changes during an incremental snapshot. When enabled the connector will detect schema change during an incremental snapshot and re-select a current chunk to avoid locking DDLs. Note that changes to a primary key are not supported and can cause incorrect results if performed during an incremental snapshot. Another limitation is that if a schema change affects only columns' default values, then the change won't be detected until the DDL is processed from the binlog stream. This doesn't affect the snapshot events' values, but the schema of snapshot events may have outdated defaults.
incremental.snapshot.chunk.size	**1024**	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.
max.iteration.transactions	0	Specifies the maximum number of transactions per iteration to be used to reduce the memory footprint when streaming changes from multiple tables in a database. When set to **0** (the default), the connector uses the current maximum LSN as the range to fetch changes from. When set to a value greater than zero, the connector uses the n-th LSN specified by this setting as the range to fetch changes from.
incremental.snapshot.option.recompile	**false**	Uses OPTION(RECOMPILE) query option to all SELECT statements used during an incremental snapshot. This can help to solve parameter sniffing issues that may occur but can cause increased CPU load on the source database, depending on the frequency of query execution.

Debezium SQL Server connector database history configuration properties

Debezium provides a set of **database.history.*** properties that control how the connector interacts with the schema history topic.

The following table describes the **database.history** properties for configuring the Debezium connector.

Table 8.16. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic	No default	The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.bootstrap.servers	No default	A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. This connection is used for retrieving the database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
database.history.kafka.recovery.poll.interval.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait during startup/recovery while polling for persisted data. The default is 100ms.
database.history.kafka.query.timeout.ms	**3000**	An integer value that specifies the maximum number of milliseconds the connector should wait while fetching cluster information using Kafka admin client.
database.history.kafka.recovery.attempts	**4**	The maximum number of times that the connector should try to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is **recovery.attempts** x **recovery.poll.interval.ms**.
database.history.skip.unparseable.ddl	**false**	A Boolean value that specifies whether the connector should ignore malformed or unknown database statements or stop processing so a human can fix the issue. The safe default is **false**. Skipping should be used only with care as it can lead to data loss or mangling when the binlog is being processed.
database.history.store.only.monitored.tables.ddl *Deprecated and scheduled for removal in a future release; use **database.history.store.only.captured.tables.ddl** instead.*	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Property	Default	Description
database.history.store.only.captured.tables.ddl	**false**	A Boolean value that specifies whether the connector should record all DDL statements **true** records only those DDL statements that are relevant to tables whose changes are being captured by Debezium. Set to **true** with care because missing data might become necessary if you change which tables have their changes captured. The safe default is **false**.

Pass-through database history properties for configuring producer and consumer clients

Debezium relies on a Kafka producer to write schema changes to database history topics. Similarly, it relies on a Kafka consumer to read from database history topics when a connector starts. You define the configuration for the Kafka producer and consumer clients by assigning values to a set of pass-through configuration properties that begin with the **database.history.producer.*** and **database.history.consumer.*** prefixes. The pass-through producer and consumer database history properties control a range of behaviors, such as how these clients secure connections with the Kafka broker, as shown in the following example:

```
database.history.producer.security.protocol=SSL
database.history.producer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.producer.ssl.keystore.password=test1234
database.history.producer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.producer.ssl.truststore.password=test1234
database.history.producer.ssl.key.password=test1234

database.history.consumer.security.protocol=SSL
database.history.consumer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore.jks
database.history.consumer.ssl.keystore.password=test1234
database.history.consumer.ssl.truststore.location=/var/private/ssl/kafka.server.truststore.jks
database.history.consumer.ssl.truststore.password=test1234
database.history.consumer.ssl.key.password=test1234
```

Debezium strips the prefix from the property name before it passes the property to the Kafka client.

See the Kafka documentation for more details about Kafka producer configuration properties and Kafka consumer configuration properties.

Debezium SQL Server connector pass-through database driver configuration properties

The Debezium connector provides for pass-through configuration of the database driver. Pass-through database properties begin with the prefix **database.***. For example, the connector passes properties such as **database.foobar=false** to the JDBC URL.

As is the case with the pass-through properties for database history clients, Debezium strips the prefixes from the properties before it passes them to the database driver.

8.5. REFRESHING CAPTURE TABLES AFTER A SCHEMA CHANGE

When change data capture is enabled for a SQL Server table, as changes occur in the table, event records are persisted to a capture table on the server. If you introduce a change in the structure of the source table change, for example, by adding a new column, that change is not dynamically reflected in the change table. For as long as the capture table continues to use the outdated schema, the Debezium connector is unable to emit data change events for the table correctly. You must intervene to refresh the capture table to enable the connector to resume processing change events.

Because of the way that CDC is implemented in SQL Server, you cannot use Debezium to update capture tables. To refresh capture tables, one must be a SQL Server database operator with elevated privileges. As a Debezium user, you must coordinate tasks with the SQL Server database operator to complete the schema refresh and restore streaming to Kafka topics.

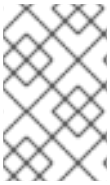
You can use one of the following methods to update capture tables after a schema change:

- Offline schema updates require you to stop the Debezium connector before you can update capture tables.
- Online schema updates can update capture tables while the Debezium connector is running.

There are advantages and disadvantages to using each type of procedure.

WARNING

Whether you use the online or offline update method, you must complete the entire schema update process before you apply subsequent schema updates on the same source table. The best practice is to execute all DDLs in a single batch so the procedure can be run only once.



NOTE

Some schema changes are not supported on source tables that have CDC enabled. For example, if CDC is enabled on a table, SQL Server does not allow you to change the schema of the table if you renamed one of its columns or changed the column type.

NOTE

After you change a column in a source table from **NULL** to **NOT NULL** or vice versa, the SQL Server connector cannot correctly capture the changed information until after you create a new capture instance. If you do not create a new capture table after a change to the column designation, change event records that the connector emits do not correctly indicate whether the column is optional. That is, columns that were previously defined as optional (or **NULL**) continue to be, despite now being defined as **NOT NULL**. Similarly, columns that had been defined as required (**NOT NULL**), retain that designation, although they are now defined as **NULL**.

8.5.1. Running an offline update after a schema change

Offline schema updates provide the safest method for updating capture tables. However, offline updates might not be feasible for use with applications that require high-availability.

Prerequisites

Prerequisites

- An update was committed to the schema of a SQL Server table that has CDC enabled.
- You are a SQL Server database operator with elevated privileges.

Procedure

1. Suspend the application that updates the database.
2. Wait for the Debezium connector to stream all unstreamed change event records.
3. Stop the Debezium connector.
4. Apply all changes to the source table schema.
5. Create a new capture table for the update source table using **sys.sp_cdc_enable_table** procedure with a unique value for parameter **@capture_instance**.
6. Resume the application that you suspended in Step 1.
7. Start the Debezium connector.
8. After the Debezium connector starts streaming from the new capture table, drop the old capture table by running the stored procedure **sys.sp_cdc_disable_table** with the parameter **@capture_instance** set to the old capture instance name.

8.5.2. Running an online update after a schema change

The procedure for completing an online schema updates is simpler than the procedure for running an offline schema update, and you can complete it without requiring any downtime in application and data processing. However, with online schema updates, a potential processing gap can occur after you update the schema in the source database, but before you create the new capture instance. During that interval, change events continue to be captured by the old instance of the change table, and the change data that is saved to the old table retains the structure of the earlier schema. So, for example, if you added a new column to a source table, change events that are produced before the new capture table is ready, do not contain a field for the new column. If your application does not tolerate such a transition period, it is best to use the offline schema update procedure.

Prerequisites

- An update was committed to the schema of a SQL Server table that has CDC enabled.
- You are a SQL Server database operator with elevated privileges.

Procedure

1. Apply all changes to the source table schema.
2. Create a new capture table for the update source table by running the **sys.sp_cdc_enable_table** stored procedure with a unique value for the parameter **@capture_instance**.
3. When Debezium starts streaming from the new capture table, you can drop the old capture table by running the **sys.sp_cdc_disable_table** stored procedure with the parameter **@capture_instance** set to the old capture instance name.

Example: Running an online schema update after a database schema change

The following example shows how to complete an online schema update in the change table after the column **phone_number** is added to the **customers** source table.

1. Modify the schema of the **customers** source table by running the following query to add the **phone_number** field:

```
ALTER TABLE customers ADD phone_number VARCHAR(32);
```

2. Create the new capture instance by running the **sys.sp_cdc_enable_table** stored procedure.

```
EXEC sys.sp_cdc_enable_table @source_schema = 'dbo', @source_name = 'customers',
@role_name = NULL, @supports_net_changes = 0, @capture_instance =
'dbo_customers_v2';
GO
```

3. Insert new data into the **customers** table by running the following query:

```
INSERT INTO customers(first_name,last_name,email,phone_number) VALUES
('John','Doe','john.doe@example.com', '+1-555-123456');
GO
```

The Kafka Connect log reports on configuration updates through entries similar to the following message:

```
connect_1    | 2019-01-17 10:11:14,924 INFO   ||  Multiple capture instances present for the
same table: Capture instance "dbo_customers" [sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_CT, startLsn=00000024:00000d98:0036,
changeTableObjectId=1525580473, stopLsn=00000025:00000ef8:0048] and Capture
instance "dbo_customers_v2" [sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
connect_1    | 2019-01-17 10:11:14,924 INFO   ||  Schema will be changed for ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
...
connect_1    | 2019-01-17 10:11:33,719 INFO   ||  Migrating schema to ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
```

Eventually, the **phone_number** field is added to the schema and its value appears in messages written to the Kafka topic.

```
...
    {
      "type": "string",
      "optional": true,
      "field": "phone_number"
    }
```

```
...
    "after": {
      "id": 1005,
      "first_name": "John",
      "last_name": "Doe",
      "email": "john.doe@example.com",
      "phone_number": "+1-555-123456"
    },
```

4. Drop the old capture instance by running the **sys.sp_cdc_disable_table** stored procedure.

```
EXEC sys.sp_cdc_disable_table @source_schema = 'dbo', @source_name =
'dbo_customers', @capture_instance = 'dbo_customers';
GO
```

8.6. MONITORING DEBEZIUM SQL SERVER CONNECTOR PERFORMANCE

The Debezium SQL Server connector provides three types of metrics that are in addition to the built-in support for JMX metrics that Zookeeper, Kafka, and Kafka Connect provide. The connector provides the following metrics:

- Snapshot metrics for monitoring the connector when performing snapshots.
- Streaming metrics for monitoring the connector when reading CDC table data.
- Schema history metrics for monitoring the status of the connector's schema history.

For information about how to expose the preceding metrics through JMX, see the Debezium monitoring documentation.

8.6.1. Debezium SQL Server connector snapshot metrics

The **MBean** is **debezium.sql_server:type=connector-metrics,server=*<sqlserver.server.name>*,task=*<task.id>*,context=snapshot**. Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.

Attributes	Type	Description
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the **CapturedTables** metric instead.*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.

Attributes	Type	Description
RowsScanned	Map<String, Long>	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The connector also provides the following additional snapshot metrics when an incremental snapshot is executed:

Attributes	Type	Description
ChunkId	string	The identifier of the current snapshot chunk.
ChunkFrom	string	The lower bound of the primary key set defining the current chunk.
ChunkTo	string	The upper bound of the primary key set defining the current chunk.
TableFrom	string	The lower bound of the primary key set of the currently snapshotted table.
TableTo	string	The upper bound of the primary key set of the currently snapshotted table.

8.6.2. Debezium SQL Server connector streaming metrics

The **MBean** is **debezium.sql_server:type=connector-metrics,server=*<sqlserver.server.name>*,task=*<task.id>*,context=streaming**. The following table lists the streaming metrics that are available.

Attributes	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since the last start or metrics reset.
TotalNumberOfCreateEventsSeen	**long**	The total number of create events that this connector has seen since the last start or metrics reset.
TotalNumberOfUpdateEventsSeen	**long**	The total number of update events that this connector has seen since the last start or metrics reset.
TotalNumberOfDeleteEventsSeen	**long**	The total number of delete events that this connector has seen since the last start or metrics reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables *Deprecated and scheduled for removal in a future release; use the 'CapturedTables' metric instead*	**string[]**	The list of tables that are monitored by the connector.
CapturedTables	**string[]**	The list of tables that are captured by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.

Attributes	Type	Description
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	**long**	The maximum buffer of the queue in bytes. This metric is available if **max.queue.size.in.bytes** is set to a positive long value.
CurrentQueueSizeInBytes	**long**	The current volume, in bytes, of records in the queue.

8.6.3. Debezium SQL Server connector schema history metrics

The MBean is **debezium.sql_server:type=connector-metrics,context=schema-history,server=*<sqlserver.server.name>***.

The following table lists the schema history metrics that are available.

Attributes	Type	Description
Status	**string**	One of **STOPPED**, **RECOVERING** (recovering history from the storage), **RUNNING** describing the state of the database history.
RecoveryStartTime	**long**	The time in epoch seconds at what recovery has started.
ChangesRecovered	**long**	The number of changes that were read during recovery phase.
ChangesApplied	**long**	the total number of schema changes applied during recovery and runtime.
MilliSecondsSinceLast RecoveredChange	**long**	The number of milliseconds that elapsed since the last change was recovered from the history store.
MilliSecondsSinceLastAppliedChange	**long**	The number of milliseconds that elapsed since the last change was applied.
LastRecoveredChange	**string**	The string representation of the last change recovered from the history store.
LastAppliedChange	**string**	The string representation of the last applied change.

CHAPTER 9. MONITORING DEBEZIUM

You can use the JMX metrics provided by Apache Zookeeper, Apache Kafka, and Kafka Connect to monitor Debezium. To use these metrics, you must enable them when you start the Zookeeper, Kafka, and Kafka Connect services. Enabling JMX involves setting the correct environment variables.

NOTE

If you are running multiple services on the same machine, be sure to use distinct JMX ports for each service.

9.1. METRICS FOR MONITORING DEBEZIUM CONNECTORS

In addition to the built-in support for JMX metrics in Kafka, Zookeeper, and Kafka Connect, each connector provides additional metrics that you can use to monitor their activities.

- MySQL connector metrics
- MongoDB connector metrics
- PostgreSQL connector metrics
- SQL Server connector metrics
- Db2 connector metrics

9.2. ENABLING JMX IN LOCAL INSTALLATIONS

With Zookeeper, Kafka, and Kafka Connect, you enable JMX by setting the appropriate environment variables when you start each service.

9.2.1. Zookeeper JMX environment variables

Zookeeper has built-in support for JMX. When running Zookeeper using a local installation, the **zkServer.sh** script recognizes the following environment variables:

JMXPORT

Enables JMX and specifies the port number that will be used for JMX. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.port=$JMXPORT**.

JMXAUTH

Whether JMX clients must use password authentication when connecting. Must be either **true** or **false**. The default is **false**. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.authenticate=$JMXAUTH**.

JMXSSL

Whether JMX clients connect using SSL/TLS. Must be either **true** or **false**. The default is **false**. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.ssl=$JMXSSL**.

JMXLOG4J

Whether the Log4J JMX MBeans should be disabled. Must be either **true** (default) or **false**. The default is **true**. The value is used to specify the JVM parameter **-Dzookeeper.jmx.log4j.disable=$JMXLOG4J**.

9.2.2. Kafka JMX environment variables

When running Kafka using a local installation, the **kafka-server-start.sh** script recognizes the following environment variables:

JMX_PORT

Enables JMX and specifies the port number that will be used for JMX. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.port=$JMX_PORT**.

KAFKA_JMX_OPTS

The JMX options, which are passed directly to the JVM during startup. The default options are:

- **-Dcom.sun.management.jmxremote**
- **-Dcom.sun.management.jmxremote.authenticate=false**
- **-Dcom.sun.management.jmxremote.ssl=false**

9.2.3. Kafka Connect JMX environment variables

When running Kafka using a local installation, the **connect-distributed.sh** script recognizes the following environment variables:

JMX_PORT

Enables JMX and specifies the port number that will be used for JMX. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.port=$JMX_PORT**.

KAFKA_JMX_OPTS

The JMX options, which are passed directly to the JVM during startup. The default options are:

- **-Dcom.sun.management.jmxremote**
- **-Dcom.sun.management.jmxremote.authenticate=false**
- **-Dcom.sun.management.jmxremote.ssl=false**

9.3. MONITORING DEBEZIUM ON OPENSHIFT

If you are using Debezium on OpenShift, you can obtain JMX metrics by opening a JMX port on **9999**. For more information, see JMX Options in Using AMQ Streams on OpenShift.

In addition, you can use Prometheus and Grafana to monitor the JMX metrics. For more information, see Introducing Metrics to Kafka , in Deploying and Upgrading AMQ Streams on OpenShift.

CHAPTER 10. DEBEZIUM LOGGING

Debezium has extensive logging built into its connectors, and you can change the logging configuration to control which of these log statements appear in the logs and where those logs are sent. Debezium (as well as Kafka, Kafka Connect, and Zookeeper) use the Log4j logging framework for Java.

By default, the connectors produce a fair amount of useful information when they start up, but then produce very few logs when the connector is keeping up with the source databases. This is often sufficient when the connector is operating normally, but may not be enough when the connector is behaving unexpectedly. In such cases, you can change the logging level so that the connector generates much more verbose log messages describing what the connector is doing and what it is not doing.

10.1. DEBEZIUM LOGGING CONCEPTS

Before configuring logging, you should understand what Log4J *loggers*, *log levels*, and *appenders* are.

Loggers

Each log message produced by the application is sent to a specific *logger* (for example, **io.debezium.connector.mysql**). Loggers are arranged in hierarchies. For example, the **io.debezium.connector.mysql** logger is the child of the **io.debezium.connector** logger, which is the child of the **io.debezium** logger. At the top of the hierarchy, the *root logger* defines the default logger configuration for all of the loggers beneath it.

Log levels

Every log message produced by the application also has a specific *log level*:

1. **ERROR** - errors, exceptions, and other significant problems
2. **WARN** - *potential* problems and issues
3. **INFO** - status and general activity (usually low-volume)
4. **DEBUG** - more detailed activity that would be useful in diagnosing unexpected behavior
5. **TRACE** - very verbose and detailed activity (usually very high-volume)

Appenders

An *appender* is essentially a destination where log messages are written. Each appender controls the format of its log messages, giving you even more control over what the log messages look like.

To configure logging, you specify the desired level for each logger and the appender(s) where those log messages should be written. Since loggers are hierarchical, the configuration for the root logger serves as a default for all of the loggers below it, although you can override any child (or descendant) logger.

10.2. DEFAULT DEBEZIUM LOGGING CONFIGURATION

If you are running Debezium connectors in a Kafka Connect process, then Kafka Connect uses the Log4j configuration file (for example, **/opt/kafka/config/connect-log4j.properties**) in the Kafka installation. By default, this file contains the following configuration:

connect-log4j.properties

```
log4j.rootLogger=INFO, stdout (1)
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender (2)
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout (3)
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n (4)
...
```

(1) The root logger, which defines the default logger configuration. By default, loggers include **INFO**, **WARN**, and **ERROR** messages. These log messages are written to the **stdout** appender.

(2) The **stdout** appender writes log messages to the console (as opposed to a file).

(3) The **stdout** appender uses a pattern matching algorithm to format the log messages.

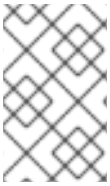
(4) The pattern for the **stdout** appender (see the Log4j documentation for details).

Unless you configure other loggers, all of the loggers that Debezium uses inherit the **rootLogger** configuration.

10.3. CONFIGURING DEBEZIUM LOGGING

By default, Debezium connectors write all **INFO**, **WARN**, and **ERROR** messages to the console. You can change the default logging configuration by using one of the following methods:

- Setting the logging level by configuring loggers
- Dynamically setting the logging level with the Kafka Connect REST API
- Setting the logging level by adding mapped diagnostic contexts



NOTE

There are other methods that you can use to configure Debezium logging with Log4j. For more information, search for tutorials about setting up and using appenders to send log messages to specific destinations.

10.3.1. Changing the Debezium logging level by configuring loggers

The default Debezium logging level provides sufficient information to show whether a connector is healthy or not. However, if a connector is not healthy, you can change its logging level to troubleshoot the issue.

In general, Debezium connectors send their log messages to loggers with names that match the fully-qualified name of the Java class that is generating the log message. Debezium uses packages to organize code with similar or related functions. This means that you can control all of the log messages for a specific class or for all of the classes within or under a specific package.

Procedure

1. Open the **log4j.properties** file.

2. Configure a logger for the connector.
   This example configures loggers for the MySQL connector and the database history implementation used by the connector, and sets them to log **DEBUG** level messages:

   log4j.properties

   ```
   ...
   log4j.logger.io.debezium.connector.mysql=DEBUG, stdout (1)
   log4j.logger.io.debezium.relational.history=DEBUG, stdout (2)

   log4j.additivity.io.debezium.connector.mysql=false (3)
   log4j.additivity.io.debezium.relational.history=false (4)
   ...
   ```

   (1) Configures the logger named **io.debezium.connector.mysql** to send **DEBUG**, **INFO**, **WARN**, and **ERROR** messages to the **stdout** appender.

   (2) Configures the logger named **io.debezium.relational.history** to send **DEBUG**, **INFO**, **WARN**, and **ERROR** messages to the **stdout** appender.

   (3) (4) Turns off *additivity*, which results in log messages not being sent to the appenders of parent loggers (this can prevent seeing duplicate log messages when using multiple appenders).

3. If necessary, change the logging level for a specific subset of the classes within the connector.
   Increasing the logging level for the entire connector increases the log verbosity, which can make it difficult to understand what is happening. In these cases, you can change the logging level just for the subset of classes that are related to the issue that you are troubleshooting.

   a. Set the connector's logging level to either **DEBUG** or **TRACE**.

   b. Review the connector's log messages.
      Find the log messages that are related to the issue that you are troubleshooting. The end of each log message shows the name of the Java class that produced the message.

   c. Set the connector's logging level back to **INFO**.

   d. Configure a logger for each Java class that you identified.
      For example, consider a scenario in which you are unsure why the MySQL connector is skipping some events when it is processing the binlog. Rather than turn on **DEBUG** or **TRACE** logging for the entire connector, you can keep the connector's logging level at **INFO** and then configure **DEBUG** or **TRACE** on just the class that is reading the binlog:

      log4j.properties

      ```
      ...
      log4j.logger.io.debezium.connector.mysql=INFO, stdout
      log4j.logger.io.debezium.connector.mysql.BinlogReader=DEBUG, stdout
      log4j.logger.io.debezium.relational.history=INFO, stdout

      log4j.additivity.io.debezium.connector.mysql=false
      log4j.additivity.io.debezium.relational.history=false
      log4j.additivity.io.debezium.connector.mysql.BinlogReader=false
      ...
      ```

10.3.2. Dynamically changing the Debezium logging level with the Kafka Connect API

You can use the Kafka Connect REST API to set logging levels for a connector dynamically at runtime. Unlike log level changes that you set in **log4j.properties**, changes that you make via the API take effect immediately, and do not require you to restart the worker.

The log level setting that you specify in the API applies only to the worker at the endpoint that receives the request. The log levels of other workers in the cluster remain unchanged.

The specified level is not persisted after the worker restarts. To make persistent changes to the logging level, set the log level in **log4j.properties** by configuring loggers or adding mapped diagnostic contexts.

Procedure

- Set the log level by sending a PUT request to the **admin/loggers** endpoint that specifies the following information:
  - The package for which you want to change the log level.
  - The log level that you want to set.

    ```
    curl -s -X PUT -H "Content-Type:application/json"
    http://localhost:8083/admin/loggers/io.debezium.connector.<connector_package> -d
    '{"level": "<log_level>"}'
    ```

    For example, to log debug information for a Debezium MySQL connector, send the following request to Kafka Connect:

    ```
    curl -s -X PUT -H "Content-Type:application/json"
    http://localhost:8083/admin/loggers/io.debezium.connector.mysql -d '{"level": "DEBUG"}'
    ```

10.3.3. Changing the Debezium logging levely by adding mapped diagnostic contexts

Most Debezium connectors (and the Kafka Connect workers) use multiple threads to perform different activities. This can make it difficult to look at a log file and find only those log messages for a particular logical activity. To make the log messages easier to find, Debezium provides several *mapped diagnostic contexts* (MDC) that provide additional information for each thread.

Debezium provides the following MDC properties:

dbz.connectorType

A short alias for the type of connector. For example, **MySql**, **Mongo**, **Postgres**, and so on. All threads associated with the same *type* of connector use the same value, so you can use this to find all log messages produced by a given type of connector.

dbz.connectorName

The name of the connector or database server as defined in the connector's configuration. For example **products**, **serverA**, and so on. All threads associated with a specific *connector instance* use the same value, so you can find all of the log messages produced by a specific connector instance.

dbz.connectorContext

A short name for an activity running as a separate thread running within the connector's task. For example, **main**, **binlog**, **snapshot**, and so on. In some cases, when a connector assigns threads to specific resources (such as a table or collection), the name of that resource could be used instead.

Each thread associated with a connector would use a distinct value, so you can find all of the log messages associated with this particular activity.

To enable MDC for a connector, you configure an appender in the **log4j.properties** file.

Procedure

1. Open the **log4j.properties** file.
2. Configure an appender to use any of the supported Debezium MDC properties.
   In the following example, the **stdout** appender is configured to use these MDC properties:

   log4j.properties

   ```
   ...
   log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} %-5p
   %X{dbz.connectorType}|%X{dbz.connectorName}|%X{dbz.connectorContext}  %m   [%c]%n
   ...
   ```

   The configuration in the preceding example produces log messages similar to the ones in the following output:

   ```
   ...
   2017-02-07 20:49:37,692 INFO   MySQL|dbserver1|snapshot  Starting snapshot for
   jdbc:mysql://mysql:3306/?
   useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnicode=true
   &characterEncoding=UTF-8&characterSetResults=UTF-
   8&zeroDateTimeBehavior=convertToNull with user 'debezium'
   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,696 INFO   MySQL|dbserver1|snapshot  Snapshot is using user
   'debezium' with these MySQL grants:   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,697 INFO   MySQL|dbserver1|snapshot   GRANT SELECT, RELOAD,
   SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO
   'debezium'@'%'   [io.debezium.connector.mysql.SnapshotReader]
   ...
   ```

   Each line in the log includes the connector type (for example, **MySQL**), the name of the connector (for example, **dbserver1**), and the activity of the thread (for example, **snapshot**).

10.4. DEBEZIUM LOGGING ON OPENSHIFT

If you are using Debezium on OpenShift, you can use the Kafka Connect loggers to configure the Debezium loggers and logging levels. For more information about configuring logging properties in a Kafka Connect schema, see Using AMQ Streams on OpenShift .

CHAPTER 11. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION

When the default Debezium connector behavior is not right for your application, you can use the following Debezium features to configure the behavior you need.

Kafka Connect automatic topic creation

Enables Connect to create topics at runtime, and apply configuration settings to those topics based on their names.

Avro serialization

Support for configuring Debezium PostgreSQL, MongoDB, or SQL Server connectors to use Avro to serialize message keys and value, making it easier for change event record consumers to adapt to a changing record schema.

CloudEvents converter

Enables a Debezium connector to emit change event records that conform to the CloudEvents specification.

Sending signals to a Debezium connector

Provides a way to modify the behavior of a connector, or trigger an action, such as initiating an ad hoc snapshot.

11.1. CUSTOMIZATION OF KAFKA CONNECT AUTOMATIC TOPIC CREATION

Kafka provides two mechanisms for creating topics automatically. You can enable automatic topic creation for the Kafka broker, and, beginning with Kafka 2.6.0, you can also enable Kafka Connect to create topics. The Kafka broker uses the **auto.create.topics.enable** property to control automatic topic creation. In Kafka Connect, the **topic.creation.enable** property specifies whether Kafka Connect is permitted to create topics. In both cases, the default settings for the properties enables automatic topic creation.

When automatic topic creation is enabled, if a Debezium source connector emits a change event record for a table for which no target topic already exists, the topic is created at runtime as the event record is ingested into Kafka.

Differences between automatic topic creation at the broker and in Kafka Connect

Topics that the broker creates are limited to sharing a single default configuration. The broker cannot apply unique configurations to different topics or sets of topics. By contrast, Kafka Connect can apply any of several configurations when creating topics, setting the replication factor, number of partitions, and other topic-specific settings as specified in the Debezium connector configuration. The connector configuration defines a set of topic creation groups, and associates a set of topic configuration properties with each group.

The broker configuration and the Kafka Connect configuration are independent of each other. Kafka Connect can create topics regardless of whether you disable topic creation at the broker. If you enable automatic topic creation at both the broker and in Kafka Connect, the Connect configuration takes precedence, and the broker creates topics only if none of the settings in the Kafka Connect configuration apply.

See the following topics for more information:

- Section 11.1.1, “Disabling automatic topic creation for the Kafka broker”

- Section 11.1.2, “Configuring automatic topic creation in Kafka Connect”
- Section 11.1.3, “Configuration of automatically created topics”
- Section 11.1.3.1, “Topic creation groups”
- Section 11.1.3.2, “Topic creation group configuration properties”
- Section 11.1.3.3, “Specifying the configuration for the Debezium default topic creation group”
- Section 11.1.3.4, “Specifying the configuration for Debezium custom topic creation groups”
- Section 11.1.3.5, “Registering Debezium custom topic creation groups”

11.1.1. Disabling automatic topic creation for the Kafka broker

By default, the Kafka broker configuration enables the broker to create topics at runtime if the topics do not already exist. Topics created by the broker cannot be configured with custom properties. If you use a Kafka version earlier than 2.6.0, and you want to create topics with specific configurations, you must to disable automatic topic creation at the broker, and then explicitly create the topics, either manually, or through a custom deployment process.

Procedure

- In the broker configuration, set the value of **auto.create.topics.enable** to **false**.

11.1.2. Configuring automatic topic creation in Kafka Connect

Automatic topic creation in Kafka Connect is controlled by the **topic.creation.enable** property. The default value for the property is **true**, enabling automatic topic creation, as shown in the following example:

```
topic.creation.enable = true
```

The setting for the **topic.creation.enable** property applies to all workers in the Connect cluster.

Kafka Connect automatic topic creation requires you to define the configuration properties that Kafka Connect applies when creating topics. You specify topic configuration properties in the Debezium connector configuration by defining topic groups, and then specifying the properties to apply to each group. The connector configuration defines a default topic creation group, and, optionally, one or more custom topic creation groups. Custom topic creation groups use lists of topic name patterns to specify the topics to which the group’s settings apply.

For details about how Kafka Connect matches topics to topic creation groups, see Topic creation groups. For more information about how configuration properties are assigned to groups, see Topic creation group configuration properties.

By default, topics that Kafka Connect creates are named based on the pattern **server.schema.table**, for example, **dbserver.myschema.inventory**.

Procedure

- To prevent Kafka Connect from creating topics automatically, set the value of **topic.creation.enable** to **false** in the Kafka Connect custom resource, as in the following example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster

...

spec:
  config:
    topic.creation.enable: "false"
```

NOTE

Kafka Connect automatic topic creation requires the **replication.factor** and **partitions** properties to be set for at least the **default** topic creation group. It is valid for groups to obtain the values for the required properties from the default values for the Kafka broker.

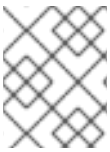
11.1.3. Configuration of automatically created topics

For Kafka Connect to create topics automatically, it requires information from the source connector about the configuration properties to apply when creating topics. You define the properties that control topic creation in the configuration for each Debezium connector. As Kafka Connect creates topics for event records that a connector emits, the resulting topics obtain their configuration from the applicable group. The configuration applies to event records emitted by that connector only.

11.1.3.1. Topic creation groups

A set of topic properties is associated with a topic creation group. Minimally, you must define a **default** topic creation group and specify its configuration properties. Beyond that you can optionally define one or more custom topic creation groups and specify unique properties for each.

When you create custom topic creation groups, you define the member topics for each group based on topic name patterns. You can specify naming patterns that describe the topics to include or exclude from each group. The **include** and **exclude** properties contain comma-separated lists of regular expressions that define topic name patterns. For example, if you want a group to include all topics that start with the string **dbserver1.inventory**, set the value of its **topic.creation.inventory.include** property to **dbserver1\\.inventory\\.***.



NOTE

If you specify both **include** and **exclude** properties for a custom topic group, the exclusion rules take precedence, and override the inclusion rules.

11.1.3.2. Topic creation group configuration properties

The **default** topic creation group and each custom group is associated with a unique set of configuration properties. You can configure a group to include any of the Kafka topic-level configuration properties. For example, you can specify the cleanup policy for old topic segments, retention time, or the topic compression type for a topic group. You must define at least a minimum set of properties to describe the configuration of the topics to be created.

If no custom groups are registered, or if the **include** patterns for any registered groups don't match the names of any topics to be created, then Kafka Connect uses the configuration of the **default** group to create topics.

For general information about configuring topics, see Kafka topic creation recommendations in Installing Debezium on OpenShift.

11.1.3.3. Specifying the configuration for the Debezium default topic creation group

Before you can use Kafka Connect automatic topic creation, you must create a default topic creation group and define a configuration for it. The configuration for the default topic creation group is applied to any topics with names that do not match the **include** list pattern of a custom topic creation group.

Prerequisites

- In the Kafka Connect custom resource, the **use-connector-resources** value in **metadata.annotations** specifies that the cluster Operator uses KafkaConnector custom resources to configure connectors in the cluster. For example:

```
...
  metadata:
    name: my-connect-cluster
    annotations: strimzi.io/use-connector-resources: "true"
...
```

Procedure

- To define properties for the **topic.creation.default** group, add them to **spec.config** in the connector custom resource, as shown in the following example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
...

  config:
...
    topic.creation.default.replication.factor: 3 (1)
    topic.creation.default.partitions: 10 (2)
    topic.creation.default.cleanup.policy: compact (3)
    topic.creation.default.compression.type: lz4 (4)
...
```

You can include any Kafka topic-level configuration property in the configuration for the **default** group.

Table 11.1. Connector configuration for the default topic creation group

Item	Description

Item	Description
1	**topic.creation.default.replication.factor** defines the replication factor for topics created by the default group. **replication.factor** is mandatory for the **default** group but optional for custom groups. Custom groups will fall back to the **default** group's value if not set. Use **-1** to use the Kafka broker's default value.
2	**topic.creation.default.partitions** defines the number of partitions for topics created by the default group. **partitions** is mandatory for the **default** group but optional for custom groups. Custom groups will fall back to the **default** group's value if not set. Use **-1** to use the Kafka broker's default value.
3	**topic.creation.default.cleanup.policy** is mapped to the **cleanup.policy** property of the topic level configuration parameters and defines the log retention policy.
4	**topic.creation.default.compression.type** is mapped to the **compression.type** property of the topic level configuration parameters and defines how messages are compressed on hard disk.

NOTE

Custom groups fall back to the **default** group settings only for the required **replication.factor** and **partitions** properties. If the configuration for a custom topic group leaves other properties undefined, the values specified in the **default** group are not applied.

11.1.3.4. Specifying the configuration for Debezium custom topic creation groups

You can define multiple custom topic groups, each with its own configuration.

Procedure

- To define a custom topic group, add a **topic.creation.*<group_name>*.include** property to **spec.config** in the connector custom resource, followed by the configuration properties that you want to apply to topics in the custom group.
  The following example shows an excerpt of a custom resource that defines the custom topic creation groups **inventory** and **applicationlogs**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
...
spec:
...

  config:
... 1
  topic.creation.inventory.include: dbserver1\\.inventory\\.* 2
  topic.creation.inventory.partitions: 20
```

```
    topic.creation.inventory.cleanup.policy: compact
    topic.creation.inventory.delete.retention.ms: 7776000000

    3
    topic.creation.applicationlogs.include: dbserver1\\.logs\\.applog-.* 4
    topic.creation.applicationlogs.exclude": dbserver1\\.logs\\.applog-old-.* 5
    topic.creation.applicationlogs.replication.factor: 1
    topic.creation.applicationlogs.partitions: 20
    topic.creation.applicationlogs.cleanup.policy: delete
    topic.creation.applicationlogs.retention.ms: 7776000000
    topic.creation.applicationlogs.compression.type: lz4
...
...
```

Table 11.2. Connector configuration for custom inventory and applicationlogs topic creation groups

Item	Description
1	Defines the configuration for the **inventory** group. The **replication.factor** and **partitions** properties are optional for custom groups. If no value is set, custom groups fall back to the value set for the **default** group. Set the value to **-1** to use the value that is set for the Kafka broker.
2	**topic.creation.inventory.include** defines a regular expression to match all topics that start with **dbserver1.inventory.**. The configuration that is defined for the **inventory** group is applied only to topics with names that match the specified regular expression.
3	Defines the configuration for the **applicationlogs** group. The **replication.factor** and **partitions** properties are optional for custom groups. If no value is set, custom groups fall back to the value set for the **default** group. Set the value to **-1** to use the value that is set for the Kafka broker.
4	**topic.creation.applicationlogs.include** defines a regular expression to match all topics that start with **dbserver1.logs.applog-**. The configuration that is defined for the **applicationlogs** group is applied only to topics with names that match the specified regular expression. Because an **exclude** property is also defined for this group, the topics that match the **include** regular expression might be further restricted by the that **exclude** property.
5	**topic.creation.applicationlogs.exclude** defines a regular expression to match all topics that start with **dbserver1.logs.applog-old-**. The configuration that is defined for the **applicationlogs** group is applied only to topics with name that do **not** match the given regular expression. Because an **include** property is also defined for this group, the configuration of the **applicationlogs** group is applied only to topics with names that match the specified **include** regular expressions **and** that do *not* match the specified **exclude** regular expressions.

11.1.3.5. Registering Debezium custom topic creation groups

After you specify the configuration for any custom topic creation groups, register the groups.

Procedure

- Register custom groups by adding the **topic.creation.groups** property to the connector custom resource, and specifying a comma-separated list of custom topic creation groups. The following excerpt from a connector custom resource registers the custom topic creation groups **inventory** and **applicationlogs**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
...
spec:
...

    config:
      topic.creation.groups: inventory,applicationlogs

...
```

Completed configuration

The following example shows a completed configuration that includes the configuration for a **default** topic group, along with the configurations for an **inventory** and an **applicationlogs** custom topic creation group:

Example: Configuration for a default topic creation group and two custom groups

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
...
spec:
...

    config:
...
    topic.creation.default.replication.factor: 3,
    topic.creation.default.partitions: 10,
    topic.creation.default.cleanup.policy: compact
    topic.creation.default.compression.type: lz4
    topic.creation.groups: inventory,applicationlogs
    topic.creation.inventory.include: dbserver1\\.inventory\\.*
    topic.creation.inventory.partitions: 20
    topic.creation.inventory.cleanup.policy: compact
    topic.creation.inventory.delete.retention.ms: 7776000000
    topic.creation.applicationlogs.include: dbserver1\\.logs\\.applog-.*
    topic.creation.applicationlogs.exclude": dbserver1\\.logs\\.applog-old-.*
    topic.creation.applicationlogs.replication.factor: 1
    topic.creation.applicationlogs.partitions: 20
    topic.creation.applicationlogs.cleanup.policy: delete
    topic.creation.applicationlogs.retention.ms: 7776000000
    topic.creation.applicationlogs.compression.type: lz4
...
```

11.2. CONFIGURING DEBEZIUM CONNECTORS TO USE AVRO SERIALIZATION

A Debezium connector works in the Kafka Connect framework to capture each row-level change in a database by generating a change event record. For each change event record, the Debezium connector completes the following actions:

1. Applies configured transformations.
2. Serializes the record key and value into a binary form by using the configured Kafka Connect converters.
3. Writes the record to the correct Kafka topic.

You can specify converters for each individual Debezium connector instance. Kafka Connect provides a JSON converter that serializes the record keys and values into JSON documents. The default behavior is that the JSON converter includes the record's message schema, which makes each record very verbose. The Getting Started with Debezium guide shows what the records look like when both payload and schemas are included. If you want records to be serialized with JSON, consider setting the following connector configuration properties to **false**:

- **key.converter.schemas.enable**
- **value.converter.schemas.enable**

Setting these properties to **false** excludes the verbose schema information from each record.

Alternatively, you can serialize the record keys and values by using Apache Avro. The Avro binary format is compact and efficient. Avro schemas make it possible to ensure that each record has the correct structure. Avro's schema evolution mechanism enables schemas to evolve. This is essential for Debezium connectors, which dynamically generate each record's schema to match the structure of the database table that was changed. Over time, change event records written to the same Kafka topic might have different versions of the same schema. Avro serialization makes it easier for the consumers of change event records to adapt to a changing record schema.

To use Apache Avro serialization, you must deploy a schema registry that manages Avro message schemas and their versions. For information about setting up this registry, see the documentation for Installing and deploying Red Hat build of Apicurio Registry on OpenShift .

11.2.1. About the Apicurio Registry

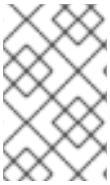
Red Hat build of Apicurio Registry provides the following components that work with Avro:

- An Avro converter that you can specify in Debezium connector configurations. This converter maps Kafka Connect schemas to Avro schemas. The converter then uses the Avro schemas to serialize the record keys and values into Avro's compact binary form.
- An API and schema registry that tracks:
  - Avro schemas that are used in Kafka topics.
  - Where the Avro converter sends the generated Avro schemas.

  Because the Avro schemas are stored in this registry, each record needs to contain only a tiny *schema identifier*. This makes each record even smaller. For an I/O bound system like Kafka, this means more total throughput for producers and consumers.

- Avro *Serdes* (serializers and deserializers) for Kafka producers and consumers. Kafka consumer applications that you write to consume change event records can use Avro Serdes to deserialize the change event records.

To use the Apicurio Registry with Debezium, add Apicurio Registry converters and their dependencies to the Kafka Connect container image that you are using for running a Debezium connector.



NOTE

The Apicurio Registry project also provides a JSON converter. This converter combines the advantage of less verbose messages with human-readable JSON. Messages do not contain the schema information themselves, but only a schema ID.

NOTE

To use converters provided by Apicurio Registry you need to provide **apicurio.registry.url**.

11.2.2. Overview of deploying a Debezium connector that uses Avro serialization

To deploy a Debezium connector that uses Avro serialization, you must complete three main tasks:

1. Deploy a Red Hat build of Apicurio Registry instance by following the instructions in Installing and deploying Red Hat build of Apicurio Registry on OpenShift.
2. Install the Avro converter by downloading the Debezium Service Registry Kafka Connect zip file and extracting it into the Debezium connector's directory.
3. Configure a Debezium connector instance to use Avro serialization by setting configuration properties as follows:

```
key.converter=io.apicurio.registry.utils.converter.AvroConverter
key.converter.apicurio.registry.url=http://apicurio:8080/apis/registry/v2
key.converter.apicurio.registry.auto-register=true
key.converter.apicurio.registry.find-latest=true
value.converter=io.apicurio.registry.utils.converter.AvroConverter
value.converter.apicurio.registry.url=http://apicurio:8080/apis/registry/v2
value.converter.apicurio.registry.auto-register=true
value.converter.apicurio.registry.find-latest=true
```

Internally, Kafka Connect always uses JSON key/value converters for storing configuration and offsets.

11.2.3. Deploying connectors that use Avro in Debezium containers

In your environment, you might want to use a provided Debezium container to deploy Debezium connectors that use Avro serialization. Complete the following procedure to build a custom Kafka Connect container image for Debezium, and configure the Debezium connector to use the Avro converter.

Prerequisites

- You have Docker installed and sufficient rights to create and manage containers.

- You downloaded the Debezium connector plug-in(s) that you want to deploy with Avro serialization.

Procedure

1. Deploy an instance of Apicurio Registry. See Installing and deploying Red Hat build of Apicurio Registry on OpenShift, which provides instructions for:
   - Installing Apicurio Registry
   - Installing AMQ Streams
   - Setting up AMQ Streams storage
2. Extract the Debezium connector archives to create a directory structure for the connector plug-ins. If you downloaded and extracted the archives for multiple Debezium connectors, the resulting directory structure looks like the one in the following example:

```
tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── ...
├── debezium-connector-mysql
│   ├── ...
├── debezium-connector-postgres
│   ├── ...
└── debezium-connector-sqlserver
    ├── ...
```

3. Add the Avro converter to the directory that contains the Debezium connector that you want to configure to use Avro serialization:
   a. Go to the Red Hat build of Debezium download site and download the Apicurio Registry Kafka Connect zip file.
   b. Extract the archive into the desired Debezium connector directory.

   To configure more than one type of Debezium connector to use Avro serialization, extract the archive into the directory for each relevant connector type. Although extracting the archive to each directory duplicates the files, by doing so you remove the possibility of conflicting dependencies.
4. Create and publish a custom image for running Debezium connectors that are configured to use the Avro converter:
   a. Create a new **Dockerfile** by using **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** as the base image. In the following example, replace *my-plugins* with the name of your plug-ins directory:

```
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

   Before Kafka Connect starts running the connector, Kafka Connect loads any third-party plug-ins that are in the **/opt/kafka/plugins** directory.

b. Build the docker container image. For example, if you saved the docker file that you created in the previous step as **debezium-container-with-avro**, then you would run the following command:
docker build -t debezium-container-with-avro:latest

c. Push your custom image to your container registry, for example:
docker push *<myregistry.io>*/debezium-container-with-avro:latest

d. Point to the new container image. Do one of the following:

- Edit the **KafkaConnect.spec.image** property of the **KafkaConnect** custom resource. If set, this property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator. For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  image: debezium-container-with-avro
```

- In the **install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml** file, edit the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable to point to the new container image and reinstall the Cluster Operator. If you edit this file you will need to apply it to your OpenShift cluster.

5. Deploy each Debezium connector that is configured to use the Avro converter. For each Debezium connector:

a. Create a Debezium connector instance. The following **inventory-connector.yaml** file example creates a **KafkaConnector** custom resource that defines a MySQL connector instance that is configured to use the Avro converter:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 1
  config:
    database.hostname: mysql
    database.port: 3306
    database.user: debezium
    database.password: dbz
    database.server.id: 184054
    database.server.name: dbserver1
    database.include.list: inventory
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092
    database.history.kafka.topic: schema-changes.inventory
    key.converter: io.apicurio.registry.utils.converter.AvroConverter
    key.converter.apicurio.registry.url: http://apicurio:8080/api
    key.converter.apicurio.registry.global-id:
```

```
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
    value.converter: io.apicurio.registry.utils.converter.AvroConverter
    value.converter.apicurio.registry.url: http://apicurio:8080/api
    value.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
```

b. Apply the connector instance, for example:
oc apply -f inventory-connector.yaml

This registers **inventory-connector** and the connector starts to run against the **inventory** database.

6. Verify that the connector was created and has started to track changes in the specified database. You can verify the connector instance by watching the Kafka Connect log output as, for example, **inventory-connector** starts.

a. Display the Kafka Connect log output:

```
oc logs $(oc get pods -o name -l strimzi.io/name=my-connect-cluster-connect)
```

b. Review the log output to verify that the initial snapshot has been executed. You should see something like the following lines:

```
...
2020-02-21 17:57:30,801 INFO Starting snapshot for jdbc:mysql://mysql:3306/?
useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnicode=
true&characterEncoding=UTF-8&characterSetResults=UTF-
8&zeroDateTimeBehavior=CONVERT_TO_NULL&connectTimeout=30000 with user
'debezium' with locking mode 'minimal' (io.debezium.connector.mysql.SnapshotReader)
[debezium-mysqlconnector-dbserver1-snapshot]
2020-02-21 17:57:30,805 INFO Snapshot is using user 'debezium' with these MySQL
grants: (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-
dbserver1-snapshot]
...
```

Taking the snapshot involves a number of steps:

```
...
2020-02-21 17:57:30,822 INFO Step 0: disabling autocommit, enabling repeatable read
transactions, and setting lock wait timeout to 10
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,836 INFO Step 1: flush and obtain global read lock to prevent
writes to database (io.debezium.connector.mysql.SnapshotReader) [debezium-
mysqlconnector-dbserver1-snapshot]
2020-02-21 17:57:30,839 INFO Step 2: start transaction with consistent snapshot
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,840 INFO Step 3: read binlog position of MySQL primary server
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,843 INFO   using binlog 'mysql-bin.000003' at position '154' and gtid
'' (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...
```

```
2020-02-21 17:57:34,423 INFO Step 9: committing transaction
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:34,424 INFO Completed snapshot in 00:00:03.632
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...
```

After completing the snapshot, Debezium begins tracking changes in, for example, the **inventory** database's **binlog** for change events:

```
...
2020-02-21 17:57:35,584 INFO Transitioning from the snapshot reader to the binlog
reader (io.debezium.connector.mysql.ChainedReader) [task-thread-inventory-connector-
0]
2020-02-21 17:57:35,613 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [task-thread-inventory-connector-0]
2020-02-21 17:57:35,630 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [blc-mysql:3306]
Feb 21, 2020 5:57:35 PM com.github.shyiko.mysql.binlog.BinaryLogClient connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:5)
2020-02-21 17:57:35,775 INFO Connected to MySQL binlog at mysql:3306, starting at
binlog file 'mysql-bin.000003', pos=154, skipping 0 events plus 0 rows
(io.debezium.connector.mysql.BinlogReader) [blc-mysql:3306]
...
```

11.2.4. About Avro name requirements

As stated in the Avro documentation, names must adhere to the following rules:

- Start with **[A-Za-z_]**
- Subsequently contains only **[A-Za-z0-9_]** characters

Debezium uses the column's name as the basis for the corresponding Avro field. This can lead to problems during serialization if the column name does not also adhere to the Avro naming rules. Each Debezium connector provides a configuration property, **sanitize.field.names** that you can set to **true** if you have columns that do not adhere to Avro rules for names. Setting **sanitize.field.names** to **true** allows serialization of non-conformant fields without having to actually modify your schema.

11.3. EMITTING DEBEZIUM CHANGE EVENT RECORDS IN CLOUDEVENTS FORMAT

CloudEvents is a specification for describing event data in a common way. Its aim is to provide interoperability across services, platforms and systems. Debezium enables you to configure a MongoDB, MySQL, PostgreSQL, or SQL Server connector to emit change event records that conform to the CloudEvents specification.

IMPORTANT

Emitting change event records in CloudEvents format is a Technology Preview feature. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about support scope, see Technology Preview Features Support Scope.

The CloudEvents specification defines:

- A set of standardized event attributes
- Rules for defining custom attributes
- Encoding rules for mapping event formats to serialized representations such as JSON or Avro
- Protocol bindings for transport layers such as Apache Kafka, HTTP or AMQP

To configure a Debezium connector to emit change event records that conform to the CloudEvents specification, Debezium provides the **io.debezium.converters.CloudEventsConverter**, which is a Kafka Connect message converter.

Currently, only structured mapping mode is supported. The CloudEvents change event envelope can be JSON or Avro and each envelope type supports JSON or Avro as the **data** format. It is expected that a future Debezium release will support binary mapping mode.

Information about emitting change events in CloudEvents format is organized as follows:

- Section 11.3.1, "Example Debezium change event records in CloudEvents format"
- Section 11.3.2, "Example of configuring Debezium CloudEvents converter"
- Section 11.3.3, "Debezium CloudEvents converter configuration options"

For information about using Avro, see:

- Avro serialization
- Apicurio Registry

11.3.1. Example Debezium change event records in CloudEvents format

The following example shows what a CloudEvents change event record emitted by a PostgreSQL connector looks like. In this example, the PostgreSQL connector is configured to use JSON as the CloudEvents format envelope and also as the **data** format.

```
{
  "id" : "name:test_server;lsn:29274832;txId:565",  (1)
  "source" : "/debezium/postgresql/test_server",  (2)
  "specversion" : "1.0",  (3)
  "type" : "io.debezium.postgresql.datachangeevent",  (4)
  "time" : "2020-01-13T13:55:39.738Z",  (5)
```

```
  "datacontenttype" : "application/json",                    6
  "iodebeziumop" : "r",                                      7
  "iodebeziumversion" : "1.9.7.Final",                       8
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578923739738",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "565",
  "iodebeziumlsn" : "29274832",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "565",                                   9
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : {                                                 10
    "before" : null,
    "after" : {
      "pk" : 1,
      "name" : "Bob"
    }
  }
}
```

1 1 1 Unique ID that the connector generates for the change event based on the change event's content.

2 2 2 The source of the event, which is the logical name of the database as specified by the **database.server.name** property in the connector's configuration.

3 3 3 The CloudEvents specification version.

4 4 4 Connector type that generated the change event. The format of this field is **io.debezium.*CONNECTOR_TYPE*.datachangeevent**. The value of ***CONNECTOR_TYPE*** is **mongodb**, **mysql**, **postgresql**, or **sqlserver**.

5 5 Time of the change in the source database.

6 Describes the content type of the **data** attribute, which is JSON in this example. The only alternative is Avro.

7 An operation identifier. Possible values are **r** for read, **c** for create, **u** for update, or **d** for delete.

8 All **source** attributes that are known from Debezium change events are mapped to CloudEvents extension attributes by using the **iodebezium** prefix for the attribute name.

9 When enabled in the connector, each **transaction** attribute that is known from Debezium change events is mapped to a CloudEvents extension attribute by using the **iodebeziumtx** prefix for the attribute name.

10 The actual data change itself. Depending on the operation and the connector, the data might contain **before**, **after** and/or **patch** fields.

The following example also shows what a CloudEvents change event record emitted by a PostgreSQL

connector looks like. In this example, the PostgreSQL connector is again configured to use JSON as the CloudEvents format envelope, but this time the connector is configured to use Avro for the **data** format.

```
{
  "id" : "name:test_server;lsn:33227720;txId:578",
  "source" : "/debezium/postgresql/test_server",
  "specversion" : "1.0",
  "type" : "io.debezium.postgresql.datachangeevent",
  "time" : "2020-01-13T14:04:18.597Z",
  "datacontenttype" : "application/avro",                    1
  "dataschema" : "http://my-registry/schemas/ids/1",         2
  "iodebeziumop" : "r",
  "iodebeziumversion" : "1.9.7.Final",
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578924258597",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "578",
  "iodebeziumlsn" : "33227720",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "578",
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : "AAAAAAEAAgICAg=="                                3
}
```

1. Indicates that the **data** attribute contains Avro binary data.
2. URI of the schema to which the Avro data adheres.
3. The **data** attribute contains base64-encoded Avro binary data.

It is also possible to use Avro for the envelope as well as the **data** attribute.

11.3.2. Example of configuring Debezium CloudEvents converter

Configure **io.debezium.converters.CloudEventsConverter** in your Debezium connector configuration. The following example shows how to configure the CloudEvents converter to emit change event records that have the following characteristics:

- Use JSON as the envelope.
- Use the schema registry at **http://my-registry/schemas/ids/1** to serialize the **data** attribute as binary Avro data.

```
...
"value.converter": "io.debezium.converters.CloudEventsConverter",
"value.converter.serializer.type" : "json",                1
```

```
"value.converter.data.serializer.type" : "avro",
"value.converter.avro.schema.registry.url": "http://my-registry/schemas/ids/1"
...
```

1 Specifying the **serializer.type** is optional, because **json** is the default.

The CloudEvents converter converts Kafka record values. In the same connector configuration, you can specify **key.converter** if you want to operate on record keys. For example, you might specify **StringConverter**, **LongConverter**, **JsonConverter**, or **AvroConverter**.

11.3.3. Debezium CloudEvents converter configuration options

When you configure a Debezium connector to use the CloudEvent converter you can specify the following options.

Table 11.3. Descriptions of CloudEvents converter configuration options

Option	Default	Description
serializer.type	**json**	The encoding type to use for the CloudEvents envelope structure. The value can be **json** or **avro**.
data.serializer.type	**json**	The encoding type to use for the **data** attribute. The value can be **json** or **avro**.
json. ...	N/A	Any configuration options to be passed through to the underlying converter when using JSON. The **json.** prefix is removed.
avro. ...	N/A	Any configuration options to be passed through to the underlying converter when using Avro. The **avro.** prefix is removed. For example, for Avro **data**, you would specify the **avro.schema.registry.url** option.
schema.name.adjustment.mode	**avro**	Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. The value can be **avro** or **none**.

11.4. SENDING SIGNALS TO A DEBEZIUM CONNECTOR

The Debezium signaling mechanism provides a way to modify the behavior of a connector, or to trigger a one-time action, such as initiating an ad hoc snapshot of a table. To trigger a connector to perform a specified action, you issue a SQL command to add a signal message to a specialized signaling table, also referred to as a signaling data collection. The signaling table, which you create on the source database, is designated exclusively for communicating with Debezium. When Debezium detects that a new logging record or ad hoc snapshot record is added to the signaling table, it reads the signal, and initiates the requested operation.

Signaling is available for use with the following Debezium connectors:

- Db2
- MongoDB (Technology Preview)
- MySQL
- Oracle
- PostgreSQL
- SQL Server

11.4.1. Enabling Debezium signaling

By default, the Debezium signaling mechanism is disabled. You must explicitly enable signaling for each connector that you want to use it with.

Procedure

1. On the source database, create a signaling data collection table for sending signals to the connector. For information about the required structure of the signaling data collection, see Structure of a signaling data collection .
2. For source databases such as Db2 or SQL Server that implement a native change data capture (CDC) mechanism, enable CDC for the signaling table.
3. Add the name of the signaling data collection to the Debezium connector configuration.
   In the connector configuration, add the property **signal.data.collection**, and set its value to the fully-qualified name of the signaling data collection that you created in Step 1.

   For example, **signal.data.collection = inventory.debezium_signals**.

   The format for the fully-qualified name of the signaling collection depends on the connector. The following example shows the naming formats to use for each connector:

   Db2

   <schemaName>.<tableName>

   MongoDB

   <databaseName>.<collectionName>

   MySQL

   <databaseName>.<tableName>

   Oracle

   <databaseName>.<schemaName>.<tableName>

   PostgreSQL

   <schemaName>.<tableName>

   SQL Server

   <databaseName>.<schemaName>.<tableName>

   For more information about setting the **signal.data.collection** property, see the table of configuration properties for your connector.
4. Add the signaling table to the list of tables to monitor.

In the configuration for the Debezium connector, add the name of the data collection that you created in Step 1 to the **table.include.list** property.

For more information about the **table.include.list** property, see the table of configuration properties for your connector.

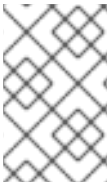
11.4.1.1. Required structure of a Debezium signaling data collection

A signaling data collection, or signaling table, stores signals that you send to a connector to trigger a specified operation. The structure of the signaling table must conform to the following standard format.

- Contains three fields (columns).
- Fields are arranged in a specific order, as shown in Table 1.

Table 11.4. Required structure of a signaling data collection

Field	Type	Description
id (required)	**string**	An arbitrary unique string that identifies a signal instance. You assign an **id** to each signal that you submit to the signaling table. Typically the ID is a UUID string. You can use signal instances for logging, debugging, or de-duplication. When a signal triggers Debezium to perform an incremental snapshot, it generates a signal message with an arbitrary **id** string. The **id** string that the generated message contains is unrelated to the **id** string in the submitted signal.
type (required)	**string**	Specifies the type of signal to send. You can use some signal types with any connector for which signaling is available, while other signal types are available for specific connectors only.
data (optional)	**string**	Specifies JSON-formatted parameters to pass to a signal action. Each signal type requires a specific set of data.



NOTE

The field names in a data collection are arbitrary. The preceding table provides suggested names. If you use a different naming convention, ensure that the values in each field are consistent with the expected content.

11.4.1.2. Creating a Debezium signaling data collection

You create a signaling table by submitting a standard SQL DDL query to the source database.

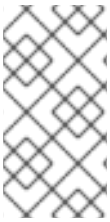
Prerequisites

- You have sufficient access privileges to create a table on the target database.

Procedure

- Submit a SQL query to the source database to create a table that is consistent with the required structure, as shown in the following example:

CREATE TABLE *<tableName>* (id VARCHAR(*<varcharValue>*) PRIMARY KEY, type VARCHAR(*<varcharValue>*) NOT NULL, data VARCHAR(*<varcharValue>*) NULL);



NOTE

The amount of space that you allocate to the **VARCHAR** parameter of the **id** variable must be sufficient to accommodate the size of the ID strings of signals sent to the signaling table.
If the size of an ID exceeds the available space, the connector cannot process the signal.

The following example shows a **CREATE TABLE** command that creates a three-column **debezium_signal** table:

```
CREATE TABLE debezium_signal (id VARCHAR(42) PRIMARY KEY, type VARCHAR(32) NOT NULL, data VARCHAR(2048) NULL);
```

11.4.2. Types of Debezium signal actions

You can use signaling to initiate the following actions:

- Add messages to the log.
- Trigger ad hoc snapshots.

Some signals are not compatible with all connectors.

11.4.2.1. Logging signals

You can request a connector to add an entry to the log by creating a signaling table entry with the **log** signal type. After processing the signal, the connector prints the specified message to the log. Optionally, you can configure the signal so that the resulting message includes the streaming coordinates.

Table 11.5. Example of a signaling record for adding a log message

Column	Value	Description
id	**924e3ff8-2245-43ca-ba77-2af9af02fa07**	
type	**log**	The action type of the signal.
data	**{"message": "Signal message at offset {}"}**	The **message** parameter specifies the string to print to the log. If you add a placeholder (**{}**) to the message, it is replaced with streaming coordinates.

11.4.2.2. Ad hoc snapshot signals

You can request a connector to initiate an ad hoc snapshot by creating a signaling table entry with the **execute-snapshot** signal type. After processing the signal, the connector runs the requested snapshot operation.

Unlike the initial snapshot that a connector runs after it first starts, an ad hoc snapshot occurs during runtime, after the connector has already begun to stream change events from a database. You can initiate ad hoc snapshots at any time.

Ad hoc snapshots are available for the following Debezium connectors:

- Db2
- MySQL
- Oracle
- PostgreSQL
- SQL Server

Table 11.6. Example of an ad hoc snapshot signal record

Column	Value
id	**d139b9b7-7777-4547-917d-e1775ea61d41**
type	**execute-snapshot**
data	**{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}**

Currently, the **execute-snapshot** action triggers incremental snapshots only.

For more information about ad hoc snapshots, see the *Snapshots* topic in the documentation for your connector.

Additional resources

- Db2 connector ad hoc snapshots
- MySQL connector ad hoc snapshots
- Oracle connector ad hoc snapshots
- PostgreSQL connector ad hoc snapshots
- SQL Server connector ad hoc snapshots

11.4.2.3. Incremental snapshots

Incremental snapshots are a specific type of ad hoc snapshot. In an incremental snapshot, the connector captures the baseline state of the tables that you specify, similar to an initial snapshot. However, unlike

an initial snapshot, an incremental snapshot captures tables in chunks, rather than all at once. The connector uses a watermarking method to track the progress of the snapshot.

By capturing the initial state of the specified tables in chunks rather than in a single monolithic operation, incremental snapshots provide the following advantages over the initial snapshot process:

- While the connector captures the baseline state of the specified tables, streaming of near real-time events from the transaction log continues uninterrupted.
- If the incremental snapshot process is interrupted, it can be resumed from the point at which it stopped.
- You can initiate an incremental snapshot at any time.

For more information about incremental snapshots, see the *Snapshots* topic in the documentation for your connector.

Additional resources

- Db2 connector incremental snapshots
- MySQL connector incremental snapshots
- Oracle connector incremental snapshots
- PostgreSQL connector incremental snapshots
- SQL Server connector incremental snapshots

CHAPTER 12. APPLYING TRANSFORMATIONS TO MODIFY MESSAGES EXCHANGED WITH APACHE KAFKA

Debezium provides several single message transformations (SMTs) that you can use to modify change event records. You can configure a connector to apply a transformation that modifies records before its sends them to Apache Kafka. You can also apply the Debezium SMTs to a sink connector to modify records before the connector reads from a Kafka topic.

If you want to apply transformations selectively to specific messages only , you can configure a Kafka Connect predicate to define the conditions for applying the SMT.

Debezium provides the following SMTs:

Topic router SMT

Reroutes change event records to specific topics based on a regular expression that is applied to the original topic name.

Content-based router SMT

Reroutes specified change event records based on the event content.

Message filtering SMT

Enables you to propagate a subset of event records to the destination Kafka topic. The transformation applies a regular expression to the change event records that a connector emits, based on the content of the event record. Only records that match the expression are written to the target topic. Other records are ignored.

New record state extraction SMT

Flattens the complex structure of a Debezium change event record into a simplified format. The simplified structure enables processing by sink connectors that cannot consume the original structure.

Outbox event router SMT

Provides support for the outbox pattern to enable safe and reliable data exchange among multiple services.

MongoDB outbox event router SMT

Provides support for using the outbox pattern with the MongoDB connector to enable safe and reliable data exchange among multiple services.

12.1. APPLYING TRANSFORMATIONS SELECTIVELY WITH SMT PREDICATES

When you configure a single message transformation (SMT) for a connector, you can define a predicate for the transformation. The predicate specifies how to apply the transformation conditionally to a subset of the messages that the connector processes. You can assign predicates to transformations that you configure for source connectors, such as Debezium, or to sink connectors.

12.1.1. About SMT predicates

Debezium provides several single message transformations (SMTs) that you can use to modify event records before Kafka Connect saves the records to Kafka topics. By default, when you configure one of these SMTs for a Debezium connector, Kafka Connect applies that transformation to every record that the connector emits. However, there might be instances in which you want to apply a transformation selectively, so that it modifies only that subset of change event messages that share a common characteristic.

For example, for a Debezium connector, you might want to run the transformation only on event messages from a specific table or that include a specific header key. In environments that run Apache Kafka 2.6 or greater, you can append a predicate statement to a transformation to instruct Kafka Connect to apply the SMT only to certain records. In the predicate, you specify a condition that Kafka Connect uses to evaluate each message that it processes. When a Debezium connector emits a change event message, Kafka Connect checks the message against the configured predicate condition. If the condition is true for the event message, Kafka Connect applies the transformation, and then writes the message to a Kafka topic. Messages that do not match the condition are sent to Kafka unmodified.

The situation is similar for predicates that you define for a sink connector SMT. The connector reads messages from a Kafka topic and Kafka Connect evaluates the messages against the predicate condition. If a message matches the condition, Kafka Connect applies the transformation and then passes the messages to the sink connector.

After you define a predicate, you can reuse it and apply it to multiple transforms. Predicates also include a **negate** option that you can use to invert a predicate so that the predicate condition is applied only to records that do *not* match the condition that is defined in the predicate statement. You can use the **negate** option to pair the predicate with other transforms that are based on negating the condition.

Predicate elements

Predicates include the following elements:

- **predicates** prefix
- Alias (for example, **isOutboxTable**)
- Type (for example, **org.apache.kafka.connect.transforms.predicates.TopicNameMatches**). Kafka Connect provides a set of default predicate types, which you can supplement by defining your own custom predicates.
- Condition statement and any additional configuration properties, depending on the type of predicate (for example, a regex naming pattern)

Default predicate types

The following predicate types are available by default:

HasHeaderKey

Specifies a key name in the header in the event message that you want Kafka Connect to evaluate. The predicate evaluates to true for any records that include a header key that has the specified name.

RecordIsTombstone

Matches Kafka *tombstone* records. The predicate evaluates to **true** for any record that has a **null** value. Use this predicate in combination with a filter SMT to remove tombstone records. This predicate has no configuration parameters.
A tombstone in Kafka is a record that has a key with a 0-byte, **null** payload. When a Debezium connector processes a delete operation in the source database, the connector emits two change events for the delete operation:

- A delete operation (**"op" : "d"**) event that provides the previous value of the database record.
- A tombstone event that has the same key, but a **null** value.
  The tombstone represents a delete marker for the row. When log compaction is enabled for Kafka, during compaction Kafka removes all events that share the same key as the

tombstone. Log compaction occurs periodically, with the compaction interval controlled by the **delete.retention.ms** setting for the topic.

Although it is possible to configure Debezium so that it does not emit tombstone events , it's best to permit Debezium to emit tombstones to maintain the expected behavior during log compaction. Suppressing tombstones prevents Kafka from removing records for a deleted key during log compaction. If your environment includes sink connectors that cannot process tombstones, you can configure the sink connector to use an SMT with the **RecordIsTombstone** predicate to filter out the tombstone records.

TopicNameMatches

A regular expression that specifies the name of a topic that you want Kafka Connect to match. The predicate is true for connector records in which the topic name matches the specified regular expression. Use this predicate to apply an SMT to records based on the name of the source table.

Additional resources

- KIP-585: Filter and Conditional SMTs
- Apache Kafka documentation for Kafka Connect predicates

12.1.2. Defining SMT predicates

By default, Kafka Connect applies each single message transformation in the Debezium connector configuration to every change event record that it receives from Debezium. Beginning with Apache Kafka 2.6, you can define an SMT predicate for a transformation in the connector configuration that controls how Kafka Connect applies the transformation. The predicate statement defines the conditions under which Kafka Connect applies the transformation to event records emitted by Debezium. Kafka Connect evaluates the predicate statement and then applies the SMT selectively to the subset of records that match the condition that is defined in the predicate. Configuring Kafka Connect predicates is similar to configuring transforms. You specify a predicate alias, associate the alias with a transform, and then define the type and configuration for the predicate.

Prerequisites

- The Debezium environment runs Apache Kafka 2.6 or greater.
- An SMT is configured for the Debezium connector.

Procedure

1. In the Debezium connector configuration, specify a predicate alias for the **predicates** parameter, for example, **IsOutboxTable**.
2. Associate the predicate alias with the transform that you want to apply conditionally, by appending the predicate alias to the transform alias in the connector configuration:

```
transforms.<TRANSFORM_ALIAS>.predicate=<PREDICATE_ALIAS>
```

For example:

```
transforms.outbox.predicate=IsOutboxTable
```

3. Configure the predicate by specifying its type and providing values for configuration parameters.

   a. For the type, specify one of the following default types that are available in Kafka Connect:

      - HasHeaderKey
      - RecordIsTombstone
      - TopicNameMatches
        For example:

        ```
        predicates.IsOutboxTable.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
        ```

   b. For the TopicNameMatch or **HasHeaderKey** predicates, specify a regular expression for the topic or header name that you want to match.
      For example:

      ```
      predicates.IsOutboxTable.pattern=outbox.event.*
      ```

4. If you want to negate a condition, append the **negate** keyword to the transform alias and set it to **true**.
   For example:

   ```
   transforms.outbox.negate=true
   ```

   The preceding property inverts the set of records that the predicate matches, so that Kafka Connect applies the transform to any record that does not match the condition specified in the predicate.

Example: TopicNameMatch predicate for the outbox event router transformation

The following example shows a Debezium connector configuration that applies the outbox event router transformation only to messages that Debezium emits to the Kafka **outbox.event.order** topic.

Because the **TopicNameMatch** predicate evaluates to *true* only for messages from the outbox table (**outbox.event.***), the transformation is not applied to messages that originate from other tables in the database.

```
transforms=outbox
transforms.outbox.predicate=IsOutboxTable
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
predicates=IsOutboxTable
predicates.IsOutboxTable.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
predicates.IsOutboxTable.pattern=outbox.event.*
```

12.1.3. Ignoring tombstone events

You can control whether Debezium emits tombstone events, and how long Kafka retains them. Depending on your data pipeline, you might want to set the **tombstones.on.delete** property for a connector so that Debezium does not emit tombstone events.

Whether you enable Debezium to emit tombstones depends on how topics are consumed in your environment and by the characteristics of the sink consumer. Some sink connectors rely on tombstone

events to remove records from downstream data stores. In cases where sink connectors rely on tombstone records to indicate when to delete records in downstream data stores, configure Debezium to emit them.

When you configure Debezium to generate tombstones, further configuration is required to ensure that sink connectors receive the tombstone events. The retention policy for a topic must be set so that the connector has time to read event messages before Kafka removes them during log compaction. The length of time that a topic retains tombstones before compaction is controlled by the **delete.retention.ms** property for the topic.

By default, the **tombstones.on.delete** property for a connector is set to **true** so that the connector generates a tombstone after each delete event. If you set the property to **false** to prevent Debezium from saving tombstone records to Kafka topics, the absence of tombstone records might lead to unintended consequences. Kafka relies on tombstone during log compaction to remove records that are related to a deleted key.

If you need to support sink connectors or downstream Kafka consumers that cannot process records with null values, rather than preventing Debezium from emitting tombstones, consider configuring an SMT for the connector with a predicate that uses the **RecordIsTombstone** predicate type to remove tombstone messages before consumers read them.

Procedure

- To prevent Debezium from emitting tombstone events for deleted database records, set the connector option **tombstones.on.delete** to **false**.
  For example:

  ```
  “tombstones.on.delete”: “false”
  ```

12.2. ROUTING DEBEZIUM EVENT RECORDS TO TOPICS THAT YOU SPECIFY

Each Kafka record that contains a data change event has a default destination topic. If you need to, you can re-route records to topics that you specify before the records reach the Kafka Connect converter. To do this, Debezium provides the topic routing single message transformation (SMT). Configure this transformation in the Debezium connector’s Kafka Connect configuration. Configuration options enable you to specify the following:

- An expression for identifying the records to re-route
- An expression that resolves to the destination topic
- How to ensure a unique key among the records being re-routed to the destination topic

It is up to you to ensure that the transformation configuration provides the behavior that you want. Debezium does not validate the behavior that results from your configuration of the transformation.

The topic routing transformation is a Kafka Connect SMT.

The following topics provide details:

- Section 12.2.1, “Use case for routing Debezium records to topics that you specify”
- Section 12.2.2, “Example of routing Debezium records for multiple tables to one topic”

- Section 12.2.3, "Ensuring unique keys across Debezium records routed to the same topic"
- Section 12.2.5, "Options for configuring Debezium topic routing transformation"

12.2.1. Use case for routing Debezium records to topics that you specify

The default behavior is that a Debezium connector sends each change event record to a topic whose name is formed from the name of the database and the name of the table in which the change was made. In other words, a topic receives records for one physical table. When you want a topic to receive records for more than one physical table, you must configure the Debezium connector to re-route the records to that topic.

Logical tables

A logical table is a common use case for routing records for multiple physical tables to one topic. In a logical table, there are multiple physical tables that all have the same schema. For example, sharded tables have the same schema. A logical table might consist of two or more sharded tables: **db_shard1.my_table** and **db_shard2.my_table**. The tables are in different shards and are physically distinct but together they form a logical table. You can re-route change event records for tables in any of the shards to the same topic.

Partitioned PostgreSQL tables

When the Debezium PostgreSQL connector captures changes in a partitioned table, the default behavior is that change event records are routed to a different topic for each partition. To emit records from all partitions to one topic, configure the topic routing SMT. Because each key in a partitioned table is guaranteed to be unique, configure **key.enforce.uniqueness=false** so that the SMT does not add a key field to ensure unique keys. The addition of a key field is default behavior.

12.2.2. Example of routing Debezium records for multiple tables to one topic

To route change event records for multiple physical tables to the same topic, configure the topic routing transformation in the Kafka Connect configuration for the Debezium connector. Configuration of the topic routing SMT requires you to specify regular expressions that determine:

- The tables for which to route records. These tables must all have the same schema.
- The destination topic name.

For example, configuration in a **.properties** file looks like this:

```
transforms=Reroute
transforms.Reroute.type=io.debezium.transforms.ByLogicalTableRouter
transforms.Reroute.topic.regex=(.*)customers_shard(.*)
transforms.Reroute.topic.replacement=$1customers_all_shards
```

topic.regex

Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
In the example, the regular expression, **(.*)customers_shard(.*)** matches records for changes to tables whose names include the **customers_shard** string. This would re-route records for tables with the following names:

myserver.mydb.customers_shard1
myserver.mydb.customers_shard2
myserver.mydb.customers_shard3

topic.replacement

Specifies a regular expression that represents the destination topic name. The transformation routes each matching record to the topic identified by this expression. In this example, records for the three sharded tables listed above would be routed to the **myserver.mydb.customers_all_shards** topic.

schema.name.adjustment.mode

Specifies how the message key schema names derived from the resulting topic name should be adjusted for compatibility with the message converter used by the connector. The value can be **avro** (default) or **none**.

Customizing the configuration

To customize the configuration you can define an SMT predicate statement that specifies the tables that you want the transformation to process, or not to process. A predicate might be useful if you configure the SMT to route tables that match a regular expression, and you do not want the SMT to reroute one particular table that matches the expression.

12.2.3. Ensuring unique keys across Debezium records routed to the same topic

A Debezium change event key uses the table columns that make up the table's primary key. To route records for multiple physical tables to one topic, the event key must be unique across all of those tables. However, it is possible for each physical table to have a primary key that is unique within only that table. For example, a row in the **myserver.mydb.customers_shard1** table might have the same key value as a row in the **myserver.mydb.customers_shard2** table.

To ensure that each event key is unique across the tables whose change event records go to the same topic, the topic routing transformation inserts a field into change event keys. By default, the name of the inserted field is **__dbz__physicalTableIdentifier**. The value of the inserted field is the default destination topic name.

If you want to, you can configure the topic routing transformation to insert a different field into the key. To do this, specify the **key.field.name** option and set it to a field name that does not clash with existing primary key field names. For example:

```
transforms=Reroute
transforms.Reroute.type=io.debezium.transforms.ByLogicalTableRouter
transforms.Reroute.topic.regex=(.*)customers_shard(.*)
transforms.Reroute.topic.replacement=$1customers_all_shards
transforms.Reroute.key.field.name=shard_id
```

This example adds the **shard_id** field to the key structure in routed records.

If you want to adjust the value of the key's new field, configure both of these options:

key.field.regex

Specifies a regular expression that the transformation applies to the default destination topic name to capture one or more groups of characters.

key.field.replacement

Specifies a regular expression for determining the value of the inserted key field in terms of those captured groups.

For example:

```
transforms.Reroute.key.field.regex=(.*)customers_shard(.*)
transforms.Reroute.key.field.replacement=$2
```

With this configuration, suppose that the default destination topic names are:

myserver.mydb.customers_shard1
myserver.mydb.customers_shard2
myserver.mydb.customers_shard3

The transformation uses the values in the second captured group, the shard numbers, as the value of the key's new field. In this example, the inserted key field's values would be **1**, **2**, or **3**.

If your tables contain globally unique keys and you do not need to change the key structure, you can set the **key.enforce.uniqueness** option to **false**:

```
...
transforms.Reroute.key.enforce.uniqueness=false
...
```

12.2.4. Options for applying the topic routing transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages.

You can use one of the following methods to configure the connector to apply the SMT selectively:

- Configure an SMT predicate for the transformation.
- Use the topic.regex configuration option for the SMT.

12.2.5. Options for configuring Debezium topic routing transformation

The following table describes topic routing SMT configuration options.

Table 12.1. Topic routing SMT configuration options

Option	Default	Description
topic.regex		Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
topic.replacement		Specifies a regular expression that represents the destination topic name. The transformation routes each matching record to the topic identified by this expression. This expression can refer to groups captured by the regular expression that you specify for **topic.regex**. To refer to a group, specify **$1**, **$2**, and so on.

Option	Default	Description
key.enforce.uniqueness	**true**	Indicates whether to add a field to the record's change event key. Adding a key field ensures that each event key is unique across the tables whose change event records go to the same topic. This helps to prevent collisions of change events for records that have the same key but that originate from different source tables. Specify **false** if you do not want the transformation to add a key field. For example, if you are routing records from a partitioned PostgreSQL table to one topic, you can configure **key.enforce.uniqueness=false** because unique keys are guaranteed in partitioned PostgreSQL tables.
key.field.name	**__dbz__physicalTableIdentifier**	Name of a field to be added to the change event key. The value of this field identifies the original table name. For the SMT to add this field, **key.enforce.uniqueness** must be **true**, which is the default.
key.field.regex		Specifies a regular expression that the transformation applies to the default destination topic name to capture one or more groups of characters. For the SMT to apply this expression, **key.enforce.uniqueness** must be **true**, which is the default.
key.field.replacement		Specifies a regular expression for determining the value of the inserted key field in terms of the groups captured by the expression specified for **key.field.regex**. For the SMT to apply this expression, **key.enforce.uniqueness** must be**true**, which is the default.

12.3. ROUTING CHANGE EVENT RECORDS TO TOPICS ACCORDING TO EVENT CONTENT

By default, Debezium streams all of the change events that it reads from a table to a single static topic. However, there might be situations in which you might want to reroute selected events to other topics, based on the event content. The process of routing messages based on their content is described in the Content-based routing messaging pattern. To apply this pattern in Debezium, you use the content-based routing single message transform (SMT) to write expressions that are evaluated for each event. Depending how an event is evaluated, the SMT either routes the event message to the original destination topic, or reroutes it to the topic that you specify in the expression.

IMPORTANT

The Debezium content-based routing SMT is a Technology Preview feature. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about support scope, see Technology Preview Features Support Scope.

While it is possible to use Java to create a custom SMT to encode routing logic, using a custom-coded SMT has its drawbacks. For example:

- It is necessary to compile the transformation up front and deploy it to Kafka Connect.
- Every change needs code recompilation and redeployment, leading to inflexible operations.

The content-based routing SMT supports scripting languages that integrate with JSR 223 (Scripting for the Java™ Platform).

Debezium does not come with any implementations of the JSR 223 API. To use an expression language with Debezium, you must download the JSR 223 script engine implementation for the language. Depending on the method that you use to deploy Debezium, you can automatically download the required artifacts from Maven Central, or you can manually download the artifacts, and then add them to your Debezium connector plug-in directories, along any other JAR files used by the language implementation.

12.3.1. Setting up the Debezium content-based-routing SMT

For security reasons, the content-based routing SMT is not included with the Debezium connector archives. Instead, it is provided in a separate artifact, **debezium-scripting-1.9.7.Final.tar.gz**.

If you deploy the Debezium connector by building a custom Kafka Connect container image from a Dockerfile, to use the filter SMT, you must explicitly add the SMT artifact to your Kafka Connect environment. When you use AMQ Streams to deploy the connector, it can download the required artifacts automatically based on configuration parameters that you specify in the Kafka Connect custom resource. IMPORTANT: After the routing SMT is present in a Kafka Connect instance, any user who is allowed to add a connector to the instance can run scripting expressions. To ensure that scripting expressions can be run only by authorized users, be sure to secure the Kafka Connect instance and its configuration interface before you add the routing SMT.

The following procedure applies if you build your Kafka Connect container image from a Dockerfile. If you use AMQ Streams to create the Kafka Connect image, follow the instructions in the deployment topic for your connector.

Procedure

1. From a browser, open the Red Hat build of Debezium download site , and download the Debezium scripting SMT archive (**debezium-scripting-1.9.7.Final.tar.gz**).
2. Extract the contents of the archive into the Debezium plug-in directories of your Kafka Connect environment.
3. Obtain a JSR-223 script engine implementation and add its contents to the Debezium plug-in directories of your Kafka Connect environment.

4. Restart the Kafka Connect process to pick up the new JAR files.

The Groovy language needs the following libraries on the classpath:

- **groovy**
- **groovy-json** (optional)
- **groovy-jsr223**

The JavaScript language needs the following libraries on the classpath:

- **graalvm.js**
- **graalvm.js.scriptengine**

12.3.2. Example: Debezium basic content-based routing configuration

To configure a Debezium connector to route change event records based on the event content, you configure the **ContentBasedRouter** SMT in the Kafka Connect configuration for the connector.

Configuration of the content-based routing SMT requires you to specify a regular expression that defines the filtering criteria. In the configuration, you create a regular expression that defines routing criteria. The expression defines a pattern for evaluating event records. It also specifies the name of a destination topic where events that match the pattern are routed. The pattern that you specify might designate an event type, such as a table insert, update, or delete operation. You might also define a pattern that matches a value in a specific column or row.

For example, to reroute all update (**u**) records to an **updates** topic, you might add the following configuration to your connector configuration:

```
...
transforms=route
transforms.route.type=io.debezium.transforms.ContentBasedRouter
transforms.route.language=jsr223.groovy
transforms.route.topic.expression=value.op == 'u' ? 'updates' : null
...
```

The preceding example specifies the use of the **Groovy** expression language.

Records that do not match the pattern are routed to the default topic.

Customizing the configuration

The preceding example shows a simple SMT configuration that is designed to process only DML events, which contain an **op** field. Other types of messages that a connector might emit (heartbeat messages, tombstone messages, or metadata messages about transactions or schema changes) do not contain this field. To avoid processing failures, you can define an SMT predicate statement that selectively applies the transformation to specific events only.

12.3.3. Variables for use in Debezium content-based routing expressions

Debezium binds certain variables into the evaluation context for the SMT. When you create expressions to specify conditions to control the routing destination, the SMT can look up and interpret the values of these variables to evaluate conditions in an expression.

The following table lists the variables that Debezium binds into the evaluation context for the content-based routing SMT:

Table 12.2. Content-based routing expression variables

Name	Description	Type
key	A key of the message.	**org.apache.kafka.connect.data.Struct**
value	A value of the message.	**org.apache.kafka.connect.data.Struct**
keySchema	Schema of the message key.	**org.apache.kafka.connect.data.Schema**
valueSchema	Schema of the message value.	**org.apache.kafka.connect.data.Schema**
topic	Name of the target topic.	String
headers	A Java map of message headers. The key field is the header name. The **headers** variable exposes the following properties: • **value** (of type **Object**) • **schema** (of type **org.apache.kafka.connect.data.Schema**)	**java.util.Map<String, io.debezium.transforms.scripting.RecordHeader>**

An expression can invoke arbitrary methods on its variables. Expressions should resolve to a Boolean value that determines how the SMT dispositions the message. When the routing condition in an expression evaluates to **true**, the message is retained. When the routing condition evaluates to **false**, the message is removed.

Expressions should not result in any side-effects. That is, they should not modify any variables that they pass.

12.3.4. Options for applying the content-based routing transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages. You can use one of the following methods to configure the connector to apply the SMT selectively:

- Configure an SMT predicate for the transformation.
- Use the topic.regex configuration option for the SMT.

12.3.5. Configuration of content-based routing conditions for other scripting languages

The way that you express content-based routing conditions depends on the scripting language that you use. For example, as shown in the basic configuration example, when you use **Groovy** as the expression language, the following expression reroutes all update (**u**) records to the **updates** topic, while routing other records to the default topic:

```
value.op == 'u' ? 'updates' : null
```

Other languages use different methods to express the same condition.

TIP

The Debezium MongoDB connector emits the **after** and **patch** fields as serialized JSON documents rather than as structures.
To use the ContentBasedRouting SMT with the MongoDB connector, you must first unwind the array fields in the JSON into separate documents.
You can use a JSON parser within an expression to generate separate output documents for each array item. For example, if you use Groovy as the expression language, add the **groovy-json** artifact to the classpath, and then add an expression such as **(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'**.

Javascript

When you use JavaScript as the expression language, you can call the **Struct#get()** method to specify the content-based routing condition, as in the following example:

```
value.get('op') == 'u' ? 'updates' : null
```

Javascript with Graal.js

When you create content-based routing conditions by using JavaScript with Graal.js, you use an approach that is similar to the one use with Groovy. For example:

```
value.op == 'u' ? 'updates' : null
```

12.3.6. Options for configuring the content-based routing transformation

Property	Default	Description
topic.regex		An optional regular expression that evaluates the name of the destination topic for an event to determine whether to apply the condition logic. If the name of the destination topic matches the value in **topic.regex**, the transformation applies the condition logic before it passes the event to the topic. If the name of the topic does not match the value in **topic.regex**, the SMT passes the event to the topic unmodified.

language		The language in which the expression is written. Must begin with **jsr223.**, for example, **jsr223.groovy**, or **jsr223.graal.js**. Debezium supports bootstrapping through the JSR 223 API ("Scripting for the Java ™ Platform") only.
topic.expression		The expression to be evaluated for every message. Must evaluate to a **String** value where a result of non-null reroutes the message to a new topic, and a **null** value routes the message to the default topic.
null.handling.mode	**keep**	Specifies how the transformation handles **null** (tombstone) messages. You can specify one of the following options: **keep** (Default) Pass the messages through. **drop** Remove the messages completely. **evaluate** Apply the condition logic to the messages.

12.4. FILTERING DEBEZIUM CHANGE EVENT RECORDS

By default, Debezium delivers every data change event that it receives to the Kafka broker. However, in many cases, you might be interested in only a subset of the events emitted by the producer. To enable you to process only the records that are relevant to you, Debezium provides the *filter* single message transform (SMT).

IMPORTANT

The Debezium filter SMT is a Technology Preview feature. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete; therefore, Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about support scope, see Technology Preview Features Support Scope .

While it is possible to use Java to create a custom SMT to encode filtering logic, using a custom-coded SMT has its drawbacks. For example:

- It is necessary to compile the transformation up front and deploy it to Kafka Connect.
- Every change needs code recompilation and redeployment, leading to inflexible operations.

The filter SMT supports scripting languages that integrate with JSR 223 (Scripting for the Java™ Platform).

Debezium does not come with any implementations of the JSR 223 API. To use an expression language with Debezium, you must download the JSR 223 script engine implementation for the language. Depending on the method that you use to deploy Debezium, you can automatically download the required artifacts from Maven Central, or you can manually download the artifacts, and then add them to your Debezium connector plug-in directories, along any other JAR files used by the language implementation.

12.4.1. Setting up the Debezium filter SMT

For security reasons, the filter SMT is not included with the Debezium connector archives. Instead, it is provided in a separate artifact, **debezium-scripting-1.9.7.Final.tar.gz**.

If you deploy the Debezium connector by building a custom Kafka Connect container image from a Dockerfile, to use the filter SMT, you must explicitly download the SMT archive and deploy the files alongside the connector plug-in. When you use AMQ Streams to deploy the connector, it can download the required artifacts automatically based on configuration parameters that you specify in the Kafka Connect custom resource. IMPORTANT: After the filter SMT is present in a Kafka Connect instance, any user who is allowed to add a connector to the instance can run scripting expressions. To ensure that scripting expressions can be run only by authorized users, be sure to secure the Kafka Connect instance and its configuration interface before you add the filter SMT.

The following procedure applies if you build your Kafka Connect container image from a Dockerfile. If you use AMQ Streams to create the Kafka Connect image, follow the instructions in the deployment topic for your connector.

Procedure

1. From a browser, open the Red Hat build of Debezium download site , and download the Debezium scripting SMT archive (**debezium-scripting-1.9.7.Final.tar.gz**).
2. Extract the contents of the archive into the Debezium plug-in directories of your Kafka Connect environment.
3. Obtain a JSR-223 script engine implementation and add its contents to the Debezium plug-in directories of your Kafka Connect environment.
4. Restart the Kafka Connect process to pick up the new JAR files.

The Groovy language needs the following libraries on the classpath:

- **groovy**
- **groovy-json** (optional)
- **groovy-jsr223**

The JavaScript language needs the following libraries on the classpath:

- **graalvm.js**
- **graalvm.js.scriptengine**

12.4.2. Example: Debezium basic filter SMT configuration

You configure the filter transformation in the Debezium connector's Kafka Connect configuration. In the configuration, you specify the events that you are interested in by defining filter conditions that are

based on business rules. As the filter SMT processes the event stream, it evaluates each event against the configured filter conditions. Only events that meet the criteria of the filter conditions are passed to the broker.

To configure a Debezium connector to filter change event records, configure the **Filter** SMT in the Kafka Connect configuration for the Debezium connector. Configuration of the filter SMT requires you to specify a regular expression that defines the filtering criteria.

For example, you might add the following configuration in your connector configuration.

```
...
transforms=filter
transforms.filter.type=io.debezium.transforms.Filter
transforms.filter.language=jsr223.groovy
transforms.filter.condition=value.op == 'u' && value.before.id == 2
...
```

The preceding example specifies the use of the **Groovy** expression language. The regular expression **value.op == 'u' && value.before.id == 2** removes all messages, except those that represent update (**u**) records with **id** values that are equal to **2**.

Customizing the configuration

The preceding example shows a simple SMT configuration that is designed to process only DML events, which contain an **op** field. Other types of messages that a connector might emit (heartbeat messages, tombstone messages, or metadata messages about schema changes and transactions) do not contain this field. To avoid processing failures, you can define an SMT predicate statement that selectively applies the transformation to specific events only.

12.4.3. Variables for use in filter expressions

Debezium binds certain variables into the evaluation context for the filter SMT. When you create expressions to specify filter conditions, you can use the variables that Debezium binds into the evaluation context. By binding variables, Debezium enables the SMT to look up and interpret their values as it evaluates the conditions in an expression.

The following table lists the variables that Debezium binds into the evaluation context for the filter SMT:

Table 12.3. Filter expression variables

Name	Description	Type
key	A key of the message.	**org.apache.kafka.connect.data.Struct**
value	A value of the message.	**org.apache.kafka.connect.data.Struct**
keySchema	Schema of the message key.	**org.apache.kafka.connect.data.Schema**
valueSchema	Schema of the message value.	**org.apache.kafka.connect.data.Schema**

Name	Description	Type
topic	Name of the target topic.	String
headers	A Java map of message headers. The key field is the header name. The **headers** variable exposes the following properties: • **value** (of type **Object**) • **schema** (of type **org.apache.kafka.connect.data.Schema**)	**java.util.Map<String, io.debezium.transforms.scripting.RecordHeader>**

An expression can invoke arbitrary methods on its variables. Expressions should resolve to a Boolean value that determines how the SMT dispositions the message. When the filter condition in an expression evaluates to **true**, the message is retained. When the filter condition evaluates to **false**, the message is removed.

Expressions should not result in any side-effects. That is, they should not modify any variables that they pass.

12.4.4. Options for applying the filter transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages. You can use one of the following methods to configure the connector to apply the SMT selectively:

- Configure an SMT predicate for the transformation.
- Use the topic.regex configuration option for the SMT.

12.4.5. Filter condition configuration for other scripting languages

The way that you express filtering conditions depends on the scripting language that you use.

For example, as shown in the basic configuration example, when you use **Groovy** as the expression language, the following expression removes all messages, except for update records that have **id** values set to **2**:

```
value.op == 'u' && value.before.id == 2
```

Other languages use different methods to express the same condition.

TIP

The Debezium MongoDB connector emits the **after** and **patch** fields as serialized JSON documents rather than as structures.
To use the filter SMT with the MongoDB connector, you must first unwind the array fields in the JSON into separate documents.
You can use a JSON parser within an expression to generate separate output documents for each array item. For example, if you use Groovy as the expression language, add the **groovy-json** artifact to the classpath, and then add an expression such as **(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'**.

Javascript

If you use JavaScript as the expression language, you can call the **Struct#get()** method to specify the filtering condition, as in the following example:

```
value.get('op') == 'u' && value.get('before').get('id') == 2
```

Javascript with Graal.js

If you use JavaScript with Graal.js to define filtering conditions, you use an approach that is similar to the one that you use with Groovy. For example:

```
value.op == 'u' && value.before.id == 2
```

12.4.6. Options for configuring filter transformation

The following table lists the configuration options that you can use with the filter SMT.

Table 12.4. filter SMT configuration options

Property	Default	Description
topic.regex		An optional regular expression that evaluates the name of the destination topic for an event to determine whether to apply filtering logic. If the name of the destination topic matches the value in **topic.regex**, the transformation applies the filter logic before it passes the event to the topic. If the name of the topic does not match the value in **topic.regex**, the SMT passes the event to the topic unmodified.
language		The language in which the expression is written. Must begin with **jsr223.**, for example, **jsr223.groovy**, or **jsr223.graal.js**. Debezium supports bootstrapping through the JSR 223 API ("Scripting for the Java ™ Platform") only.
condition		The expression to be evaluated for every message. Must evaluate to a Boolean value where a result of **true** keeps the message, and a result of **false** removes it.

null.handling.mode	keep	Specifies how the transformation handles **null** (tombstone) messages. You can specify one of the following options: **keep** (Default) Pass the messages through. **drop** Remove the messages completely. **evaluate** Apply the filter condition to the messages.

12.5. EXTRACTING SOURCE RECORD AFTER STATE FROM DEBEZIUM CHANGE EVENTS

A Debezium data change event has a complex structure that provides a wealth of information. Kafka records that convey Debezium change events contain all of this information. However, parts of a Kafka ecosystem might expect Kafka records that provide a flat structure of field names and values. To provide this kind of record, Debezium provides the event flattening single message transformation (SMT). Configure this transformation when consumers need Kafka records that have a format that is simpler than Kafka records that contain Debezium change events.

The event flattening transformation is a Kafka Connect SMT.

This transformation is available to only SQL database connectors.

The following topics provide details:

- Section 12.5.1, "Description of Debezium change event structure"
- Section 12.5.2, "Behavior of Debezium event flattening transformation"
- Section 12.5.3, "Configuration of Debezium event flattening transformation"
- Section 12.5.4, "Example of adding Debezium metadata to the Kafka record"
- Section 12.5.6, "Options for configuring Debezium event flattening transformation"

12.5.1. Description of Debezium change event structure

Debezium generates data change events that have a complex structure. Each event consists of three parts:

- Metadata, which includes but is not limited to:
  - The operation that made the change
  - Source information such as the names of the database and table where the change was made
  - Time stamp for when the change was made
  - Optional transaction information

- Row data before the change
- Row data after the change

For example, part of the structure of an **UPDATE** change event looks like this:

```
{
 "op": "u",
 "source": {
  ...
 },
 "ts_ms" : "...",
 "before" : {
  "field1" : "oldvalue1",
  "field2" : "oldvalue2"
 },
 "after" : {
  "field1" : "newvalue1",
  "field2" : "newvalue2"
 }
}
```

This complex format provides the most information about changes happening in the system. However, other connectors or other parts of the Kafka ecosystem usually expect the data in a simple format like this:

```
{
 "field1" : "newvalue1",
 "field2" : "newvalue2"
}
```

To provide the needed Kafka record format for consumers, configure the event flattening SMT.

12.5.2. Behavior of Debezium event flattening transformation

The event flattening SMT extracts the **after** field from a Debezium change event in a Kafka record. The SMT replaces the original change event with only its **after** field to create a simple Kafka record.

You can configure the event flattening SMT for a Debezium connector or for a sink connector that consumes messages emitted by a Debezium connector. The advantage of configuring event flattening for a sink connector is that records stored in Apache Kafka contain whole Debezium change events. The decision to apply the SMT to a source or sink connector depends on your particular use case.

You can configure the transformation to do any of the following:

- Add metadata from the change event to the simplified Kafka record. The default behavior is that the SMT does not add metadata.
- Keep Kafka records that contain change events for **DELETE** operations in the stream. The default behavior is that the SMT drops Kafka records for **DELETE** operation change events because most consumers cannot yet handle them.

A database **DELETE** operation causes Debezium to generate two Kafka records:

- A record that contains **"op": "d",** the **before** row data, and some other fields.

- A tombstone record that has the same key as the deleted row and a value of **null**. This record is a marker for Apache Kafka. It indicates that log compaction can remove all records that have this key.

Instead of dropping the record that contains the **before** row data, you can configure the event flattening SMT to do one of the following:

- Keep the record in the stream and edit it to have only the **"value": "null"** field.
- Keep the record in the stream and edit it to have a **value** field that contains the key/value pairs that were in the **before** field with an added **"__deleted": "true"** entry.

Similary, instead of dropping the tombstone record, you can configure the event flattening SMT to keep the tombstone record in the stream.

12.5.3. Configuration of Debezium event flattening transformation

Configure the Debezium event flattening SMT in a Kafka Connect source or sink connector by adding the SMT configuration details to your connector's configuration. To obtain the default behavior, in a **.properties** file, you would specify something like the following:

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
```

As with any Kafka Connect connector configuration, you can set **transforms=** to multiple, comma-separated, SMT aliases in the order in which you want Kafka Connect to apply the SMTs.

The following **.properties** example sets several event flattening SMT options:

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
transforms.unwrap.drop.tombstones=false
transforms.unwrap.delete.handling.mode=rewrite
transforms.unwrap.add.fields=table,lsn
```

drop.tombstones=false

Keeps tombstone records for **DELETE** operations in the event stream.

delete.handling.mode=rewrite

For **DELETE** operations, edits the Kafka record by flattening the **value** field that was in the change event. The **value** field directly contains the key/value pairs that were in the **before** field. The SMT adds **__deleted** and sets it to **true**, for example:

```
"value": {
  "pk": 2,
  "cola": null,
  "__deleted": "true"
}
```

add.fields=table,lsn

Adds change event metadata for the **table** and **lsn** fields to the simplified Kafka record.

Customizing the configuration

The connector might emit many types of event messages (heartbeat messages, tombstone messages,

or metadata messages about transactions or schema changes). To apply the transformation to a subset of events, you can define an SMT predicate statement that selectively applies the transformation to specific events only.

12.5.4. Example of adding Debezium metadata to the Kafka record

The event flattening SMT can add original, change event metadata to the simplified Kafka record. For example, you might want the simplified record's header or value to contain any of the following:

- The type of operation that made the change
- The name of the database or table that was changed
- Connector-specific fields such as the Postgres LSN field

To add metadata to the simplified Kafka record's header, specify the **add.headers** option. To add metadata to the simplified Kafka record's value, specify the **add.fields** option. Each of these options takes a comma separated list of change event field names. Do not specify spaces. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field. For example:

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
transforms.unwrap.add.fields=op,table,lsn,source.ts_ms
transforms.unwrap.add.headers=db
transforms.unwrap.delete.handling.mode=rewrite
```

With that configuration, a simplified Kafka record would contain something like the following:

```
{
 ...
 "__op" : "c",
 "__table": "MY_TABLE",
 "__lsn": "123456789",
 "__source_ts_ms" : "123456789",
 ...
}
```

Also, simplified Kafka records would have a **__db** header.

In the simplified Kafka record, the SMT prefixes the metadata field names with a double underscore. When you specify a struct, the SMT also inserts an underscore between the struct name and the field name.

To add metadata to a simplified Kafka record that is for a **DELETE** operation, you must also configure **delete.handling.mode=rewrite**.

12.5.5. Options for applying the event flattening transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages.

For more information about how to apply the SMT selectively, see Configure an SMT predicate for the transformation.

12.5.6. Options for configuring Debezium event flattening transformation

The following table describes the options that you can specify to configure the event flattening SMT.

Table 12.5. Descriptions of event flattening SMT configuration options

Option	Default	Description
drop.tombstones	**true**	Debezium generates a tombstone record for each **DELETE** operation. The default behavior is that event flattening SMT removes tombstone records from the stream. To keep tombstone records in the stream, specify **drop.tombstones=false**.
delete.handling.mode	**drop**	Debezium generates a change event record for each **DELETE** operation. The default behavior is that event flattening SMT removes these records from the stream. To keep Kafka records for **DELETE** operations in the stream, set **delete.handling.mode** to **none** or **rewrite**. Specify **none** to keep the change event record in the stream. The record contains only **"value": "null"**. Specify **rewrite** to keep the change event record in the stream and edit the record to have a **value** field that contains the key/value pairs that were in the **before** field and also add **__deleted: true** to the **value**. This is another way to indicate that the record has been deleted. When you specify **rewrite**, the updated simplified records for **DELETE** operations might be all you need to track deleted records. You can consider accepting the default behavior of dropping the tombstone records that the Debezium connector creates.

Option	Default	Description
route.by.field		To use row data to determine the topic to route the record to, set this option to an **after** field attribute. The SMT routes the record to the topic whose name matches the value of the specified **after** field attribute. For a**DELETE** operation, set this option to a **before** field attribute. For example, configuration of **route.by.field=destination** routes records to the topic whose name is the value of **after.destination**. The default behavior is that a Debezium connector sends each change event record to a topic whose name is formed from the name of the database and the name of the table in which the change was made. If you are configuring the event flattening SMT on a sink connector, setting this option might be useful when the destination topic name dictates the name of the database table that will be updated with the simplified change event record. If the topic name is not correct for your use case, you can configure **route.by.field** to re-route the event.
add.fields.prefix	__ (double-underscore)	Set this optional string to prefix a field.

Option	Default	Description
add.fields		Set this option to a comma-separated list, with no spaces, of metadata fields to add to the simplified Kafka record's value. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field, for example **source.ts_ms**. Optionally, you can override the field name via **<field name>:<new field name>**, e.g. like so: new field name like **version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP**. Please note that the **new field name** is case-sensitive. When the SMT adds metadata fields to the simplified record's value, it prefixes each metadata field name with a double underscore. For a struct specification, the SMT also inserts an underscore between the struct name and the field name. If you specify a field that is not in the change event record, the SMT still adds the field to the record's value.
add.headers.prefix	__ (double-underscore)	Set this optional string to prefix a header.

Option	Default	Description
add.headers		Set this option to a comma-separated list, with no spaces, of metadata fields to add to the header of the simplified Kafka record. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field, for example **source.ts_ms**. Optionally, you can override the field name via **<field name>:<new field name>**, e.g. like so: new field name like **version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP**. Please note that the **new field name** is case-sensitive. When the SMT adds metadata fields to the simplified record's header, it prefixes each metadata field name with a double underscore. For a struct specification, the SMT also inserts an underscore between the struct name and the field name. If you specify a field that is not in the change event record, the SMT does not add the field to the header.

12.6. CONFIGURING DEBEZIUM CONNECTORS TO USE THE OUTBOX PATTERN

The outbox pattern is a way to safely and reliably exchange data between multiple (micro) services. An outbox pattern implementation avoids inconsistencies between a service's internal state (as typically persisted in its database) and state in events consumed by services that need the same data.

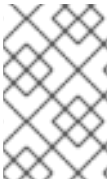
To implement the outbox pattern in a Debezium application, configure a Debezium connector to:

- Capture changes in an outbox table
- Apply the Debezium outbox event router single message transformation (SMT)

A Debezium connector that is configured to apply the outbox SMT should capture changes that occur in an outbox table only. For more information, see Options for applying the transformation selectively.

A connector can capture changes in more than one outbox table only if each outbox table has the same structure.

See Reliable Microservices Data Exchange With the Outbox Pattern to learn about why the outbox pattern is useful and how it works.



NOTE

The outbox event router SMT is not compatible with the MongoDB connector.

MongoDB users can run the MongoDB outbox event router SMT.

The following topics provide details:

- Section 12.6.1, "Example of a Debezium outbox message"
- Section 12.6.2, "Outbox table structure expected by Debezium outbox event router SMT"
- Section 12.6.3, "Basic Debezium outbox event router SMT configuration"
- Section 12.6.4, "Options for applying the Outbox event router transformation selectively"
- Section 12.6.5, "Using Avro as the payload format in Debezium outbox messages"
- Section 12.6.6, "Emitting additional fields in Debezium outbox messages"
- Section 12.6.7, "Expanding escaped JSON String as JSON"
- Section 12.6.8, "Options for configuring outbox event router transformation"

12.6.1. Example of a Debezium outbox message

To understand how the Debezium outbox event router SMT is configured, review the following example of a Debezium outbox message:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\",
\"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\":
\"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
\"customerId\": 123}"
}
```

A Debezium connector that is configured to apply the outbox event router SMT generates the above message by transforming a Debezium raw message like this:

```
# Kafka Message key: "406c07f3-26f0-4eea-a50c-109940064b8f"
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484
{
  "before": null,
  "after": {
    "id": "406c07f3-26f0-4eea-a50c-109940064b8f",
    "aggregateid": "1",
    "aggregatetype": "Order",
    "payload": "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\":
\"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\",
\"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
```

```
\"customerId\": 123}",
    "timestamp": 1556890294344,
    "type": "OrderCreated"
  },
  "source": {
    "version": "1.9.7.Final",
    "connector": "postgresql",
    "name": "dbserver1-bare",
    "db": "orderdb",
    "ts_usec": 1556890294448870,
    "txId": 584,
    "lsn": 24064704,
    "schema": "inventory",
    "table": "outboxevent",
    "snapshot": false,
    "last_snapshot_record": null,
    "xmin": null
  },
  "op": "c",
  "ts_ms": 1556890294484
}
```


This example of a Debezium outbox message is based on the default outbox event router configuration , which assumes an outbox table structure and event routing based on aggregates. To customize behavior, the outbox event router SMT provides numerous configuration options.

12.6.2. Outbox table structure expected by Debezium outbox event router SMT

To apply the default outbox event router SMT configuration, your outbox table is assumed to have the following columns:

```
Column       |          Type          | Modifiers
-------------+------------------------+-----------
id           | uuid                   | not null
aggregatetype | character varying(255) | not null
aggregateid   | character varying(255) | not null
type          | character varying(255) | not null
payload       | jsonb                  |
```

Table 12.6. Descriptions of expected outbox table columns

Column	Effect
id	Contains the unique ID of the event. In an outbox message, this value is a header. You can use this ID, for example, to remove duplicate messages. To obtain the unique ID of the event from a different outbox table column, set the **table.field.event.id** SMT option in the connector configuration.

Column	Effect
aggregatetype	Contains a value that the SMT appends to the name of the topic to which the connector emits an outbox message. The default behavior is that this value replaces the default **${routedByValue}** variable in the **route.topic.replacement** SMT option. For example, in a default configuration, the **route.by.field** SMT option is set to **aggregatetype** and the **route.topic.replacement** SMT option is set to **outbox.event.${routedByValue}**. Suppose that your application adds two records to the outbox table. In the first record, the value in the **aggregatetype** column is **customers**. In the second record, the value in the **aggregatetype** column is **orders**. The connector emits the first record to the **outbox.event.customers** topic. The connector emits the second record to the **outbox.event.orders** topic. To obtain this value from a different outbox table column, set the **route.by.field** SMT option in the connector configuration.
aggregateid	Contains the event key, which provides an ID for the payload. The SMT uses this value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions. To obtain the event key from a different outbox table column, set the **table.field.event.key** SMT option in the connector configuration.
payload	A representation of the outbox change event. The default structure is JSON. By default, the Kafka message value is solely comprised of the **payload** value. However, if the outbox event is configured to include additional fields, the Kafka message value contains an envelope encapsulating both payload and the additional fields, and each field is represented separately. For more information, see Emitting messages with additional fields. To obtain the event payload from a different outbox table column, set the **table.field.event.payload** SMT option in the connector configuration.
Additional custom columns	Any additional columns from the outbox table can be added to outbox events either within the payload section or as a message header. One example could be a column **eventType** which conveys a user-defined value that helps to categorize or organize events.

12.6.3. Basic Debezium outbox event router SMT configuration

To configure a Debezium connector to support the outbox pattern, configure the **outbox.EventRouter** SMT. For example, the basic configuration in a **.properties** file looks like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
```

Customizing the configuration

The connector might emit many types of event messages (for example, heartbeat messages, tombstone messages, or metadata messages about transactions or schema changes). To apply the transformation only to events that originate in the outbox table, define an SMT predicate statement that selectively applies the transformation to those events only.

12.6.4. Options for applying the Outbox event router transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages. You can use one of the following methods to configure the connector to apply the SMT selectively:

- Configure an SMT predicate for the transformation.
- Use the **route.topic.regex** configuration option for the SMT.

12.6.5. Using Avro as the payload format in Debezium outbox messages

The outbox event router SMT supports arbitrary payload formats. The **payload** column value in an outbox table is passed on transparently. An alternative to working with JSON is to use Avro. This can be beneficial for message format governance and for ensuring that outbox event schemas evolve in a backwards-compatible way.

How a source application produces Avro formatted content for outbox message payloads is out of the scope of this documentation. One possibility is to leverage the **KafkaAvroSerializer** class to serialize **GenericRecord** instances. To ensure that the Kafka message value is the exact Avro binary data, apply the following configuration to the connector:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.ByteBufferConverter
```

By default, the **payload** column value (the Avro data) is the only message value. Configuration of **ByteBufferConverter** as the value converter propagates the **payload** column value as-is into the Kafka message value.

The Debezium connectors may be configured to emit heartbeat, transaction metadata, or schema change events (support varies by connector). These events cannot be serialized by the **ByteBufferConverter** so additional configuration must be provided so the converter knows how to serialize these events. As an example, the following configuration illustrates using the Apache Kafka **JsonConverter** with no schemas:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.ByteBufferConverter
value.converter.delegate.converter.type=org.apache.kafka.connect.json.JsonConverter
value.converter.delegate.converter.type.schemas.enable=false
```

The delegate **Converter** implementation is specified by the **delegate.converter.type** option. If any extra configuration options are needed by the converter, they can also be specified, such as the disablement of schemas shown above using **schemas.enable=false**.

12.6.6. Emitting additional fields in Debezium outbox messages

Your outbox table might contain columns whose values you want to add to the emitted outbox messages. For example, consider an outbox table that has a value of **purchase-order** in the **aggregatetype** column and another column, **eventType**, whose possible values are **order-created** and **order-shipped**. Additional fields can be added with the syntax **column:placement:alias**.

The allowed values for **placement** are: - **header** - **envelope** - **partition**

To emit the **eventType** column value in the outbox message header, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:header:type
```

The result will be a header on the Kafka message with **type** as its key, and the value of the **eventType** column as its value.

To emit the **eventType** column value in the outbox message envelope, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:envelope:type
```

To control which partition the outbox message is produced on, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=partitionColumn:partition
```

Note that for the **partition** placement, adding an alias will have no effect.

12.6.7. Expanding escaped JSON String as JSON

You may have noticed that the Debezium outbox message contains the **payload** represented as a String. So when this string, is actually JSON, it appears as escaped in the result Kafka message like shown below:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\",
\"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\":
\"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
\"customerId\": 123}"
}
```

The outbox event router allows you to expand this message content to "real" JSON with the companion schema being deduced from the JSON document itself. That way the result in Kafka message looks like:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
```

```
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "id": 1, "lineItems": [{"id": 1, "item": "Debezium in Action", "status": "ENTERED", "quantity": 2,
"totalPrice": 39.98}, {"id": 2, "item": "Debezium for Dummies", "status": "ENTERED", "quantity": 1,
"totalPrice": 29.99}], "orderDate": "2019-01-31T12:13:01", "customerId": 123
}
```

To enable this transformation, you have to set the **table.expand.json.payload** to true and use the **JsonConverter** like below:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.expand.json.payload=true
value.converter=org.apache.kafka.connect.json.JsonConverter
```

12.6.8. Options for configuring outbox event router transformation

The following table describes the options that you can specify for the outbox event router SMT. In the table, the **Group** column indicates a configuration option classification for Kafka.

Table 12.7. Descriptions of outbox event router SMT configuration options

Option	Default	Group	Description
table.op.invalid.behavior	**warn**	Table	Determines the behavior of the SMT when there is an **UPDATE** operation on the outbox table. Possible settings are: • **warn** - The SMT logs a warning and continues to the next outbox table record. • **error** - The SMT logs an error and continues to the next outbox table record. • **fatal** - The SMT logs an error and the connector stops processing. All changes in an outbox table are expected to be **INSERT** operations. That is, an outbox table functions as a queue; updates to records in an outbox table are not allowed. The SMT automatically filters out **DELETE** operations on an outbox table.
table.field.event.id	**id**	Table	Specifies the outbox table column that contains the unique event ID. This ID will be stored in the emitted event's headers under the **id** key.

Option	Default	Group	Description
table.field.event.key	**aggregateid**	Table	Specifies the outbox table column that contains the event key. When this column contains a value, the SMT uses that value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions.
table.field.event.timestamp		Table	By default, the timestamp in the emitted outbox message is the Debezium event timestamp. To use a different timestamp in outbox messages, set this option to an outbox table column that contains the timestamp that you want to be in emitted outbox messages.
table.field.event.payload	**payload**	Table	Specifies the outbox table column that contains the event payload.
table.field.event.payload.id	**aggregateid**	Table	Specifies the outbox table column that contains the payload ID. This ID will be used as the emitted event's key. This option is deprecated, use table.field.event.key instead.
table.expand.json.payload	**false**	Table	Specifies whether the JSON expansion of a String payload should be done. If no content found or in case of parsing error, the content is kept "as is". Fore more details, please see the expanding escaped json section.

Option	Default	Group	Description
table.fields.additional.placement		Table, Envelope	Specifies one or more outbox table columns that you want to add to outbox message headers or envelopes. Specify a comma-separated list of pairs. In each pair, specify the name of a column and whether you want the value to be in the header or the envelope. Separate the values in the pair with a colon, for example: **id:header,my-field:envelope** To specify an alias for the column, specify a trio with the alias as the third value, for example: **id:header,my-field:envelope:my-alias** The second value is the placement and it must always be **header** or **envelope**. Configuration examples are in emitting additional fields in Debezium outbox messages.
table.field.event.schema.version		Table, Schema	When set, this value is used as the schema version as described in the Kafka Connect Schema Javadoc.
route.by.field	**aggregatetype**	Router	Specifies the name of a column in the outbox table. The default behavior is that the value in this column becomes a part of the name of the topic to which the connector emits the outbox messages. An example is in the description of the expected outbox table.
route.topic.regex	**(?<routedByValue>.*)**	Router	Specifies a regular expression that the outbox SMT applies in the RegexRouter to outbox table records. This regular expression is part of the setting of the **route.topic.replacement** SMT option. The default behavior is that the SMT replaces the default **${routedByValue}** variable in the setting of the **route.topic.replacement** SMT option with the setting of the **route.by.field** outbox SMT option.

Option	Default	Group	Description
route.topic.replacement	**outbox.event.${routedByValue}**	Router	Specifies the name of the topic to which the connector emits outbox messages. The default topic name is **outbox.event.** followed by the **aggregatetype** column value in the outbox table record. For example, if the **aggregatetype** value is **customers**, the topic name is **outbox.event.customers**. To change the topic name, you can: • Set the **route.by.field** option to a different column. • Set the route.topic.regex option to a different regular expression.
route.tombstone.on.empty.payload	**false**	Router	Indicates whether an empty or **null** payload causes the connector to emit a tombstone event.

12.7. CONFIGURING DEBEZIUM MONGODB CONNECTORS TO USE THE OUTBOX PATTERN

NOTE

This SMT is for use with the Debezium MongoDB connector only. For information about using the outbox event router SMT for relational databases, see Outbox event router.

The outbox pattern is a way to safely and reliably exchange data between multiple (micro) services. An outbox pattern implementation avoids inconsistencies between a service's internal state (as typically persisted in its database) and state in events consumed by services that need the same data.

To implement the outbox pattern in a Debezium application, configure a Debezium connector to:

- Capture changes in an outbox collection
- Apply the Debezium MongoDB outbox event router single message transformation (SMT)

A Debezium connector that is configured to apply the MongoDB outbox SMT should capture changes that occur in an outbox collection only. For more information, see Options for applying the transformation selectively.

A connector can capture changes in more than one outbox collection only if each outbox collection has the same structure.

NOTE

To use this SMT, operations on the actual business collection(s) and the insert into the outbox collection must be done as part of a multi-document transaction, which have been being supported since MongoDB 4.0, to prevent potential data inconsistencies between business collection(s) and outbox collection. For future update, to enable updating existing data and inserting outbox event in an ACID transaction without multi-document transactions, we have planned to support additional configurations for storing outbox events in a form of a sub-document of the existing collection, rather than an independent outbox collection.

For more information about the outbox pattern, see Reliable Microservices Data Exchange With the Outbox Pattern.

The following topics provide details:

- Section 12.7.1, "Example of a Debezium MongoDB outbox message"
- Section 12.7.2, "Outbox collection structure expected by Debezium mongodb outbox event router SMT"
- Section 12.7.3, "Basic Debezium MongoDB outbox event router SMT configuration"
- Section 12.7.5, "Using Avro as the payload format in Debezium MongoDB outbox messages"
- Section 12.7.6, "Emitting additional fields in Debezium MongoDB outbox messages"
- Section 12.7.8, "Options for configuring outbox event router transformation"

12.7.1. Example of a Debezium MongoDB outbox message

To understand how to configure the Debezium MongoDB outbox event router SMT, consider the following example of a Debezium outbox message:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "b2730779e1f596e275826f08"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": {\"$oid\": \"da8d6de63b7745ff8f4457db\"}, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\", \"customerId\": 123}"
}
```

A Debezium connector that is configured to apply the MongoDB outbox event router SMT generates the preceding message by transforming a raw Debezium change event message as in the following example:

```
# Kafka Message key: { "id": "{\"$oid\": \"596e275826f08b2730779e1f\"}" }
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484
{
  "patch": null,
  "after": "{\"_id\": {\"$oid\": \"596e275826f08b2730779e1f\"}, \"aggregateid\": {\"$oid\":
```

```
\"b2730779e1f596e275826f08\"}, \"aggregatetype\": \"Order\", \"type\": \"OrderCreated\", \"payload\":
{\"_id\": {\"$oid\": \"da8d6de63b7745ff8f4457db\"}, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in
Action\", \"status\": \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium
for Dummies\", \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-
31T12:13:01\", \"customerId\": 123}}",
  "source": {
    "version": "1.9.7.Final",
    "connector": "mongodb",
    "name": "fulfillment",
    "ts_ms": 1558965508000,
    "snapshot": false,
    "db": "inventory",
    "rs": "rs0",
    "collection": "customers",
    "ord": 31,
    "h": 1546547425148721999
  },
  "op": "c",
  "ts_ms": 1556890294484
}
```


This example of a Debezium outbox message is based on the default outbox event router configuration, which assumes an outbox collection structure and event routing based on aggregates. To customize behavior, the outbox event router SMT provides numerous configuration options.

12.7.2. Outbox collection structure expected by Debezium mongodb outbox event router SMT

To apply the default MongoDB outbox event router SMT configuration, your outbox collection is assumed to have the following fields:

```
{
  "_id": "objectId",
  "aggregatetype": "string",
  "aggregateid": "objectId",
  "type": "string",
  "payload": "object"
}
```

Table 12.8. Descriptions of expected outbox collection fields

Field	Effect
id	Contains the unique ID of the event. In an outbox message, this value is a header. You can use this ID, for example, to remove duplicate messages. To obtain the unique ID of the event from a different outbox collection field, set the **collection.field.event.id** SMT option in the connector configuration.

Field	Effect
aggregatetype	Contains a value that the SMT appends to the name of the topic to which the connector emits an outbox message. The default behavior is that this value replaces the default **${routedByValue}** variable in the **route.topic.replacement** SMT option. For example, in a default configuration, the **route.by.field** SMT option is set to **aggregatetype** and the **route.topic.replacement** SMT option is set to **outbox.event.${routedByValue}**. Suppose that your application adds two documents to the outbox collection. In the first document, the value in the **aggregatetype** field is **customers**. In the second document, the value in the **aggregatetype** field is **orders**. The connector emits the first document to the **outbox.event.customers** topic. The connector emits the second document to the **outbox.event.orders** topic. To obtain this value from a different outbox collection field, set the **route.by.field** SMT option in the connector configuration.
aggregateid	Contains the event key, which provides an ID for the payload. The SMT uses this value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions. To obtain the event key from a different outbox collection field, set the **collection.field.event.key** SMT option in the connector configuration.
payload	A representation of the outbox change event. The default structure is JSON. By default, the Kafka message value is solely comprised of the **payload** value. However, if the outbox event is configured to include additional fields, the Kafka message value contains an envelope encapsulating both payload and the additional fields, and each field is represented separately. For more information, see Emitting messages with additional fields. To obtain the event payload from a different outbox collection field, set the **collection.field.event.payload** SMT option in the connector configuration.
Additional custom fields	Any additional fields from the outbox collection can be added to outbox events either within the payload section or as a message header. One example could be a field **eventType** which conveys a user-defined value that helps to categorize or organize events.

12.7.3. Basic Debezium MongoDB outbox event router SMT configuration

To configure a Debezium connector to support the outbox pattern, configure the **outbox.EventRouter** SMT. The following example shows the basic configuration for the SMT in a **.properties** file:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
```

Customizing the configuration

The connector might emit many types of event messages (for example, heartbeat messages, tombstone messages, or metadata messages about transactions). To apply the transformation only to events that originate in the outbox collection, define an SMT predicate statement that selectively applies the transformation to those events only.

12.7.4. Options for applying the MongoDB outbox event router transformation selectively

In addition to the change event messages that a Debezium connector emits when a database change occurs, the connector also emits other types of messages, including heartbeat messages, and metadata messages about schema changes and transactions. Because the structure of these other messages differs from the structure of the change event messages that the SMT is designed to process, it's best to configure the connector to selectively apply the SMT, so that it processes only the intended data change messages. You can use one of the following methods to configure the connector to apply the SMT selectively:

- Configure an SMT predicate for the transformation.
- Use the **route.topic.regex** configuration option for the SMT.

12.7.5. Using Avro as the payload format in Debezium MongoDB outbox messages

The MongoDB outbox event router SMT supports arbitrary payload formats. The **payload** field value in an outbox collection is passed on transparently. An alternative to working with JSON is to use Avro. This can be beneficial for message format governance and for ensuring that outbox event schemas evolve in a backwards-compatible way.

How a source application produces Avro formatted content for outbox message payloads is out of the scope of this documentation. One possibility is to leverage the **KafkaAvroSerializer** class to serialize **GenericRecord** instances. To ensure that the Kafka message value is the exact Avro binary data, apply the following configuration to the connector:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
value.converter=io.debezium.converters.ByteArrayConverter
```

By default, the **payload** field value (the Avro data) is the only message value. Configuration of **ByteArrayConverter** as the value converter propagates the **payload** field value as-is into the Kafka message value.

Note that this differs from the **ByteBufferConverter** suggested for other SMTs. This is due to the different approach MongoDB takes to storing byte arrays internally.

The Debezium connectors may be configured to emit heartbeat, transaction metadata, or schema change events (support varies by connector). These events cannot be serialized by the **ByteArrayConverter** so additional configuration must be provided so the converter knows how to serialize these events. As an example, the following configuration illustrates using the Apache Kafka **JsonConverter** with no schemas:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
value.converter=io.debezium.converters.ByteArrayConverter
```

```
value.converter.delegate.converter.type=org.apache.kafka.connect.json.JsonConverter
value.converter.delegate.converter.type.schemas.enable=false
```

The delegate **Converter** implementation is specified by the **delegate.converter.type** option. If any extra configuration options are needed by the converter, they can also be specified, such as the disablement of schemas shown above using **schemas.enable=false**.

12.7.6. Emitting additional fields in Debezium MongoDB outbox messages

Your outbox collection might contain fields whose values you want to add to the emitted outbox messages. For example, consider an outbox collection that has a value of **purchase-order** in the **aggregatetype** field and another field, **eventType**, whose possible values are **order-created** and **order-shipped**. Additional fields can be added with the syntax **field:placement:alias**.

The allowed values for **placement** are: - **header** - **envelope** - **partition**

To emit the **eventType** field value in the outbox message header, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:header:type
```

The result will be a header on the Kafka message with **type** as its key, and the value of the **eventType** field as its value.

To emit the **eventType** field value in the outbox message envelope, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:envelope:type
```

To control which partition the outbox message is produced on, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=partitionField:partition
```

Note that for the **partition** placement, adding an alias will have no effect.

12.7.7. Expanding escaped JSON String as JSON

By default, the **payload** of the Debezium outbox message is represented as a string. When the original source of the string is in JSON format, the resulting Kafka message uses escape sequences to represent the string, as shown in the following example:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": {\"$oid\": \"da8d6de63b7745ff8f4457db\"}, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in
Action\", \"status\": \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium
```

```
for Dummies\", \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\", \"customerId\": 123}"
}
```

You can configure the outbox event router to expand the message content, converting the escaped JSON back to its original, unescaped JSON format. In the converted string, the companion schema is deduced from the original JSON document. The following examples shows the expanded JSON in the resulting Kafka message:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "id": "da8d6de63b7745ff8f4457db", "lineItems": [{"id": 1, "item": "Debezium in Action", "status": "ENTERED", "quantity": 2, "totalPrice": 39.98}, {"id": 2, "item": "Debezium for Dummies", "status": "ENTERED", "quantity": 1, "totalPrice": 29.99}], "orderDate": "2019-01-31T12:13:01", "customerId": 123
}
```

To enable string conversion in the transformation, set the value of **collection.expand.json.payload** to **true** and use the **StringConverter** as shown in the following example:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
transforms.outbox.collection.expand.json.payload=true
value.converter=org.apache.kafka.connect.storage.StringConverter
```

12.7.8. Options for configuring outbox event router transformation

The following table describes the options that you can specify for the outbox event router SMT. In the table, the **Group** column indicates a configuration option classification for Kafka.

Table 12.9. Descriptions of outbox event router SMT configuration options

Option	Default	Group	Description

Option	Default	Group	Description
collection.op.invalid.behavior	**warn**	Collection	Determines the behavior of the SMT when there is an update operation on the outbox collection. Possible settings are: • **warn** - The SMT logs a warning and continues to the next outbox collection document. • **error** - The SMT logs an error and continues to the next outbox collection document. • **fatal** - The SMT logs an error and the connector stops processing. All changes in an outbox collection are expected to be an insert or delete operation. That is, an outbox collection functions as a queue; updates to documents in an outbox collection are not allowed. The SMT automatically filters out delete operations (for removing proceeded outbox events) on an outbox collection.
collection.field.event.id	**_id**	Collection	Specifies the outbox collection field that contains the unique event ID. This ID will be stored in the emitted event's headers under the **id** key.
collection.field.event.key	**aggregateid**	Collection	Specifies the outbox collection field that contains the event key. When this field contains a value, the SMT uses that value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions.
collection.field.event.timestamp		Collection	By default, the timestamp in the emitted outbox message is the Debezium event timestamp. To use a different timestamp in outbox messages, set this option to an outbox collection field that contains the timestamp that you want to be in emitted outbox messages.
collection.field.event.payload	**payload**	Collection	Specifies the outbox collection field that contains the event payload.

Option	Default	Group	Description
collection.expand.json.payload	**false**	Collection	Specifies whether the JSON expansion of a String payload should be done. If no content found or in case of parsing error, the content is kept "as is". Fore more details, please see the expanding escaped json section.
collection.fields.additional.placement		Collection, Envelope	Specifies one or more outbox collection fields that you want to add to outbox message headers or envelopes. Specify a comma-separated list of pairs. In each pair, specify the name of a field and whether you want the value to be in the header or the envelope. Separate the values in the pair with a colon, for example: **id:header,my-field:envelope** To specify an alias for the field, specify a trio with the alias as the third value, for example: **id:header,my-field:envelope:my-alias** The second value is the placement and it must always be **header** or **envelope**. Configuration examples are in emitting additional fields in Debezium outbox messages.
collection.field.event.schema.version		Collection, Schema	When set, this value is used as the schema version as described in the Kafka Connect Schema Javadoc.
route.by.field	**aggregatetype**	Router	Specifies the name of a field in the outbox collection. By default, the value specified in this field becomes a part of the name of the topic to which the connector emits the outbox messages. For an example, see the description of the expected outbox collection.

Option	Default	Group	Description
route.topic.regex	**(?<routedByValue>.*)**	Router	Specifies a regular expression that the outbox SMT applies in the RegexRouter to outbox collection documents. This regular expression is part of the setting of the route.topic.replacement SMT option. + The default behavior is that the SMT replaces the default **${routedByValue}** variable in the setting of the **route.topic.replacement** SMT option with the setting of the route.by.field outbox SMT option.
route.topic.replacement	**outbox.event.${routedByValue}**	Router	Specifies the name of the topic to which the connector emits outbox messages. The default topic name is **outbox.event.** followed by the **aggregatetype** field value in the outbox collection document. For example, if the **aggregatetype** value is **customers**, the topic name is **outbox.event.customers**. + To change the topic name, you can: ● Set the route.by.field option to a different field. ● Set the route.topic.regex option to a different regular expression.
route.tombstone.on.empty.payload	**false**	Router	Indicates whether an empty or **null** payload causes the connector to emit a tombstone event.

Revised on 2022-11-17 15:03:09 UTC