



Red Hat Integration 2021.Q3

Debezium User Guide

For use with Debezium 1.5

Red Hat Integration 2021.Q3 Debezium User Guide

For use with Debezium 1.5

Legal Notice

Copyright © 2021 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	7
MAKING OPEN SOURCE MORE INCLUSIVE	7
CHAPTER 1. HIGH LEVEL OVERVIEW OF DEBEZIUM	8
1.1. DEBEZIUM FEATURES	8
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
3.2.2. How Debezium Db2 connectors read change-data tables	14
3.2.3. Default names of Kafka topics that receive Debezium Db2 change event records	15
3.2.4. About the Debezium Db2 connector schema change topic	15
3.2.5. Debezium Db2 connector-generated events that represent transaction boundaries	19
3.3. DESCRIPTIONS OF DEBEZIUM DB2 CONNECTOR DATA CHANGE EVENTS	20
3.3.1. About keys in Debezium Db2 change events	22
3.3.2. About values in Debezium Db2 change events	23
3.4. HOW DEBEZIUM DB2 CONNECTORS MAP DATA TYPES	32
3.5. SETTING UP DB2 TO RUN A DEBEZIUM CONNECTOR	37
3.5.1. Configuring Db2 tables for change data capture	38
3.5.2. Effect of Db2 capture agent configuration on server load and latency	39
3.5.3. Db2 capture agent configuration parameters	40
3.6. DEPLOYMENT OF DEBEZIUM DB2 CONNECTORS	40
3.6.1. Deploying Debezium Db2 connectors	41
3.6.2. Description of Debezium Db2 connector configuration properties	45
3.7. MONITORING DEBEZIUM DB2 CONNECTOR PERFORMANCE	58
3.7.1. Monitoring Debezium during snapshots of Db2 databases	58
3.7.2. Monitoring Debezium Db2 connector record streaming	60
3.7.3. Monitoring Debezium Db2 connector schema history	61
3.8. MANAGING DEBEZIUM DB2 CONNECTORS	62
3.9. UPDATING SCHEMAS FOR DB2 TABLES IN CAPTURE MODE FOR DEBEZIUM CONNECTORS	62
3.9.1. Performing offline schema updates for Debezium Db2 connectors	63
3.9.2. Performing online schema updates for Debezium Db2 connectors	64
CHAPTER 4. DEBEZIUM CONNECTOR FOR MONGODB	66
4.1. OVERVIEW OF DEBEZIUM MONGODB CONNECTOR	66
4.2. HOW DEBEZIUM MONGODB CONNECTORS WORK	67
4.2.1. MongoDB topologies supported by Debezium connectors	68
4.2.2. How Debezium MongoDB connectors use logical names for replica sets and sharded clusters	69
4.2.3. How Debezium MongoDB connectors perform snapshots	69
4.2.4. How the Debezium MongoDB connector streams change event records	70
4.2.5. Default names of Kafka topics that receive Debezium MongoDB change event records	70
4.2.6. How event keys control topic partitioning for the Debezium MongoDB connector	71
4.2.7. Debezium MongoDB connector-generated events that represent transaction boundaries	71
4.3. DESCRIPTIONS OF DEBEZIUM MONGODB CONNECTOR DATA CHANGE EVENTS	72
4.3.1. About keys in Debezium MongoDB change events	74
4.3.2. About values in Debezium MongoDB change events	76
4.4. SETTING UP MONGODB TO WORK WITH A DEBEZIUM CONNECTOR	84
4.5. DEPLOYMENT OF DEBEZIUM MONGODB CONNECTORS	85

4.5.1. Deploying Debezium MongoDB connectors	85
4.5.2. Description of Debezium Db2 connector configuration properties	88
4.6. MONITORING DEBEZIUM MONGODB CONNECTOR PERFORMANCE	96
4.6.1. Monitoring Debezium during MongoDB snapshots	96
4.6.2. Monitoring Debezium MongoDB connector record streaming	97
4.7. HOW DEBEZIUM MONGODB CONNECTORS HANDLE FAULTS AND PROBLEMS	99
CHAPTER 5. DEBEZIUM CONNECTOR FOR MYSQL	103
5.1. HOW DEBEZIUM MYSQL CONNECTORS WORK	103
5.1.1. MySQL topologies supported by Debezium connectors	104
5.1.2. How Debezium MySQL connectors handle database schema changes	105
5.1.3. How Debezium MySQL connectors expose database schema changes	105
5.1.4. How Debezium MySQL connectors perform database snapshots	108
5.1.5. Default names of Kafka topics that receive Debezium MySQL change event records	110
5.2. DESCRIPTIONS OF DEBEZIUM MYSQL CONNECTOR DATA CHANGE EVENTS	112
5.2.1. About keys in Debezium MySQL change events	114
5.2.2. About values in Debezium MySQL change events	115
5.3. HOW DEBEZIUM MYSQL CONNECTORS MAP DATA TYPES	126
5.4. SETTING UP MYSQL TO RUN A DEBEZIUM CONNECTOR	132
5.4.1. Creating a MySQL user for a Debezium connector	132
5.4.2. Enabling the MySQL binlog for Debezium	133
5.4.3. Enabling MySQL Global Transaction Identifiers for Debezium	134
5.4.4. Configuring MySQL session timeouts for Debezium	135
5.4.5. Enabling query log events for Debezium MySQL connectors	136
5.5. DEPLOYMENT OF DEBEZIUM MYSQL CONNECTORS	137
5.5.1. Deploying Debezium MySQL connectors	137
5.5.2. Description of Debezium MySQL connector configuration properties	141
5.6. MONITORING DEBEZIUM MYSQL CONNECTOR PERFORMANCE	158
5.6.1. Monitoring Debezium during snapshots of MySQL databases	158
5.6.2. Monitoring Debezium MySQL connector record streaming	160
5.6.3. Monitoring Debezium MySQL connector schema history	163
5.7. HOW DEBEZIUM MYSQL CONNECTORS HANDLE FAULTS AND PROBLEMS	164
CHAPTER 6. DEBEZIUM CONNECTOR FOR ORACLE (DEVELOPER PREVIEW)	166
6.1. HOW DEBEZIUM ORACLE CONNECTORS WORK	166
6.1.1. How Debezium Oracle connectors perform database snapshots	167
6.1.2. Default names of Kafka topics that receive Debezium Oracle change event records	168
6.1.3. How Debezium Oracle connectors expose database schema changes	168
6.1.4. Debezium Oracle connector-generated events that represent transaction boundaries	171
6.1.4.1. Change data event enrichment	172
6.2. DESCRIPTIONS OF DEBEZIUM ORACLE CONNECTOR DATA CHANGE EVENTS	173
6.2.1. About keys in Debezium Oracle connector change events	173
6.2.2. About values in Debezium Oracle connector change events	174
6.3. HOW DEBEZIUM ORACLE CONNECTORS MAP DATA TYPES	181
6.4. SETTING UP ORACLE TO WORK WITH DEBEZIUM	186
6.4.1. Preparing Oracle databases for use with Debezium	186
6.4.2. Creating an Oracle user for the Debezium Oracle connector	187
6.5. DEPLOYMENT OF DEBEZIUM ORACLE CONNECTORS	188
6.5.1. Obtaining the Oracle JDBC driver	188
6.5.2. Deploying Debezium Oracle connectors	189
6.5.3. Descriptions of Debezium Oracle connector configuration properties	193
6.6. MONITORING DEBEZIUM ORACLE CONNECTOR PERFORMANCE	207
6.6.1. Debezium Oracle connector snapshot metrics	207

6.6.2. Debezium Oracle connector streaming metrics	208
6.6.3. Debezium Oracle connector schema history metrics	215
6.7. HOW DEBEZIUM ORACLE CONNECTORS HANDLE FAULTS AND PROBLEMS	216
CHAPTER 7. DEBEZIUM CONNECTOR FOR POSTGRESQL	217
7.1. OVERVIEW OF DEBEZIUM POSTGRESQL CONNECTOR	217
7.2. HOW DEBEZIUM POSTGRESQL CONNECTORS WORK	218
7.2.1. Security for PostgreSQL connector	219
7.2.2. How Debezium PostgreSQL connectors perform database snapshots	219
7.2.3. How Debezium PostgreSQL connectors stream change event records	220
7.2.4. Default names of Kafka topics that receive Debezium PostgreSQL change event records	221
7.2.5. Metadata in Debezium PostgreSQL change event records	222
7.2.6. Debezium PostgreSQL connector-generated events that represent transaction boundaries	223
7.3. DESCRIPTIONS OF DEBEZIUM POSTGRESQL CONNECTOR DATA CHANGE EVENTS	224
7.3.1. About keys in Debezium PostgreSQL change events	226
7.3.2. About values in Debezium PostgreSQL change events	228
7.4. HOW DEBEZIUM POSTGRESQL CONNECTORS MAP DATA TYPES	240
7.5. SETTING UP POSTGRESQL TO RUN A DEBEZIUM CONNECTOR	250
7.5.1. Configuring a replication slot for the Debezium pgoutput plug-in	250
7.5.2. Setting up PostgreSQL permissions for the Debezium connector	251
7.5.3. Setting privileges to enable Debezium to create PostgreSQL publications	251
7.5.4. Configuring PostgreSQL to allow replication with the Debezium connector host	252
7.5.5. Configuring PostgreSQL to manage Debezium WAL disk space consumption	253
7.6. DEPLOYMENT OF DEBEZIUM POSTGRESQL CONNECTORS	253
7.6.1. Deploying Debezium PostgreSQL connectors	254
7.6.2. Description of Debezium PostgreSQL connector configuration properties	257
7.7. MONITORING DEBEZIUM POSTGRESQL CONNECTOR PERFORMANCE	277
7.7.1. Monitoring Debezium during snapshots of PostgreSQL databases	277
7.7.2. Monitoring Debezium PostgreSQL connector record streaming	279
7.8. HOW DEBEZIUM POSTGRESQL CONNECTORS HANDLE FAULTS AND PROBLEMS	280
CHAPTER 8. DEBEZIUM CONNECTOR FOR SQL SERVER	284
8.1. OVERVIEW OF DEBEZIUM SQL SERVER CONNECTOR	284
8.2. HOW DEBEZIUM SQL SERVER CONNECTORS WORK	285
8.2.1. How Debezium SQL Server connectors perform database snapshots	285
8.2.2. How Debezium SQL Server connectors read change data tables	286
8.2.3. Default names of Kafka topics that receive Debezium SQL Server change event records	286
8.2.4. How the Debezium SQL Server connector uses the schema change topic	287
8.2.5. Descriptions of Debezium SQL Server connector data change events	290
8.2.5.1. About keys in Debezium SQL Server change events	292
8.2.5.2. About values in Debezium SQL Server change events	293
8.2.6. Debezium SQL Server connector-generated events that represent transaction boundaries	302
8.2.6.1. Change data event enrichment	304
8.2.7. How Debezium SQL Server connectors map data types	304
8.3. SETTING UP SQL SERVER TO RUN A DEBEZIUM CONNECTOR	310
8.3.1. Enabling CDC on the SQL Server database	310
8.3.2. Enabling CDC on a SQL Server table	311
8.3.3. Verifying that the user has access to the CDC table	312
8.3.4. SQL Server on Azure	313
8.3.5. Effect of SQL Server capture job agent configuration on server load and latency	313
8.3.6. SQL Server capture job agent configuration parameters	313
8.4. DEPLOYMENT OF DEBEZIUM SQL SERVER CONNECTORS	314
8.4.1. Deploying Debezium SQL Server connectors	314

8.4.2. Descriptions of Debezium SQL Server connector configuration properties	318
8.5. REFRESHING CAPTURE TABLES AFTER A SCHEMA CHANGE	331
8.5.1. Running an offline update after a schema change	332
8.5.2. Running an online update after a schema change	333
8.6. MONITORING DEBEZIUM SQL SERVER CONNECTOR PERFORMANCE	335
8.6.1. Debezium SQL Server connector snapshot metrics	335
8.6.2. Debezium SQL Server connector streaming metrics	337
8.6.3. Debezium SQL Server connector schema history metrics	338
CHAPTER 9. MONITORING DEBEZIUM	340
9.1. METRICS FOR MONITORING DEBEZIUM CONNECTORS	340
9.2. ENABLING JMX IN LOCAL INSTALLATIONS	340
9.2.1. Zookeeper JMX environment variables	340
9.2.2. Kafka JMX environment variables	341
9.2.3. Kafka Connect JMX environment variables	341
9.3. MONITORING DEBEZIUM ON OPENSIFT	341
CHAPTER 10. DEBEZIUM LOGGING	342
10.1. DEBEZIUM LOGGING CONCEPTS	342
10.2. THE DEFAULT DEBEZIUM LOGGING CONFIGURATION	342
10.3. CONFIGURING DEBEZIUM LOGGING	343
10.3.1. Changing the Debezium logging level	343
10.3.2. Adding Debezium mapped diagnostic contexts	344
10.4. DEBEZIUM LOGGING ON OPENSIFT	346
CHAPTER 11. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION	347
11.1. CUSTOMIZATION OF KAFKA CONNECT AUTOMATIC TOPIC CREATION	347
11.1.1. Disabling automatic topic creation for the Kafka broker	348
11.1.2. Configuring automatic topic creation in Kafka Connect	348
11.1.3. Configuration of automatically created topics	349
11.1.3.1. Topic creation groups	349
11.1.3.2. Topic creation group configuration properties	349
11.1.3.3. Specifying the configuration for the Debezium default topic creation group	350
11.1.3.4. Specifying the configuration for Debezium custom topic creation groups	351
11.1.3.5. Registering Debezium custom topic creation groups	352
11.2. CONFIGURING DEBEZIUM CONNECTORS TO USE AVRO SERIALIZATION	353
11.2.1. About the Service Registry	354
11.2.2. Overview of deploying a Debezium connector that uses Avro serialization	355
11.2.3. Deploying connectors that use Avro in Debezium containers	355
11.2.4. About Avro name requirements	359
11.3. EMITTING DEBEZIUM CHANGE EVENT RECORDS IN CLOUDEVENTS FORMAT	359
11.3.1. Example Debezium change event records in CloudEvents format	360
11.3.2. Example of configuring Debezium CloudEvents converter	362
11.3.3. Debezium CloudEvents converter configuration options	362
CHAPTER 12. APPLYING TRANSFORMATIONS TO MODIFY MESSAGES EXCHANGED WITH APACHE KAFKA	364
12.1. APPLYING TRANSFORMATIONS SELECTIVELY WITH SMT PREDICATES	364
12.1.1. About SMT predicates	364
12.1.2. Defining SMT predicates	366
12.1.3. Ignoring tombstone events	367
12.2. ROUTING DEBEZIUM EVENT RECORDS TO TOPICS THAT YOU SPECIFY	368
12.2.1. Use case for routing Debezium records to topics that you specify	368
12.2.2. Example of routing Debezium records for multiple tables to one topic	369

12.2.3. Ensuring unique keys across Debezium records routed to the same topic	370
12.2.4. Options for applying the topic routing transformation selectively	371
12.2.5. Options for configuring Debezium topic routing transformation	371
12.3. ROUTING CHANGE EVENT RECORDS TO TOPICS ACCORDING TO EVENT CONTENT	372
12.3.1. Setting up the Debezium content-based-routing SMT	373
12.3.2. Example: Debezium basic content-based routing configuration	373
12.3.3. Variables for use in Debezium content-based routing expressions	374
12.3.4. Options for applying the content-based routing transformation selectively	375
12.3.5. Configuration of content-based routing conditions for other scripting languages	375
12.3.6. Options for configuring the content-based routing transformation	376
12.4. FILTERING DEBEZIUM CHANGE EVENT RECORDS	377
12.4.1. Setting up the Debezium filter SMT	377
12.4.2. Example: Debezium basic filter SMT configuration	378
12.4.3. Variables for use in filter expressions	379
12.4.4. Options for applying the filter transformation selectively	379
12.4.5. Filter condition configuration for other scripting languages	380
12.4.6. Options for configuring filter transformation	380
12.5. EXTRACTING SOURCE RECORD AFTER STATE FROM DEBEZIUM CHANGE EVENTS	381
12.5.1. Description of Debezium change event structure	382
12.5.2. Behavior of Debezium event flattening transformation	383
12.5.3. Configuration of Debezium event flattening transformation	383
12.5.4. Example of adding Debezium metadata to the Kafka record	384
12.5.5. Options for applying the event flattening transformation selectively	385
12.5.6. Options for configuring Debezium event flattening transformation	385
12.6. CONFIGURING DEBEZIUM CONNECTORS TO USE THE OUTBOX PATTERN	389
12.6.1. Example of a Debezium outbox message	390
12.6.2. Outbox table structure expected by Debezium outbox event router SMT	391
12.6.3. Basic Debezium outbox event router SMT configuration	392
12.6.4. Options for applying the Outbox event router transformation selectively	393
12.6.5. Using Avro as the payload format in Debezium outbox messages	393
12.6.6. Emitting additional fields in Debezium outbox messages	393
12.6.7. Options for configuring outbox event router transformation	394

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 \(Developer Preview\)*](#) (Developer Preview)
- [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. 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.
- Ready-to-use **message transformations** for:
 - [Message routing](#)
 - [Content-based routing](#)
 - [Extraction of new record state for relational connectors](#)

- [Filtering](#)
- [Routing of events](#) from a transactional outbox table

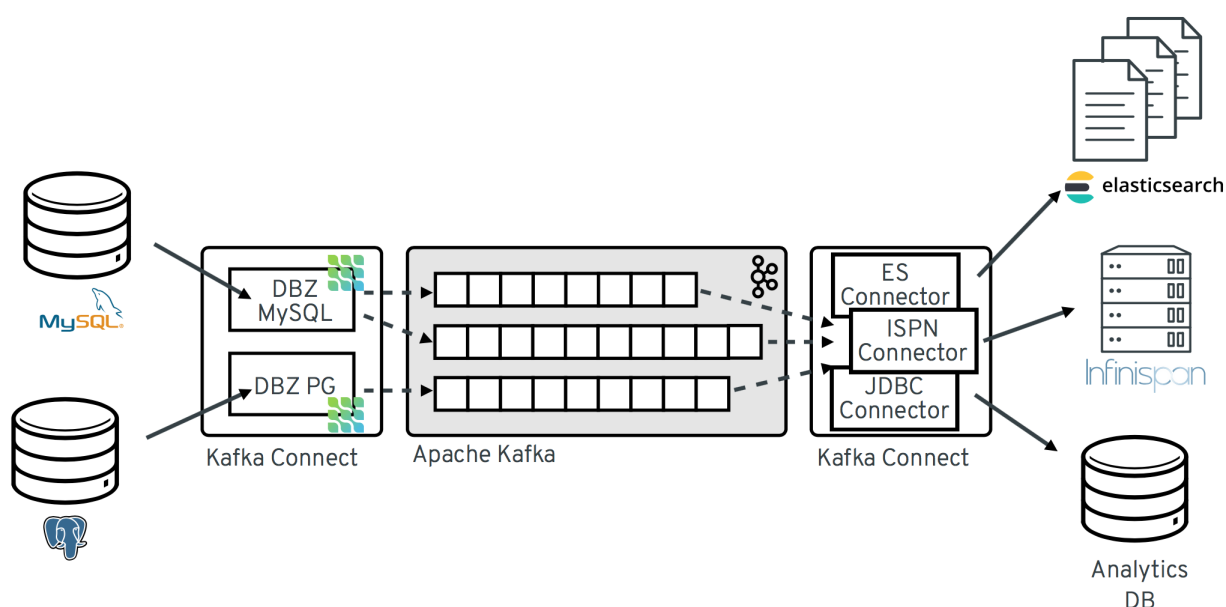
The documentation for each connector provides details about the connectors features and configuration options.

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 PostgreSQL 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 Streams2021.q3, 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 Streams2021.q3 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 **{ApiVersion}**. For more information about the tool and for the complete instructions about how to upgrade AMQ Streams, see [NameDeployStreamsOpenShift](#).



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. 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.

3.2.2. 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 this 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 insert, update, and delete operations on a single table to a single Kafka topic. The name of the Kafka topic has the following format:

databaseName.schemaName.tableName

databaseName

The logical name of the connector as specified with 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 **MYSHEMA** schema. The connector would emit events to these four Kafka topics:

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

To configure a Db2 connector to emit change events to differently-named Kafka topics, see [Routing Debezium event records to topics that you specify](#).

3.2.4. About the Debezium Db2 connector schema change topic

For a table that is in capture mode, the Debezium Db2 connector stores the history of schema changes to that table in a database history topic. This topic reflects an internal connector state and you should not use it. If your application needs to track schema changes, there is a public schema change topic. The name of the schema change topic is the same as the logical server name specified in the connector configuration.

**WARNING**

The format of 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:

- 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.

A message to the schema change topic contains a logical representation of the table schema, for example:

```
{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "1.5.4.Final",
      "connector": "db2",
      "name": "db2",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "testdb",
      "schema": "DB2INST1",
      "table": "CUSTOMERS",
      "change_isn": null,
      "commit_isn": "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 3.1. 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.
4	type	Describes the kind of change. The value is one of the following: <ul style="list-style-type: none"> ● 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 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.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. 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 the given 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
    }
  ]
}

```

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** - 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 3.2. Overview of change event basic content

Item	Field name	Description
1	schema	<p>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.</p> <p>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 the key identified by that property.</p>
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 3.3. 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	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.MY SCHEMA.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 <i>connector-name.database-name.table-name.Key</i> . In this example: <ul style="list-style-type: none"> ● 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.MYSHEMA.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": "1.5.4.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 3.4. 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	<p>In the schema section, each name field specifies the schema for a field in the value's payload.</p> <p>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.</p> <p>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.</p>
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	<p>The value's actual data. This is the information that the change event is providing.</p> <p>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.</p>

Item	Field name	Description
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.
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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • 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. <p>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.</p>

update events

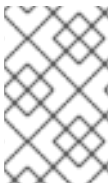
The value of a change event for an update in the sample **customers** table has the same schema as a *create* event for that table. Likewise, the *update* event value's payload has the same structure. However, the event value payload contains different values in an *update* event. Here is an example 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": "1.5.4.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
    },
    "op": "u", 4
    "ts_ms": 1559729998706 5
  }
}
```

Table 3.5. 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 <i>update</i> 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	<p>Mandatory field that describes the source metadata for the event. The source field structure contains the same fields as in a <i>create</i> event, but some values are different, for example, the sample <i>update</i> 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:</p> <ul style="list-style-type: none"> • 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	<p>Mandatory string that describes the type of operation. In an <i>update</i> event value, the op field value is u, signifying that this row changed because of an update.</p>
5	ts_ms	<p>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.</p> <p>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.</p>



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": "1.5.4.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 3.6. 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 <i>delete</i> 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 <i>delete</i> event value, the after field is null , signifying that the row no longer exists.
3	source	<p>Mandatory field that describes the source metadata for the event. In a <i>delete</i> event value, the source field structure is the same as for <i>create</i> and <i>update</i> events for the same table. Many source field values are also the same. In a <i>delete</i> event value, the ts_ms and LSN field values, as well as other values, might have changed. But the source field in a <i>delete</i> event value provides the same metadata:</p> <ul style="list-style-type: none"> ● 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.

Details are in the following sections:

- [Basic types](#)
- [Temporal types](#)
- [Timestamp types](#)
- [Table 3.10, "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 3.7. 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 3.8. 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 3.9. 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 3.10. 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 is 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 **db2inst1** 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 **db2inst1** user so the UDFs should be installed on the **db2inst1** user.

```
db2 connect to DB_NAME
```

- Copy the Debezium management UDFs and set permissions for them:

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

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

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

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

- Create the ASN control tables:

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

- 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.

- Start the ASN agent:

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

- Put tables into capture mode. Invoke the following statement for each table that you want to put into capture. Replace **MYSHEMA** 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('MYSHEMA', 'MYTABLE');
```

- 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

To deploy a Debezium Db2 connector, you add the connector files to Kafka Connect, create a custom container to run the connector, and then add the connector configuration to your container. For details about deploying the Debezium Db2 connector, see the following topics:

- [Section 3.6.1, “Deploying Debezium Db2 connectors”](#)
- [Section 3.6.2, “Description of Debezium Db2 connector configuration properties”](#)

3.6.1. Deploying Debezium Db2 connectors

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.
- 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. Download the Debezium [Db2 connector archive](#).
 - b. Extract the Debezium Db2 connector archive to create a directory structure for the connector plug-in, for example:

```

./my-plugins/
├── debezium-connector-db2
└── ...

```

- c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```

cat <<EOF >debezium-container-for-db2.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF

```

1 1 1 1 1 1 1 You can specify any file name that you want.

2 2 2 2 2 2 2 Replace **my-plugins** with the name of your plug-ins directory.

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

- d. 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**.

- e. 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
```

- f. 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
```

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.

- g. 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: fulfillment 10
    database.include.list: public.inventory 11

```

Table 3.11. 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.

Item	Description
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.
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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```

- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...
... INFO Snapshot is using user 'debezium' ...
```

If the connector starts correctly without errors, it creates a topic for each table whose changes the connector is capturing. For the example CR, there would be a topic for the table specified in the **include.list** property. Downstream applications can subscribe to these topics.

- c. Verify that the connector created the topics by running the following command:

```
oc get kafkatopics
```

For the complete list of the configuration properties that you can set for the Debezium Db2 connector, see [Db2 connector properties](#).

Results

When 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.

3.6.2. Description 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		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.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		IP address or hostname of the Db2 database server.
database.port	50000	Integer port number of the Db2 database server.
database.user		Name of the Db2 database user for connecting to the Db2 database server.

Property	Default	Description
database.password		Password to use when connecting to the Db2 database server.
database.dbname		The name of the Db2 database from which to stream the changes
database.server.name		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 and underscores should 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.
table.include.list		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 <i>schemaName.tableName</i> . By default, the connector captures changes in every non-system table. Do not also set the table.exclude.list property.
table.exclude.list		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 <i>schemaName.tableName</i> . Do not also set the table.include.list property.
column.exclude.list	<i>empty string</i>	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 <i>schemaName.tableName.columnName</i> . Primary key columns are always included in the event's key, even if they are excluded from the value.

Property	Default	Description
<p>column.mask.hash.hashAlgorithm.with.salt.salt</p>	<p>n/a</p>	<p>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 <i>schemaName.tableName.columnName</i>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. 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.</p> <p>In the following example, CzQMA0cB5K is a randomly selected salt.</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>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.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p>
<p>time.precision.mode</p>	<p>adaptive</p>	<p>Time, date, and timestamps can be represented with different kinds of precision:</p> <p>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.</p> <p>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.</p>

Property	Default	Description
tombstones.on.delete	true	<p>Controls whether a <i>delete</i> event is followed by a tombstone event.</p> <p>true - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event.</p> <p>false - only a <i>delete</i> event is emitted.</p> <p>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.</p>
include.schema.changes	true	<p>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.</p>
column.truncate.to._length_chars	<i>n/a</i>	<p>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 <i>schemaName.tableName.columnName</i>. In change event records, values in these columns are truncated if they are longer than the number of characters specified by <i>length</i> 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.</p>

Property	Default	Description
<code>column.mask.with._length_chars</code>	<i>n/a</i>	<p>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 <i>schemaName.tableName.columnName</i>. In change event values, the values in the specified table columns are replaced with <i>length</i> 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.</p>
<code>column.propagate.source.type</code>	<i>n/a</i>	<p>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 <i>databaseName.tableName.columnName</i>, or <i>databaseName.schemaName.tableName.columnName</i>.</p> <p>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:</p> <p>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</p> <p>This property is useful for properly sizing corresponding columns in sink databases.</p>

Property	Default	Description
datatype.propagate.source.type	<i>n/a</i>	<p>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 <i>databaseName.tableName.typeName</i>, or <i>databaseName.schemaName.tableName.typeName</i>.</p> <p>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:</p> <p>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</p> <p>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.</p> <p>See Db2 data types for the list of Db2-specific data type names.</p>

Property	Default	Description
message.key.columns	<i>empty string</i>	<p>A semicolon separated list of tables with regular expressions that match table column names. The connector maps values in matching columns to key fields in change event records that it sends to Kafka topics. This is useful when a table does not have a primary key, or when you want to order change event records in a Kafka topic according to a field that is not a primary key.</p> <p>Separate entries with semicolons. Insert a colon between the fully-qualified table name and its regular expression. The format is:</p> <pre>schema-name.table-name:_regexp_;</pre> <p>For example,</p> <pre>schemaA.table_a:regex_1;schemaB.table_b:regex_2;schemaC.table_c:regex_3</pre> <p>If table_a has an id column, and regex_1 is ^i (matches any column that starts with i), the connector maps the value in table_a's id column to a key field in change events that the connector sends to Kafka.</p>

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
snapshot.mode	initial	<p>Specifies the criteria for performing a snapshot when the connector starts:</p> <p>initial - For tables in capture mode, the connector takes a snapshot of the schema for the table and the data in the table. This is useful for populating Kafka topics with a complete representation of the data.</p> <p>schema_only - For tables in capture mode, the connector takes a snapshot of only the schema for the table. This is useful when only the changes that are happening from now on need to be emitted to Kafka topics. After the snapshot is complete, the connector continues by reading change events from the database's redo logs.</p>
snapshot.isolation.mode	repeatable_read	<p>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:</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>

Property	Default	Description
event.processing.failure.handling.mode	fail	<p>Specifies how the connector handles exceptions during processing of events. The possible values are:</p> <p>fail - The connector logs the offset of the problematic event and stops processing.</p> <p>warn - The connector logs the offset of the problematic event and continues processing with the next event.</p> <p>skip - The connector skips the problematic event and continues processing with the next event.</p>
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.queue.size	8192	Positive integer value for the maximum size of the blocking queue. The connector places change events that it reads from the database log into the blocking queue before writing them to Kafka. This queue can provide backpressure for reading change-data tables when, for example, writing records to Kafka is slower than it should be or Kafka is not available. Events that appear in the queue are not included in the offsets that are periodically recorded by the connector. The max.queue.size value should always be larger than the value of the max.batch.size connector configuration property.
max.batch.size	2048	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
max.queue.size.in.bytes	0	Long value for the maximum size in bytes of the blocking queue. The feature is disabled by default, it will be active if it's set with a positive long value.

Property	Default	Description
heartbeat.interval.ms	0	<p>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</p> <p>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.</p> <p>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.</p>
heartbeat.topics.prefix	<code>__debezium-heartbeat</code>	<p>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>.</p>
snapshot.delay.ms		<p>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.</p>
snapshot.fetch.size	2000	<p>During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.</p>

Property	Default	Description
snapshot.lock.timeout.ms	10000	<p>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:</p> <p>0 - The connector immediately fails when it cannot obtain a lock.</p> <p>-1 - The connector waits infinitely.</p>
snapshot.select.statement.overrides		<p>Controls which table rows are included in snapshots. This property affects snapshots only. It does not affect events that the connector reads from the log. Specify a comma-separated list of fully-qualified table names in the form <i>schemaName.tableName</i>.</p> <p>For each table that you specify, also specify another configuration property: snapshot.select.statement.overrides.SCHEMA_NAME.TABLE_NAME. For example: snapshot.select.statement.overrides.customers.orders. Set this property to a SELECT statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this SELECT statement to retrieve data from that table.</p> <p>A possible use case for setting these properties is large, append-only tables. You can specify a SELECT statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.</p>
sanitize.field.names	<p>true if connector configuration sets the key.converter or value.converter property to the Avro converter.</p> <p>false if not.</p>	<p>Indicates whether field names are sanitized to adhere to Avro naming requirements.</p>

Property	Default	Description
provide.transaction.meta.data	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.

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 3.12. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic		The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.bootstrap.servers		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.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.

Property	Default	Description
database.history.store.only.monitored.tables.ddl <i>Deprecated and scheduled for removal in a future release; use database.history.store.only.captured.tables.ddl instead.</i>	false	<p>A Boolean value that specifies whether the connector should record all DDL statements</p> <p>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.</p> <p>The safe default is false.</p>
database.history.store.only.captured.tables.ddl	false	<p>A Boolean value that specifies whether the connector should record all DDL statements</p> <p>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.</p> <p>The safe default is false.</p>

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.foofoo=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=<database.server.name>**.

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	string[]	The list of tables that are monitored 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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

3.7.2. Monitoring Debezium Db2 connector record streaming

The MBean is `debezium.db2:type=connector-metrics,context=streaming,server=<database.server.name>`.

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 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	string[]	The list of tables that are monitored 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 incorporate 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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

3.7.3. Monitoring Debezium Db2 connector schema history

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

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.

Attributes	Type	Description
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 3.13. 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');
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.

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.

The MongoDB connector uses this same replication mechanism, 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.

As the MongoDB connector processes changes, it periodically records the position in the oplog where the event originated. When the MongoDB connector stops, it records the last oplog 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](#)

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”](#)

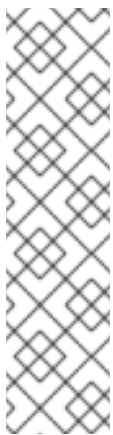
4.2.1. MongoDB topologies supported by Debezium connectors

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 position of each replica set.

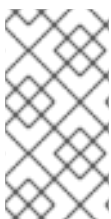
Assign a unique logical name to each MongoDB connector. The name should meaningfully describe the source MongoDB system. It's best to assign logical names that begin with an alphabetic or underscore character and that include only alphanumeric or underscore characters.

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 is performing a snapshot of any replica sets. The connector does log messages with the progress of the snapshot. For utmost control, run a separate cluster of Kafka Connect for each connector.

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 connects to the replica set's primary node and start 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

Under normal operating conditions, Kafka consumers read every message **exactly once**. However, if an error occurs, Kafka guarantees only that consumers 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. 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
      }
    ]
  }
}

```

The 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:

```

{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
    ...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "1462833718356672513",
    "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 4.1. 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 4.2. 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	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 <i>connector-name.database-name.collection-name</i> . Key . In this example: <ul style="list-style-type: none"> ● 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 4.3. 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\" }" }
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](#)

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": { ❶
  "type": "struct",
  "fields": [
    {
      "type": "string",
      "optional": true,
      "name": "io.debezium.data.Json", ❷
      "version": 1,
      "field": "after"
    },
    {
      "type": "string",
      "optional": true,
      "name": "io.debezium.data.Json",
      "version": 1,
      "field": "patch"
    },
    {
      "type": "string",
      "optional": true,
      "name": "io.debezium.data.Json",
      "version": 1,
      "field": "filter"
    },
    {
      "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": "1.5.4.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 4.4. 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.
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 <i>create</i> event is the only kind of event that contains an after field. An <i>update</i> event contains a filter field and a patch field. A <i>delete</i> 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 <code>_create_events</code> ; in other words, a <i>create</i> event is the only kind of event that contains an <i>after</i> field.

Item	Field name	Description
7	source	<p>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:</p> <ul style="list-style-type: none"> ● 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.
8	op	<p>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:</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (applies to only snapshots)
9	ts_ms	<p>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.</p> <p>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.</p>

update events

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": "1.5.4.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 6,
      "h": 1546547425148721999
    }
  }
}
```

Table 4.5. 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	<p>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.</p> <p>An <i>update</i> event value does not contain an after field.</p>
4	filter	<p>Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be updated.</p>
5	source	<p>Mandatory field that describes the source metadata for the event. This field contains the same information as a <i>create</i> 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:</p> <ul style="list-style-type: none"> ● 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.

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** value. 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": "1.5.4.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 6,
      "h": 1546547425148721999
    }
  }
}
```

Table 4.6. 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.
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	filter	Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be deleted.
4	source	<p>Mandatory field that describes the source metadata for the event. This field contains the same information as a <i>create</i> or <i>update</i> 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:</p> <ul style="list-style-type: none"> ● 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.

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 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.

4.5. DEPLOYMENT OF DEBEZIUM MONGODB CONNECTORS

To deploy a Debezium MongoDB connector, add the connector files to Kafka Connect, create a custom container to run the connector, and add the connector configuration to your container. Details are in the following topics:

- [Section 4.5.1, “Deploying Debezium MongoDB connectors”](#)
- [Section 4.5.2, “Description of Debezium Db2 connector configuration properties”](#)

4.5.1. Deploying Debezium MongoDB connectors

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. Download the Debezium [MongoDB connector archive](#).
 - b. Extract the Debezium MongoDB connector archive to create a directory structure for the connector plug-in, for example:

```

|
| ./my-plugins/
| |  └─ debezium-connector-mongodb
| |    └─ ...

```

- c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```
cat <<EOF >debezium-container-for-mongodb.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF
```

1 1 1 1 1 1 1 You can specify any file name that you want.

2 2 2 2 2 2 2 Replace **my-plugins** with the name of your plug-ins directory.

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

- d. 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**.

- e. 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
```

- f. 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
```

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

- g. 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. **fulfillment** 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
```

- 1** The name that is used to register the connector with Kafka Connect.
- 2** The name of the MongoDB connector class.
- 3** The host addresses to use to connect to the MongoDB replica set.
- 4** 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.
- 5** 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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```

- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...
... INFO Snapshot is using user 'debezium' ...
```

If the connector starts correctly without errors, it creates a topic for each collection from which the connector captures changes. For the CR in the preceding example, there would be a topic for the collection specified in the **collection.include.list** property. Downstream applications can subscribe to the topics that the connector creates.

- c. Verify that the connector created topics by running the following command:

```
oc get kafkatopics
```

For the complete list of the configuration properties that you can set for the Debezium MongoDB connector, see [MongoDB connector configuration properties](#).

Results

When the connector starts, it completes the following actions:

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

4.5.2. Description of Debezium Db2 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 4.7. 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).
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. Only alphanumeric characters and underscores should be used.
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 .
mongodb.ssl.enabled	false	Connector will use SSL to connect to MongoDB instances.

Property	Default	Description
mongodb.ssl.invalid.host.name.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	<i>empty string</i>	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	<i>empty string</i>	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	<i>empty string</i>	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 <i>databaseName.collectionName</i> . 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	<i>empty string</i>	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 <i>databaseName.collectionName</i> . Must not be used with collection.include.list .
snapshot.mode	initial	Specifies the criteria for running a snapshot upon startup of the connector. The default is initial , and specifies the connector reads a snapshot when either no offset is found or if the oplog 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.

Property	Default	Description
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	<i>empty string</i>	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 <i>databaseName.collectionName.fieldName.nestedFieldName</i> , where <i>databaseName</i> and <i>collectionName</i> may contain the wildcard (*) which matches any characters.
field.renames	<i>empty string</i>	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 <i>databaseName.collectionName.fieldName.nestedFieldName:newNestedFieldName</i> , where <i>databaseName</i> and <i>collectionName</i> 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 initial sync of the collections in a replica set. Defaults to 1.

Property	Default	Description
tombstones.on.delete	true	<p>Controls whether a <i>delete</i> event is followed by a tombstone event.</p> <p>true - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event.</p> <p>false - only a <i>delete</i> event is emitted.</p> <p>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.</p>
snapshot.delay.ms		<p>An interval in milliseconds that the connector should wait before taking a snapshot after starting up;</p> <p>Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.</p>
snapshot.fetch.size	0	<p>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.</p> <p>Defaults to 0, which indicates that the server chooses an appropriate fetch size.</p>

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 4.8. Required Debezium MongoDB connector advanced configuration properties

Property	Default	Description
max.queue.size	8192	<p>Positive integer value that specifies the maximum size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the oplog reader when, for example, writes to Kafka are slower or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified in the max.batch.size property.</p>

Property	Default	Description
<code>max.batch.size</code>	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.
<code>max.queue.size.in.bytes</code>	0	Long value for the maximum size in bytes of the blocking queue. The feature is disabled by default, it will be active if it's set with a positive long value.
<code>poll.interval.ms</code>	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.
<code>connect.backoff.initial.delay.ms</code>	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).
<code>connect.backoff.max.delay.ms</code>	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).
<code>connect.max.attempts</code>	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 <code>connect.backoff.initial.delay.ms</code> and <code>connect.backoff.max.delay.ms</code> results in just over 20 minutes of attempts before failing.
<code>mongodb.members.auto.discover</code>	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 <code>mongodb.hosts</code> 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 .

Property	Default	Description
heartbeat.interval.ms	0	<p>Controls how frequently heartbeat messages are sent.</p> <p>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 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.</p> <p>Set this parameter to 0 to not send heartbeat messages at all. Disabled by default.</p>
heartbeat.topics.prefix	__debezium- heartbeat	<p>Controls the naming of the topic to which heartbeat messages are sent.</p> <p>The topic is named according to the pattern <heartbeat.topics.prefix>.<server.name>.</p>
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 oplog operations 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.

Property	Default	Description
snapshot.collection.filter.overrides		<p>Controls which collection items are included in snapshot. This property affects snapshots only. Specify a comma-separated list of collection names in the form <i>databaseName.collectionName</i>.</p> <p>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.</p>
provide.transaction.metadata	false	<p>When set to true Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata.</p> <p>See Transaction Metadata for additional details.</p>
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.

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.name>`.

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	string[]	The list of tables that are monitored 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.

Attributes	Type	Description
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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

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.sql_server:type=connector-metrics,context=streaming,server=<mongodb.name>`.

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 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	string[]	The list of tables that are monitored 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 incorporate 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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

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

Reconnection attempt number	Delay before attempt, in seconds	Total delay before attempt, in minutes and seconds
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 process 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

IMPORTANT

This release of the Debezium MySQL connector includes a new default capturing implementation that is based on the common connector framework that is used by the other Debezium connectors. The revised capturing implementation 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](#).

If the connector generates errors or unexpected behavior while running with the new capturing implementation, you can revert to the earlier implementation by setting the following configuration option:

```
internal.implementation=legacy
```

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.

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 and may have been recorded before the tables' schemas were changed.

To handle this, 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 See [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 include all DDL statements applied to databases in the MySQL server. The connector emits these events to a Kafka topic named *serverName* where *serverName* is the name of the connector as specified by the **database.server.name** connector configuration property.

If you choose to use *schema change events*, ensure that you consume records from the schema change topic. The database history topic is for connector use only.



IMPORTANT

A global order for events emitted to the schema change topic is vital. Therefore, you must not partition the database history topic. This means that you must specify a partition count of **1** when creating the database history topic. When relying on auto topic creation, make sure that Kafka's **num.partitions** configuration option, which specifies the default number of partitions, is set to **1**.

Each record that the connector emits to the schema change topic contains a message key that includes the name of the connected database when the DDL statement was applied, for example:

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

```

"type": "struct",
"name": "io.debezium.connector.mysql.SchemaChangeKey",
"optional": false,
"fields": [
  {
    "field": "databaseName",
    "type": "string",
    "optional": false
  }
]
},
"payload": {
  "databaseName": "inventory"
}
}

```

The schema change event record value contains a structure that includes the DDL statements, the name of the database to which the statements were applied, and the position in the binlog where the statements appeared, for example:

```

{
  "schema": {
    "type": "struct",
    "name": "io.debezium.connector.mysql.SchemaChangeValue",
    "optional": false,
    "fields": [
      {
        "field": "databaseName",
        "type": "string",
        "optional": false
      },
      {
        "field": "ddl",
        "type": "string",
        "optional": false
      },
      {
        "field": "source",
        "type": "struct",
        "name": "io.debezium.connector.mysql.Source",
        "optional": false,
        "fields": [
          {
            "type": "string",
            "optional": true,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          }
        ]
      }
    ]
  }
}

```

```

    },
    {
      "type": "int64",
      "optional": false,
      "field": "ts_ms"
    },
    {
      "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": "boolean",
      "optional": true,
      "default": false,
      "field": "snapshot"
    },
    {
      "type": "int64",
      "optional": true,
      "field": "thread"
    },
    {
      "type": "string",
      "optional": true,
      "field": "db"
    },
    {
      "type": "string",
      "optional": true,
      "field": "table"
    },
    {
      "type": "string",
      "optional": true,
      "field": "query"
    }
  ]
}
],
},

```

```

"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": "1.5.4.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
  }
}
}

```

The **ddl** field might contain multiple DDL statements. Each statement applies to the database in the **databaseName** field. The statements appear in the order in which they were applied to the database. The **source** field is structured exactly as a standard data change event written to table-specific topics. This field is useful to correlate events on different topics.

```

....
"payload": {
  "databaseName": "inventory",
  "ddl": "CREATE TABLE products ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY
KEY,...)",
  "source" : {
    ...
  }
}
....

```

A client can submit multiple DDL statements to be applied 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.

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

Step	Action
1	Grabs a global read lock that blocks <i>writes</i> 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 <i>consistent snapshot</i> .
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 *initial 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 5.2. Workflow for performing an initial snapshot with table-level locks

Step	Action
1	Obtains table-level locks.

Step	Action
2	Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the <i>consistent snapshot</i> .
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.

5.1.5. Default names of Kafka topics that receive Debezium MySQL change event records

The default behavior is that a Debezium MySQL connector writes events for all **INSERT**, **UPDATE**, and **DELETE** operations in one table to one Kafka topic. The Kafka topic naming convention is as follows:

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
```

Transaction metadata

Debezium can generate events that represent transaction boundaries and that enrich data change event messages. 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": "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
    }
  ]
}
```

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": "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": { ❶
    ...
  },
  "payload": { ❷
    ...
  },
  "schema": { ❸
    ...
  },
  "payload": { ❹
    ...
  },
}

```

Table 5.3. Overview of change event basic content

Item	Field name	Description
1	schema	<p>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.</p> <p>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.</p>
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 an alphabetic character 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 an alphabetic character, 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 5.4. 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	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 <i>connector-name.database-name.table-name</i> . Key . In this example: <ul style="list-style-type: none"> • 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": "1.5.4.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 5.5. 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	<p>In the schema section, each name field specifies the schema for a field in the value's payload.</p> <p>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.</p> <p>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.</p>
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.

Item	Field name	Description
5	payload	<p>The value's actual data. This is the information that the change event is providing.</p> <p>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.</p>
6	op	<p>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:</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (applies to only snapshots)
7	ts_ms	<p>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.</p> <p>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.</p>
8	before	<p>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.</p>
9	after	<p>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.</p>

Item	Field name	Description
10	source	<p>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:</p> <ul style="list-style-type: none"> ● 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 <p>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.</p>

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": "1.5.4.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 5.6. 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 <i>update</i> 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	<p>Mandatory field that describes the source metadata for the event. The source field structure has the same fields as in <i>create</i> event, but some values are different, for example, the sample <i>update</i> event is from a different position in the binlog. The source metadata includes:</p> <ul style="list-style-type: none"> ● 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 <p>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.</p>
4	op	<p>Mandatory string that describes the type of operation. In an <i>update</i> event value, the op field value is u, signifying that this row changed because of an update.</p>
5	ts_ms	<p>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.</p> <p>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.</p>



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": "1.5.4.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 5.7. 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 <i>delete</i> 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 <i>delete</i> event value, the after field is null , signifying that the row no longer exists.
3	source	<p>Mandatory field that describes the source metadata for the event. In a <i>delete</i> event value, the source field structure is the same as for <i>create</i> and <i>update</i> events for the same table. Many source field values are also the same. In a <i>delete</i> event value, the ts_ms and pos field values, as well as other values, might have changed. But the source field in a <i>delete</i> event value provides the same metadata:</p> <ul style="list-style-type: none"> ● 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 <p>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.</p>
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	<p>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.</p> <p>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.</p>

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)

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 5.8. Descriptions of basic type mappings

MySQL type	Literal type	Semantic type
BOOLEAN, BOOL	BOOLEAN	<i>n/a</i>
BIT(1)	BOOLEAN	<i>n/a</i>

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 <i>little-endian</i> 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	<i>n/a</i>
SMALLINT[(M)]	INT16	<i>n/a</i>
MEDIUMINT[(M)]	INT32	<i>n/a</i>
INT, INTEGER[(M)]	INT32	<i>n/a</i>
BIGINT[(M)]	INT64	<i>n/a</i>
REAL[(M,D)]	FLOAT32	<i>n/a</i>
FLOAT[(M,D)]	FLOAT64	<i>n/a</i>
DOUBLE[(M,D)]	FLOAT64	<i>n/a</i>
CHAR(M)]	STRING	<i>n/a</i>
VARCHAR(M)]	STRING	<i>n/a</i>
BINARY(M)]	BYTES or STRING	<i>n/a</i> 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	<i>n/a</i> 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	<i>n/a</i> 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	<i>n/a</i>

MySQL type	Literal type	Semantic type
BLOB	BYTES or STRING	<i>n/a</i> 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.
TEXT	STRING	<i>n/a</i>
MEDIUMBLOB	BYTES or STRING	<i>n/a</i> 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	<i>n/a</i>
LONGBLOB	BYTES or STRING	<i>n/a</i> 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.
LONGTEXT	STRING	<i>n/a</i>
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
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 **serverTimezone** MySQL configuration option. For example, if the database's time zone (either globally or configured for the connector by means of the **serverTimezone** 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 5.9. 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.
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.

MySQL type	Literal type	Semantic type
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 5.10. 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 5.11. 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 5.12. Mappings when `decimal.handling.mode=double`

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	FLOAT64	<i>n/a</i>
DECIMAL[(M[,D])]	FLOAT64	<i>n/a</i>

`decimal.handling.mode=string`

Table 5.13. Mappings when `decimal.handling.mode=string`

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	STRING	<i>n/a</i>
DECIMAL[(M[,D])]	STRING	<i>n/a</i>

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)**.

Following is an example configuration:

```
converters=boolean
boolean.type=io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter
boolean.selector=db1.table1.*, db1.table2.column1
```

Spatial types

Currently, the Debezium MySQL connector supports the following spatial data types.

Table 5.14. Description of spatial type mappings

MySQL type	Literal type	Semantic type
------------	--------------	---------------

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: <ul style="list-style-type: none"> ● 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 timeouts 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.

- Finalize the user's permissions:

```
mysql> FLUSH PRIVILEGES;
```

Table 5.15. 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.
REPLICATION CLIENT	Enables the connector the use of the following statements: <ul style="list-style-type: none"> • SHOW MASTER STATUS • SHOW SLAVE STATUS • SHOW BINARY LOGS <p>The connector always requires this.</p>
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:

```
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_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:

```
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
```

Table 5.16. 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 5.17. Descriptions of GTID options

Option	Description
gtid_mode	Boolean that specifies whether GTID mode of the MySQL server is enabled or not. <ul style="list-style-type: none"> • 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. <ul style="list-style-type: none"> • ON = enabled • OFF = disabled

5.4.4. Configuring MySQL session timeouts 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 5.18. 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

To deploy a Debezium MySQL connector, you add the connector files to Kafka Connect, create a custom container to run the connector, and then add the connector configuration to your container. For details about deploying the Debezium MySQL connector, see the following topics:

- [Section 5.5.1, “Deploying Debezium MySQL connectors”](#)
- [Section 5.5.2, “Description of Debezium MySQL connector configuration properties”](#)

5.5.1. Deploying Debezium MySQL connectors

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. Download the Debezium [MySQL connector archive](#).
 - b. Extract the Debezium MySQL connector archive to create a directory structure for the connector plug-in, for example:

```
./my-plugins/
├── debezium-connector-mysql
└── ...
```

- c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```
cat <<EOF >debezium-container-for-mysql.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF
```

1 1 1 1 1 1 You can specify any file name that you want.

2 2 2 2 2 2 Replace **my-plugins** with the name of your plug-ins directory.

The command creates a Docker file with the name **debezium-container-for-mysql.yaml** in the current directory.

- d. 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**.

- e. 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
```

- f. 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
```

- 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.

- g. 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 5.19. 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.
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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```


- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...
... INFO Snapshot is using user 'debezium' ...
```

If the connector starts correctly without errors, it creates a topic for each table whose changes the connector is capturing. For the example CR, there would be a topic for the table specified in the **include.list** property. Downstream applications can subscribe to these topics.

- c. Verify that the connector created the topics by running the following command:

```
oc get kafkatopics
```

For the complete list of the configuration properties that you can set for the Debezium MySQL connector, see [MySQL connector configuration properties](#).

Results

When 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.

5.5.2. Description 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 5.20. Required Debezium MySQL connector configuration properties

Property	Default	Description
name		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		The name of the Java class for the connector. Always specify io.debezium.connector.mysql.MySqlConnector for the MySQL connector.

Property	Default	Description
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		IP address or host name of the MySQL database server.
database.port	3306	Integer port number of the MySQL database server.
database.user		Name of the MySQL user to use when connecting to the MySQL database server.
database.password		Password to use when connecting to the MySQL database server.
database.server.name		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 and underscores are allowed in this name.
database.server.id	<i>random</i>	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.
database.include.list	<i>empty string</i>	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 configuration property.
database.exclude.list	<i>empty string</i>	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.

Property	Default	Description
table.include.list	<i>empty string</i>	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 <i>databaseName.tableName</i> . 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	<i>empty string</i>	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 <i>databaseName.tableName</i> . Do not also specify the table.include.list connector configuration property.
column.exclude.list	<i>empty string</i>	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 <i>databaseName.tableName.columnName</i> .
column.include.list	<i>empty string</i>	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 <i>databaseName.tableName.columnName</i> .
column.truncate.to._length_.chars	<i>n/a</i>	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 <i>databaseName.tableName.columnName</i> .

Property	Default	Description
<code>column.mask.with._length_.chars</code>	<i>n/a</i>	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 <i>databaseName.tableName.columnName</i> .
<code>column.mask.hash.hashAlgorithm.with.salt.salt</code>	<i>n/a</i>	<p>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 <i>databaseName.tableName.columnName</i>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. 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.</p> <p>In the following example, CzQMA0cB5K is a randomly selected salt.</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>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.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p>

Property	Default	Description
column.propagate.source.type	n/a	<p>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:</p> <p>__Debezium.source.column.type</p> <p>__Debezium.source.column.length</p> <p>__Debezium.source.column.scale</p> <p>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:</p> <p><i>databaseName.tableName.columnName</i></p> <p><i>databaseName.schemaName.tableName.columnName</i></p>
datatype.propagate.source.type	n/a	<p>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:</p> <p>__debezium.source.column.type</p> <p>__debezium.source.column.length</p> <p>__debezium.source.column.scale</p> <p>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:</p> <p><i>databaseName.tableName.typeName</i></p> <p><i>databaseName.schemaName.tableName.typeName</i></p> <p>See how MySQL connectors map data types for the list of MySQL-specific data type names.</p>

Property	Default	Description
time.precision.mode	adaptive_time_microseconds	<p>Time, date, and timestamps can be represented with different kinds of precision, including:</p> <p>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.</p> <p>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.</p>
decimal.handling.mode	precise	<p>Specifies how the connector should handle values for DECIMAL and NUMERIC columns:</p> <p>precise (the default) represents them precisely using java.math.BigDecimal values represented in change events in a binary form.</p> <p>double represents them using double values, which may result in a loss of precision but is easier to use.</p> <p>string encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.</p>
bigint.unsigned.handling.mode	long	<p>Specifies how BIGINT UNSIGNED columns should be represented in change events. Possible settings are:</p> <p>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.</p> <p>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.</p>

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.query	false	<p>Boolean value that specifies whether the connector should include the original SQL query that generated the change event.</p> <p>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.</p> <p>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.</p>
event.deserialization.failure.handling.mode	fail	<p>Specifies how the connector should react to exceptions during deserialization of binlog events.</p> <p>fail propagates the exception, which indicates the problematic event and its binlog offset, and causes the connector to stop.</p> <p>warn logs the problematic event and its binlog offset and then skips the event.</p> <p>ignore passes over the problematic event and does not log anything.</p>

Property	Default	Description
inconsistent.schema.handling.mode	fail	<p>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.</p> <p>fail throws an exception that indicates the problematic event and its binlog offset, and causes the connector to stop.</p> <p>warn logs the problematic event and its binlog offset and skips the event.</p> <p>skip passes over the problematic event and does not log anything.</p>
max.queue.size	8192	<p>Positive integer value that specifies the maximum size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the binlog reader when, for example, writes to Kafka are slow or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified by the max.batch.size property.</p>
max.batch.size	2048	<p>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.</p>
max.queue.size.in.bytes	0	<p>Long value for the maximum size in bytes of the blocking queue. The feature is disabled by default, it will be active if it's set with a positive long value.</p>
poll.interval.ms	1000	<p>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.</p>
connect.timeout.ms	30000	<p>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.</p>

Property	Default	Description
gtid.source.includes		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		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	<p>Controls whether a <i>delete</i> event is followed by a tombstone event.</p> <p>true - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event.</p> <p>false - only a <i>delete</i> event is emitted.</p> <p>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.</p>

Property	Default	Description
message.key.columns	<i>n/a</i>	<p>A semicolon separated list of tables with regular expressions that match table column names. The connector maps values in matching columns to key fields in change event records that it sends to Kafka topics. This is useful when a table does not have a primary key, or when you want to order change event records in a Kafka topic according to a field that is not a primary key.</p> <p>Separate entries with semicolons. Insert a colon between the fully-qualified table name and its regular expression. The format (shown with spaces for clarity only) is:</p> <pre>database-name . table-name : regex ; ...</pre> <p>For example:</p> <p>dbA.table_a:regex_1;dbB.table_b:regex_2;dbC.table_c:regex_3</p> <p>If table_a has an id column, and regex_1 is ^i (matches any column that starts with i), the connector maps the value in the id column of table_a to a key field in change events that the connector sends to Kafka.</p>
binary.handling.mode	bytes	<p>Specifies how binary columns, for example, blob, binary, varbinary, should be represented in change events. Possible settings:</p> <p>bytes represents binary data as a byte array.</p> <p>base64 represents binary data as a base64-encoded String.</p> <p>hex represents binary data as a hex-encoded (base16) String.</p>

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 5.21. Descriptions of MySQL connector advanced configuration properties

Property	Default	Description
----------	---------	-------------

Property	Default	Description
<code>connect.keep.alive</code>	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.
<code>table.ignore.builtin</code>	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.
<code>database.ssl.mode</code>	disabled	<p>Specifies whether to use an encrypted connection. Possible settings are:</p> <p>disabled specifies the use of an unencrypted connection.</p> <p>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.</p> <p>required establishes an encrypted connection or fails if one cannot be made for any reason.</p> <p>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.</p> <p>verify_identity behaves like verify_ca but additionally verifies that the server certificate matches the host of the remote connection.</p>

Property	Default	Description
<code>snapshot.mode</code>	<code>initial</code>	<p>Specifies the criteria for running a snapshot when the connector starts. Possible settings are:</p> <p>initial - the connector runs a snapshot only when no offsets have been recorded for the logical server name.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>

Property	Default	Description
snapshot.locking.mode	minimal	<p>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:</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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 <i>only</i> 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.</p>
snapshot.include.collection.list	All tables specified in table.include.list	An optional, comma-separated list of regular expressions that match names of schemas specified in table.include.list for which you want to take the snapshot.

Property	Default	Description
snapshot.select.statement.overrides		<p>Controls which table rows are included in snapshots. This property affects snapshots only. It does not affect events captured from the binlog. Specify a comma-separated list of fully-qualified table names in the form <i>databaseName.tableName</i>.</p> <p>For each table that you specify, also specify another configuration property: snapshot.select.statement.overrides.DB_NAME.TABLE_NAME. For example, the name of the other configuration property might be: snapshot.select.statement.overrides.customers.orders. Set this property to a SELECT statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this SELECT statement to retrieve data from that table.</p> <p>A possible use case for setting these properties is large, append-only tables. You can specify a SELECT statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.</p>
min.row.count.to.stream.results	1000	<p>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.</p> <p>To skip all table size checks and always stream all results during a snapshot, set this property to 0.</p>

Property	Default	Description
heartbeat.interval.ms	0	<p>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</p> <p>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.</p>
heartbeat.topics.prefix	<code>__debezium-heartbeat</code>	<p>Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern:</p> <p><i>heartbeat.topics.prefix.server.name</i></p> <p>For example, if the database server name is fulfillment, the default topic name is __debezium-heartbeat.fulfillment.</p>
database.initial.statements		<p>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, (<code>;;</code>).</p> <p>The connector might establish JDBC connections at its own discretion, so this property is only for configuring session parameters. It is not for executing DML statements.</p>
snapshot.delay.ms		<p>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.</p>
snapshot.fetch.size		<p>During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.</p>

Property	Default	Description
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	<p>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.</p> <p>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.</p>
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 .
skipped.operations		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.
provide.transaction.meta data	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.

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 5.22. Connector database history configuration properties

Property	Default	Description
----------	---------	-------------

Property	Default	Description
database.history.kafka.topic		The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.boots trap.servers		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.recov ery.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.recov ery.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.unpar seable.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 <i>Deprecated and scheduled for removal in a future release; use database.history.store.only.captured.tables.ddl instead.</i>	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 .
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.foo=bar** 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.
- [Binlog 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=<database.server.name>`.

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	string[]	The list of tables that are monitored 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.

Attributes	Type	Description
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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

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

The MBean is **debezium.mysql:type=connector-metrics,context=streaming,server=<database.server.name>**.

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.

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.

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	string[]	The list of tables that are monitored 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 incorporate 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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

The Debezium MySQL connector also provides the following additional streaming metrics:

Table 5.23. 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=<database.server.name>`.

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 (DEVELOPER PREVIEW)

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.

Debezium ingests change events from Oracle by using the native LogMiner database package .



IMPORTANT

The Debezium Oracle connector is a Developer Preview feature. Developer Preview features provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. A Developer Preview feature is not supported with Red Hat production service-level agreements (SLAs) and it might not be functionally complete; therefore, Red Hat does not recommend implementing any Developer Preview features in production environments. If you need assistance with this feature, you can engage with the [Debezium community](#).

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, "Monitoring Debezium Oracle connector performance"](#)
- [Section 6.7, "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, and uses metadata.

Details are in 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"](#)

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 an **EXCLUSIVE 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.
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).

6.1.2. Default names of Kafka topics that receive Debezium Oracle change event records

The default behavior is that a Debezium Oracle connector writes events for all **INSERT**, **UPDATE**, and **DELETE** operations in one table to one Kafka topic. The Kafka topic naming convention is as follows:

serverName.schemaName.tableName

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
```

6.1.3. How Debezium Oracle connectors expose database schema changes

The Debezium Oracle connector stores the history of schema changes in a database history topic. This topic reflects an internal connector state and you should not use it directly. Applications that require notifications about schema changes should obtain the information from the public schema change topic. The connector writes schema change events to a Kafka topic named **<serverName>**, where **serverName** is the name of the connector 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. .

The message contains a logical representation of the table schema.

Example: Message emitted to the schema change topic

The following example shows a typical schema change message in JSON format:

```
{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "1.5.4.Final",
      "connector": "oracle",
      "name": "server1",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "ORCLPDB1",
      "schema": "DEBEZIUM",
      "table": "CUSTOMERS",
      "txId": null,
      "scn": "1513734",
      "commit_scn": "1513734",
      "lcr_position": null
    },
    "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 6.2. 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 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 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.

Item	Field name	Description
7	primaryKeyColumnNames	List of columns that compose the table's primary key.
8	columns	Metadata for each column in the changed table.

Messages that the connector sends to the schema change topic use a message key that is equal to 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.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.

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 number of events emitted by changes originating from the given 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
    }
  ]
}

```

The transaction events are written to the topic named `<database.server.name>.transaction`.

6.1.4.1. 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 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"
  }
}

```

6.2. 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**, **SYSTEM**, or connector 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** 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

Like the message key, the value of a change event message has a *schema* section and *payload* section. The payload section of every change event value produced by the Oracle connector has an *envelope* structure with the following fields:

op

A mandatory field that contains a string value describing the type of operation. Values for the Oracle connector are **c** for create (or insert), **u** for update, **d** for delete, and **r** for read (in the case of a snapshot).

before

An optional field that, if present, contains 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.

**WARNING**

Whether or not this field and its elements are available is highly dependent on the [Supplemental Logging](#) configuration applying to the table.

after

An optional field that if present contains the state of the row *after* the event occurred. The structure is described by the same **server1.INVENTORY.CUSTOMERS.Value** Kafka Connect schema used in **before**.

source

A mandatory field that contains a structure describing the source metadata for the event, which in the case of Oracle contains these fields: the Debezium version, the connector name, whether the event is part of an ongoing snapshot or not, the transaction id (not while snapshotting), the SCN of the change, and a timestamp representing the point in time when the record was changed in the source database (during snapshotting, this is the point in time of snapshotting).

TIP

The **commit_scn** field is optional and describes the SCN of the transaction commit that the change event participates within. This field is only present when using the LogMiner connection adapter.

ts_ms

An optional field that, if present, contains the time (using the system clock in the JVM running the Kafka Connect task) at which the connector processed the event.

And of course, the *schema* portion of the event message's value contains a schema that describes this envelope structure and the nested fields within it.

create events

Let's look at what a *create* event value might look like for our **customers** table:

```
{
  "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": "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": "1.5.4.Final",
      "name": "server1",
      "ts_ms": 1520085154000,
      "txId": "6.28.807",
      "scn": "2122185",
      "commit_scn": "2122185",
      "snapshot": false
    },
    "op": "c",
    "ts_ms": 1532592105975
  }
}

```

Examining the **schema** portion of the preceding event's *value*, we can see how the following schema are defined:

- The *envelope*
- The **source** structure (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. This means that when using the [Avro Converter](#), the resulting Avro schemas for *each table* 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 they describe. This is true, because the JSON representation must include the *schema* and the *payload* portions of the message. You can use the [Avro Converter](#) to significantly decrease the size of the messages that the connector writes to Kafka topics.

update events

The value of an *update* change event on this table has the same *schema* as the *create* event. The payload uses the same structure, but it holds different values. Here's an example:

```

{
  "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": "1.5.4.Final",
      "name": "server1",
      "ts_ms": 1520085811000,
      "txId": "6.9.809",
      "scn": "2125544",
      "commit_scn": "2125544",
      "snapshot": false
    },
    "op": "u",
    "ts_ms": 1532592713485
  }
}

```

Comparing the value of the *update* event to the *create* (insert) event, notice the following differences in the **payload** section:

- The **op** field value is now **u**, signifying that this row changed because of an update
- The **before** field now has the state of the row with the values before the database commit
- The **after** field now has the updated state of the row, and here we can see that the **EMAIL** value is now **anne@example.com**.
- The **source** field structure has the same fields as before, but the values are different since this event is from a different position in the redo log.
- The **ts_ms** shows the timestamp that Debezium processed this 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

So far we've seen samples of *create* and *update* events. Now, let's look at the value of a *delete* event for the same table. As is the case with *create* and *update* events, for a **delete** event, the **schema** portion of the value is exactly the same:

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "ID": 1004,
      "FIRST_NAME": "Anne",
      "LAST_NAME": "Kretchmar",
      "EMAIL": "anne@example.com"
    },
    "after": null,
    "source": {
      "version": "1.5.4.Final",
      "name": "server1",
      "ts_ms": 1520085153000,
      "txId": "6.28.807",
      "scn": "2122184",
      "commit_scn": "2122184",
      "snapshot": false
    },
    "op": "d",
    "ts_ms": 1532592105960
  }
}
```

If we look at the **payload** portion, we see a number of differences compared with the *create* or *update* event payloads:

- The **op** field value is now **d**, signifying that this row was deleted
- The **before** field now has the state of the row that was deleted with the database commit.
- The **after** field is null, signifying that the row no longer exists
- The **source** field structure has many of the same values as before, except the **ts_ms**, **scn** and **txId** fields have changed
- The **ts_ms** shows the timestamp that Debezium processed this event.

This event gives a consumer all kinds of information that it can use 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 listed above still works with log compaction, since Kafka can still remove all earlier messages with that 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](#).

6.3. HOW DEBEZIUM ORACLE CONNECTORS MAP DATA TYPES

To represent changes that occur in a table rows, the Debezium Oracle connector emits change events that are structured like the table in which the rows exists. The event contains a field for each column value. Column values are represented according to the Oracle data type of the column. The following sections describe how the connector maps oracle 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.

Details are in the following sections:

- [Character types](#)
- [Binary and Character LOB types](#)
- [Numeric types](#)
- [Boolean types](#)
- [Decimal types](#)
- [Temporal types](#)

Character types

The following table describes how the connector maps basic character types.

Table 6.3. 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
VARCHAR2[(M)]	STRING	n/a

Binary and Character LOB types

The following table describes how the connector maps binary and character LOB types.

Table 6.4. Mappings for Oracle binary and character LOB types

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
BLOB	n/a	<i>This data type is not supported.</i>
CLOB	n/a	<i>This data type is not supported.</i>
LONG	n/a	<i>This data type is not supported.</i>
LONG RAW	n/a	<i>This data type is not supported.</i>
NCLOB	n/a	<i>This data type is not supported.</i>
RAW	n/a	<i>This data type is not supported.</i>

Numeric types

The following table describes how the connector maps numeric types.

Table 6.5. 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).
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.
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

Oracle Data Type	Literal type (schema type)	Semantic type (schema name) and Notes
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.
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: <ul style="list-style-type: none"> ● 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).
NUMBER (P, S > 0)	BYTES	org.apache.kafka.connect.data.Decimal
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).
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.

Boolean types

Oracle does not natively have 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.

The operator can configure the out-of-the-box **NumberOneToBooleanConverter** custom converter that would either map all **NUMBER(1)** columns to a **BOOLEAN** or if the **selector** parameter is set, then a subset of columns could be enumerated using a comma-separated list of regular expressions.

Following is an example configuration:

```
converters=boolean
boolean.type=io.debezium.connector.oracle.converters.NumberOneToBooleanConverter
boolean.selector=.*MYTABLE.FLAG,.*IS_ARCHIVED
```

Decimal types

The setting of the Oracle 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 Kafka Connect **org.apache.kafka.connect.data.Decimal** logical type for all **DECIMAL** and **NUMERIC** columns. This is the default mode.

However, when the **decimal.handling.mode** property is set to **double**, the connector represents the values as Java double values with schema type **FLOAT64**.

You can also set the **decimal.handling.mode** configuration property to use the **string** option. When the property is set to **string**, the connector represents **DECIMAL** and **NUMERIC** values as their formatted string representation with schema type **STRING**.

Temporal types

Other than Oracle's **INTERVAL**, **TIMESTAMP WITH TIME ZONE** and **TIMESTAMP WITH LOCAL TIME ZONE** data types, 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 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 past 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.
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.
TIMESTAMP(0 - 3)	INT64	io.debezium.time.Timestamp Represents the number of milliseconds past epoch, and does not include timezone information.

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIMESTAMP, TIMESTAMP(4 - 6)	INT64	io.debezium.time.MicroTimestamp Represents the number of microseconds past epoch, and does not include timezone information.
TIMESTAMP(7 - 9)	INT64	io.debezium.time.NanoTimestamp Represents the number of nanoseconds past 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 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.
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.
TIMESTAMP(0 - 3)	INT64	org.apache.kafka.connect.data.Timestamp Represents the number of milliseconds since epoch, and does not include timezone information.

Oracle data type	Literal type (schema type)	Semantic type (schema name) and Notes
TIMESTAMP(4 - 6)	INT64	org.apache.kafka.connect.data.Timestamp Represents the number of milliseconds since epoch, and does not include timezone information.
TIMESTAMP(7 - 9)	INT64	org.apache.kafka.connect.data.Timestamp Represents the number of milliseconds since 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.

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 information about using Vagrant to set up Oracle in a virtual machine, see the [Debezium Vagrant Box for Oracle database](#) GitHub repository.

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

- [Section 6.4.1, "Preparing Oracle databases for use with Debezium"](#)
- [Section 6.4.2, "Creating an Oracle user for the Debezium Oracle connector"](#)

6.4.1. 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;
```

In addition, supplemental logging must be enabled for captured tables or the database in order for data changes to capture the *before* state of changed database rows. The following illustrates how to configure this on a specific table, which is the ideal choice to minimize the amount of information captured in the Oracle redo logs.

```
ALTER TABLE inventory.customers ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

Minimal supplemental logging must be enabled at the database level and can be configured as follows.

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

6.4.2. 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 does not capture database changes made by the **SYS**, **SYSTEM**, or the connector 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;
GRANT SET CONTAINER TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$DATABASE to c##dbzuser CONTAINER=ALL;
```

```

GRANT FLASHBACK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TRANSACTION TO c##dbzuser CONTAINER=ALL;
GRANT LOGMINING TO c##dbzuser CONTAINER=ALL;

GRANT CREATE TABLE TO c##dbzuser CONTAINER=ALL;
GRANT LOCK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT ALTER ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT CREATE SEQUENCE TO c##dbzuser CONTAINER=ALL;

GRANT EXECUTE ON DBMS_LOGMNR TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR_D TO c##dbzuser CONTAINER=ALL;

GRANT SELECT ON V_$LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOG_HISTORY TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_LOGS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_CONTENTS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_PARAMETERS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGFILE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVED_LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO c##dbzuser CONTAINER=ALL;

exit;

```

6.5. DEPLOYMENT OF DEBEZIUM ORACLE CONNECTORS

To deploy a Debezium Oracle connector, you add the connector files to Kafka Connect, create a custom container to run the connector, and then add connector configuration to your container. For details about deploying the Debezium Oracle connector, see the following topics:

- [Section 6.5.1, “Obtaining the Oracle JDBC driver”](#)
- [Section 6.5.2, “Deploying Debezium Oracle connectors”](#)
- [Section 6.5.3, “Descriptions of Debezium Oracle connector configuration properties”](#)

6.5.1. Obtaining the Oracle JDBC driver

The Debezium Oracle connector requires the Oracle JDBC driver (**ojdbc8.jar**) to connect to Oracle databases. Due to licensing requirements, the required driver file is not included in the Debezium Oracle connector archive. You must download the required driver file directly from Oracle and add it to your Kafka Connect environment. The following steps describe how to download the Oracle Instant Client and extract the driver.

Procedure

1. From a browser, download the [Oracle Instant Client package](#) for your operating system.
2. Extract the archive and then open the **instantclient_<version>** directory.
For example:

```

instantclient_21_1/
├── adrci

```



```

|— BASIC_LITE_LICENSE
|— BASIC_LITE_README
|— genezi
|— libclntshcore.so -> libclntshcore.so.21.1
|— libclntshcore.so.12.1 -> libclntshcore.so.21.1
...
|— ojdbc8.jar
|— ucp.jar
|— uidrvci
|— xstreams.jar

```

Copy the `ojdbc8.jar` file to the ``_kafka_home_/libs` directory.`

6.5.2. Deploying Debezium Oracle connectors

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.
- You have a copy of the Oracle JDBC driver. Due to licensing requirements, the Debezium Oracle connector does not include the required JDBC driver files. For more information, see [Obtaining the Oracle JDBC driver](#).

Procedure

1. Create the Debezium Oracle container for Kafka Connect:
 - a. Download the Debezium [Oracle connector archive](#).
 - b. Extract the Debezium Oracle connector archive to create a directory structure for the connector plug-in, for example:

```
./my-plugins/
├── debezium-connector-oracle
└── ...
```

- c. Create a Dockerfile that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```
cat <<EOF >debezium-container-for-oracle.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF
```

1 1 1 You can specify any file name that you want.

2 2 2 Replace **my-plugins** with the name of your plug-ins directory.

The command creates a Docker file with the name **debezium-container-for-oracle.yaml** in the current directory.

- d. 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**.

- e. 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
```

- f. 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
```

- 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

- g. 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 6.6. 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.
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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```

- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...
... INFO Snapshot is using user 'c##dbzuser' ...
```

If the connector starts correctly without errors, it creates a topic for each table whose changes the connector is capturing. Downstream applications can subscribe to these topics.

- c. Verify that the connector created the topics by running the following command:

```
oc get kafkatopics
```

For the complete list of the configuration properties that you can set for the Debezium Oracle connector, see [Oracle connector properties](#).

Results

When 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.

6.5.3. 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.

tasks.max	1	The maximum number of tasks that should be created 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, and underscores.

database.connection.adapter	logminer	<p>The adapter implementation that the connector uses when it streams database changes. You can set the following values:</p> <p>logminer(default) The connector uses the native Oracle LogMiner API.</p> <p>xstream The connector uses the Oracle XStreams API.</p>
snapshot.mode	<i>initial</i>	<p>Specifies the mode that the connector uses to take snapshots of a captured table. You can set the following values:</p> <p>initial The snapshot includes the structure and data of captured tables. Specify this value to populate topics with a complete representation of the data from the captured tables.</p> <p>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.</p> <p>After the snapshot is complete, the connector continues to read change events from the database's redo logs.</p>
snapshot.select.statement.overrides	No default	<p>Specifies the table rows to include in a snapshot.</p> <p>This property contains a comma-separated list of fully-qualified tables (<schema_name.table_name>). Select statements for the individual tables are specified in further configuration properties, one for each table, identified by the id snapshot.select.statement.overrides.<schema_name>.<table_name>. The value of those properties is the SELECT statement to use when retrieving data from the specific table during snapshotting. <i>A possible use case for large append-only tables is setting a specific point where to start (resume) snapshotting, in case a previous snapshotting was interrupted.</i></p> <p>Note: This setting affects snapshots only. It does not apply to events that the connector captures during log reading.</p>

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.
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 identifier is of the form <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: <schema_name>.<table_name> . 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 are of the form <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 names for columns are of the form <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.

<p><code>column.mask.hash.<hashAlgorithm>.with.salt.<salt></code></p>	<p>No default</p>	<p>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 <i>dbName.schemaName.tableName.columnName</i>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. 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.</p> <p>In the following example, CzQMA0cB5K is a randomly selected salt.</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>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.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p>
---	-------------------	--

decimal.handling.mode	precise	<p>Specifies how the connector should handle floating point values for NUMBER, DECIMAL and NUMERIC columns. You can set one of the following options:</p> <p>precise (default) Represents values precisely by using java.math.BigDecimal values represented in change events in a binary form.</p> <p>double Represents values by using double values. Using double values is easier, but can result in a loss of precision.</p> <p>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 Decimal types.</p>
event.processing.failure.handling.mode	fail	<p>Specifies how the connector should react to exceptions during processing of events. You can set one of the following options:</p> <p>fail Propagates the exception (indicating the offset of the problematic event), causing the connector to stop.</p> <p>warn Causes the problematic event to be skipped. The offset of the problematic event is then logged.</p> <p>skip Causes the problematic event to be skipped.</p>
max.queue.size	8192	<p>A positive integer value that specifies the maximum size of the blocking queue. Change events read from the database log are placed in the blocking queue before they are written to Kafka. This queue can provide backpressure to the binlog reader when, for example, writes to Kafka are slow, or if Kafka is not available. Events that appear in the queue are not included in the offsets that the connector records periodically. Always specify a value that is larger than the maximum batch size that specified for the max.batch.size property.</p>
max.batch.size	2048	<p>A positive integer value that specifies the maximum size of each batch of events to process during each iteration of this connector.</p>

max.queue.size.in.bytes	0 (disabled)	Long value for the maximum size in bytes of the blocking queue. To activate the feature, set the value to a positive long data type.
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	<p>Controls whether a <i>delete</i> event is followed by a tombstone event. The following values are possible:</p> <p>true</p> <p>For each delete operation, the connector emits a <i>delete</i> event and a subsequent tombstone event.</p> <p>false</p> <p>For each delete operation, the connector emits only a <i>delete</i> event.</p> <p>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.</p>
message.key.columns	No default	<p>A list of regular expressions, separated by semi-colons, that match the fully-qualified tables and columns to map a primary key. Each item (regular expression) must match the <fully-qualified table>:<a comma-separated list of columns> representing the custom key.</p> <p>Define fully-qualified tables by using the following format:</p> <p><dbName>.<schemaName>.<tableName>.</p>
column.truncate.to.length.chars	No default	<p>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:</p> <p><dbName>.<schemaName>.<tableName>.<columnName>.</p>

<p>column.mask.with.length.chars</p>	<p>No default</p>	<p>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: dbName.schemaName.tableName.columnName. The connector configuration can include multiple properties that specify different lengths.</p>
<p>column.propagate.source.type</p>	<p>No default</p>	<p>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 <i>tableName.columnName</i>, or <i>schemaName.tableName.columnName</i>.</p>
<p>datatype.propagate.source.type</p>	<p>No default</p>	<p>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 <i>tableName.typeName</i>, or <i>schemaName.tableName.typeName</i>. See the list of Oracle-specific data type names</p>

heartbeat.interval.ms	0	<p>Specifies, in milliseconds, how frequently the connector sends messages to a heartbeat topic.</p> <p>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 the connector 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.</p>
heartbeat.topics.prefix	__debezium-heartbeat	<p>Specifies the string that prefixes the name of the topic to which the connector sends heartbeat messages.</p> <p>The topic is named according to the pattern <heartbeat.topics.prefix>.<server.name>.</p>
snapshot.delay.ms	No default	<p>Specifies an interval in milliseconds that the connector waits after it starts before it takes a snapshot.</p> <p>Use this property to prevent snapshot interruptions when you start multiple connectors in a cluster, which might cause re-balancing of connectors.</p>
snapshot.fetch.size	2000	<p>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.</p>

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.meta data	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.
log.mining.strategy	redo_log_catalog	The mining strategy 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.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.

<code>log.mining.sleep.time.min.ms</code>	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.
<code>log.mining.sleep.time.max.ms</code>	3000	The maximum amount of time that the connector will sleep after reading data from redo/archive logs and before starting reading data again. Value is in milliseconds.
<code>log.mining.sleep.time.default.ms</code>	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.
<code>log.mining.sleep.time.increament.ms</code>	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.
<code>log.mining.view.fetch.size</code>	10000	The number of content records that the connector fetches from the LogMiner content view.
<code>log.mining.archive.log.hours</code>	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.
<code>log.mining.transaction.retention.hours</code>	0	<p>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.</p> <p>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.</p>
<code>rac.nodes</code>	No default	A comma-separated list of Oracle Real Application Clusters (RAC) node host names or addresses. This field is required to enable use with Oracle RAC.

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 6.7. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic		The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.boots trap.servers		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.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.unpars eable.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 <i>Deprecated and scheduled for removal in a future release; use database.history.store.only.captured.tables.ddl instead.</i>	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	<p>A Boolean value that specifies whether the connector should record all DDL statements</p> <p>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.</p> <p>The safe default is false.</p>

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.foofoo=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.

6.6. 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.

6.6.1. Debezium Oracle connector snapshot metrics

The MBean is `debezium.oracle:type=connector-metrics,context=snapshot,server=<database.server.name>`.

Attributes	Type	Description
LastEvent	<code>string</code>	The last snapshot event that the connector has read.
MillisecondsSinceLastEvent	<code>long</code>	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	<code>long</code>	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	<code>long</code>	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
MonitoredTables	<code>string[]</code>	The list of tables that are monitored by the connector.
QueueTotalCapacity	<code>int</code>	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	<code>int</code>	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.

Attributes	Type	Description
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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

6.6.2. Debezium Oracle connector streaming metrics

The MBean is `debezium.oracle:type=connector-metrics,context=streaming,server=<database.server.name>`.

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 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	string[]	The list of tables that are monitored 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 incorporate 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.

Attributes	Type	Description
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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

The Debezium Oracle connector also provides the following additional streaming metrics:

Table 6.8. 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.
ComittedScn	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.
MaximumMinedLogCount	long	The maximum number of logs specified for any LogMiner session.

Attributes	Type	Description
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.
MaxMiningSessionStartTimeInMilliseconds	long	The longest duration in milliseconds to start a LogMiner session.

Attributes	Type	Description
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 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 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 the system change number has been checked for advancement and remains unchanged. This is an indicator that long-running transaction(s) are ongoing and preventing the connector from flushing the latest processed system change number to the connector's offsets. Under optimal operations, this should always be or remain close to 0 .

6.6.3. Debezium Oracle connector schema history metrics

The MBean is `debezium.oracle:type=connector-metrics,context=schema-history,server=<database.server.name>`.

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.

Attributes	Type	Description
LastRecoveredChange	string	The string representation of the last change recovered from the history store.
LastAppliedChange	string	The string representation of the last applied change.

6.7. 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.

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.

LogMiner adapter does not capture changes made by SYS or SYSTEM

Oracle uses the **SYS** and **SYSTEM** accounts for lots of internal changes and therefore the connector automatically filters changes made by these users when fetching changes from LogMiner. Never use the **SYS** or **SYSTEM** user accounts for changes to be emitted by the Debezium Oracle connector.

CHAPTER 7. DEBEZIUM CONNECTOR FOR POSTGRESQL

Debezium's PostgreSQL connector captures row-level changes in the schemas of a PostgreSQL database. PostgreSQL versions 10, 11, 12 and 13 are supported.

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. Obtain an **ACCESS SHARE MODE** lock on each of the tables being tracked to ensure that no structural changes can occur to any of the tables while the snapshot is taking place. These locks do not prevent table **INSERT**, **UPDATE** and **DELETE** operations from taking place during the snapshot.
*This step is omitted when **snapshot.mode** is set to **exported**, which allows the connector to perform a lock-free snapshot.*
3. Read the current position in the server’s transaction log.
4. Scan the database tables and schemas, generate a **READ** event for each row and write that event to the appropriate table-specific Kafka topic.
5. Commit the transaction.
6. 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 6 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 3. 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.

**WARNING**

It is strongly recommended that you configure a PostgreSQL connector to set **snapshot.mode** to **exported**. The **initial**, **initial only** and **always** modes can lose a few events while a connector switches from performing the snapshot to streaming change event records when a database is under heavy load. This is a known issue and the affected snapshot modes will be reworked to use **exported** mode internally ([DBZ-2337](#)).

Table 7.1. Settings for `snapshot.mode` connector configuration property

Setting	Description
always	<p>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:</p> <ul style="list-style-type: none"> • 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	<p>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.</p>
initial_only	<p>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.</p>
exported	<p>The connector performs a database snapshot based on the point in time when the replication slot was created. This mode is an excellent way to perform a snapshot in a lock-free way.</p>

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).

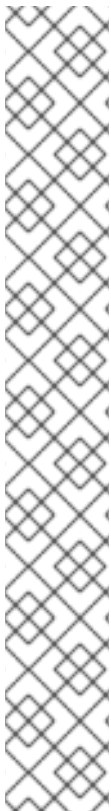
Whenever the server commits a transaction, a separate server process invokes a callback function from

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

The PostgreSQL connector writes events for all insert, update, and delete operations on a single table to a single Kafka topic. By default, the Kafka topic name is *serverName.schemaName.tableName* where:

- *serverName* is the logical name of the connector as specified with the **database.server.name** connector configuration property.
- *schemaName* is the name of the database schema where the operation occurred.
- *tableName* is the name of the database table in which the operation 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**

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. 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
    }
  ]
}
```

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 7.2. Overview of change event basic content

Item	Field name	Description
1	schema	<p>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.</p> <p>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.</p>
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 7.3. 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 <i>connector-name.database-name.table-name.Key</i> . In this example: <ul style="list-style-type: none"> • 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": { ❶
    "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", ❷
      "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": "1.5.4.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 7.4. 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	<p>In the schema section, each name field specifies the schema for a field in the value's payload.</p> <p>PostgreSQL_server.inventory.customers.Value is the schema for the payload's before and after fields. This schema is specific to the customers table.</p> <p>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.</p>
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	<p>The value's actual data. This is the information that the change event is providing.</p> <p>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.</p>
6	before	<p>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.</p> <div style="display: flex; align-items: flex-start; margin-top: 10px;">  <div> <p>NOTE</p> <p>Whether or not this field is available is dependent on the REPLICA IDENTITY setting for each table.</p> </div> </div>
7	after	<p>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.</p>
8	source	<p>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:</p> <ul style="list-style-type: none"> ● 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	<p>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:</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (applies to only snapshots)
10	ts_ms	<p>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.</p> <p>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.</p>

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": "1.5.4.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 7.5. 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 <i>update</i> 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 <i>create</i> event, but some values are different. The source metadata includes: <ul style="list-style-type: none"> • 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 <i>update</i> 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 <i>update</i> event value, the op field value is u , signifying that this row changed because of an update.

Item	Field name	Description
5	ts_ms	<p>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.</p> <p>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.</p>



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": "1.5.4.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 7.6. Descriptions of *delete* event value fields

Item	Field name	Description
1	before	<p>Optional field that specifies the state of the row before the event occurred. In a <i>delete</i> event value, the before field contains the values that were in the row before it was deleted with the database commit.</p> <p>In this example, the before field contains only the primary key column because the table's REPLICA IDENTITY setting is DEFAULT.</p>
2	after	<p>Optional field that specifies the state of the row after the event occurred. In a <i>delete</i> event value, the after field is null, signifying that the row no longer exists.</p>
3	source	<p>Mandatory field that describes the source metadata for the event. In a <i>delete</i> event value, the source field structure is the same as for <i>create</i> and <i>update</i> events for the same table. Many source field values are also the same. In a <i>delete</i> event value, the ts_ms and lsn field values, as well as other values, might have changed. But the source field in a <i>delete</i> event value provides the same metadata:</p> <ul style="list-style-type: none"> ● 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 <i>delete</i> 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	<p>Mandatory string that describes the type of operation. The op field value is d, signifying that this row was deleted.</p>

Item	Field name	Description
5	ts_ms	<p>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.</p> <p>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.</p>

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": "1.5.4.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 7.7. Descriptions of *truncate* event value fields

Item	Field name	Description
1	source	<p>Mandatory field that describes the source metadata for the event. In a <i>truncate</i> event value, the source field structure is the same as for <i>create</i>, <i>update</i>, and <i>delete</i> events for the same table, provides this metadata:</p> <ul style="list-style-type: none"> • 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 <i>delete</i> 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	<p>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.</p> <p>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.</p>

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.

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.

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 7.8. 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.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
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
CHAR[(M)]	STRING	n/a
VARCHAR[(M)]	STRING	n/a
CHARACTER[(M)]	STRING	n/a
CHARACTER VARYING[(M)]	STRING	n/a
TIMESTAMPZ, 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.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
INTERVAL [P]	STRING	<p>io.debezium.time.Interval (when interval.handling.mode is set to string)</p> <p>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.</p>
BYTEA	BYTES or STRING	<p>n/a</p> <p>Either the raw bytes (the default), a base64-encoded string, or a hex-encoded string, based on the connector's binary handling mode setting.</p>
JSON, JSONB	STRING	<p>io.debezium.data.Json</p> <p>Contains the string representation of a JSON document, array, or scalar.</p>
XML	STRING	<p>io.debezium.data.Xml</p> <p>Contains the string representation of an XML document.</p>
UUID	STRING	<p>io.debezium.data.Uuid</p> <p>Contains the string representation of a PostgreSQL UUID value.</p>
POINT	STRUCT	<p>io.debezium.data.geometry.Point</p> <p>Contains a structure with two FLOAT64 fields, (x,y). Each field represents the coordinates of a geometric point.</p>
LTREE	STRING	<p>io.debezium.data.Ltree</p> <p>Contains the string representation of a PostgreSQL LTREE value.</p>
CITEXT	STRING	n/a
INET	STRING	n/a
INT4RANGE	STRING	<p>n/a</p> <p>Range of integer.</p>

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
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.
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 7.9. 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.
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 7.10. 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.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
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.
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 7.11. 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** and **NUMERIC** columns. This is the default mode.

Table 7.12. 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.

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 7.13. Mappings of decimal types when there are no scale constraints

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.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
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** and **NUMERIC** values as Java double values and encodes them as shown in the following table.

Table 7.14. 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	

The last possible setting for the **decimal.handling.mode** configuration property is **string**. In this case, the connector represents **DECIMAL** and **NUMERIC** values as their formatted string representation, and encodes them as shown in the following table.

Table 7.15. 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	

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

When the **hstore.handling.mode** connector configuration 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 7.16. Mappings for **HSTORE** data type

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
HSTORE	STRING	io.debezium.data.Json Example: output representation using the JSON converter is <code>{"key" : "val"}</code>
HSTORE	MAP	n/a Example: output representation using the JSON converter is <code>{"key" : "val"}</code>

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 7.17. Mappings for network address types

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

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name) and Notes
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 7.18. 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: <ul style="list-style-type: none"> ● 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 .
GEOGRAPHY (spherical)	STRUCT	io.debezium.data.geometry.Geography Contains a structure with two fields: <ul style="list-style-type: none"> ● 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, **toasted.value.placeholder**.

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 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 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 1 1 1 1 1 1 1 Instructs the server to allow replication for **<youruser>** locally, that is, on the server machine.

2 2 2 2 2 2 2 2 Instructs the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV4**.

3 3 3 3 3 3 3 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 POSTGRES SQL CONNECTORS

To deploy a Debezium PostgreSQL connector, add the connector files to Kafka Connect, create a custom container to run the connector, and add connector configuration to your container. Details are in the following topics:

- [Section 7.6.1, "Deploying Debezium PostgreSQL connectors"](#)
- [Section 7.6.2, "Description of Debezium PostgreSQL connector configuration properties"](#)

7.6.1. Deploying Debezium PostgreSQL connectors

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. Download the Debezium [PostgreSQL connector archive](#).
 - b. Extract the Debezium PostgreSQL connector archive to create a directory structure for the connector plug-in, for example:

```
./my-plugins/
├── debezium-connector-postgresql
└── ...
```

- c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```
cat <<EOF >debezium-container-for-postgresql.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF
```

- 1** You can specify any file name that you want.
- 2** Replace **my-plugins** with the name of your plug-ins directory.

The command creates a Docker file with the name **debezium-container-for-postgresql.yaml** in the current directory.

- d. 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**.

- e. 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
```

- f. 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
```

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.

- g. 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 ❶
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.postgresql.PostgresConnector
  tasksMax: 1 ❷
  config: ❸
    database.hostname: 192.168.99.100 ❹
    database.port: 5432
    database.user: debezium
    database.password: dbz
    database.dbname: sampledb
    database.server.name: fulfillment ❺
    schema.include.list: public ❻
    plugin.name: pgoutput ❼
```

- ❶ The name of the connector.
- ❷ 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.
- ❸ The connector's configuration.
- ❹ The name of the database host that is running the PostgreSQL server. In this example, the database host name is **192.168.99.100**.
- ❺ 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.
- ❻ 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](#).
- ❼ The name of the PostgreSQL [logical decoding plug-in](#) installed on the PostgreSQL server.

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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```

- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...
... INFO Snapshot is using user 'debezium' ...
```

If the connector starts correctly without errors, it creates a topic for each table whose changes the connector is capturing. For the example CR, there would be a topic for each table in the **public** schema. Downstream applications can subscribe to these topics.

- c. Verify that the connector created the topics by running the following command:

```
oc get kafkatopics
```

Results

When 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.

7.6.2. Description 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 7.19. Required 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.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 .
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.

Property	Default	Description
publication.name	dbz_publication	<p>The name of the PostgreSQL publication created for streaming changes when using pgoutput.</p> <p>This publication is created at start-up if it does not already exist and it includes <i>all tables</i>. 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.</p> <p>If the publication already exists, either for all tables or configured with a subset of tables, Debezium uses the publication as it is defined.</p>
database.hostname		IP address or hostname of the PostgreSQL database server.
database.port	5432	Integer port number of the PostgreSQL database server.
database.user		Name of the PostgreSQL database user for connecting to the PostgreSQL database server.
database.password		Password to use when connecting to the PostgreSQL database server.
database.dbname		The name of the PostgreSQL database from which to stream the changes.
database.server.name		Logical name that identifies and provides a namespace for the particular PostgreSQL database server or cluster in which Debezium is capturing changes. Only alphanumeric characters and underscores should 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.

Property	Default	Description
schema.include.list		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		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		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 <i>schemaName.tableName</i> . 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		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 its changes captured. Each identifier is of the form <i>schemaName.tableName</i> . Do not also set the table.include.list property.
column.include.list		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 <i>schemaName.tableName.columnName</i> . Do not also set the column.exclude.list property.

Property	Default	Description
column.exclude.list		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 <i>schemaName.tableName.columnName</i> . Do not also set the column.include.list property.
time.precision.mode	adaptive	<p>Time, date, and timestamps can be represented with different kinds of precision:</p> <p>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.</p> <p>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.</p> <p>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.</p>
decimal.handling.mode	precise	<p>Specifies how the connector should handle values for DECIMAL and NUMERIC columns:</p> <p>precise represents values by using java.math.BigDecimal to represent values in binary form in change events.</p> <p>double represents values by using double values, which might result in a loss of precision but which is easier to use.</p> <p>string encodes values as formatted strings, which are easy to consume but semantic information about the real type is lost. See Decimal types.</p>

Property	Default	Description
hstore.handling.mode	map	<p>Specifies how the connector should handle values for hstore columns:</p> <p>map represents values by using MAP.</p> <p>json represents values by using json string. This setting encodes values as formatted strings such as {"key" : "val"}. See PostgreSQL HSTORE type.</p>
interval.handling.mode	numeric	<p>Specifies how the connector should handle values for interval columns:</p> <p>numeric represents intervals using approximate number of microseconds.</p> <p>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.</p>
database.sslmode	disable	<p>Whether to use an encrypted connection to the PostgreSQL server. Options include:</p> <p>disable uses an unencrypted connection.</p> <p>require uses a secure (encrypted) connection, and fails if one cannot be established.</p> <p>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.</p> <p>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.</p>
database.sslcert		The path to the file that contains the SSL certificate for the client. See the PostgreSQL documentation for more information.
database.sslkey		The path to the file that contains the SSL private key of the client. See the PostgreSQL documentation for more information.

Property	Default	Description
<code>database.sslpassword</code>		The password to access the client private key from the file specified by <code>database.sslkey</code> . See the PostgreSQL documentation for more information.
<code>database.sslrootcert</code>		The path to the file that contains the root certificate(s) against which the server is validated. See the PostgreSQL documentation for more information.
<code>database.tcpKeepAlive</code>	<code>true</code>	Enable TCP keep-alive probe to verify that the database connection is still alive. See the PostgreSQL documentation for more information.
<code>tombstones.on.delete</code>	<code>true</code>	Controls whether a <i>delete</i> event is followed by a tombstone event. true - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event. false - only a <i>delete</i> 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.
<code>column.truncate.to._length_chars</code>	<i>n/a</i>	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 <i>schemaName.tableName.columnName</i> . In change event records, values in these columns are truncated if they are longer than the number of characters specified by <i>length</i> 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
<code>column.mask.with._length_chars</code>	<i>n/a</i>	<p>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 <i>schemaName.tableName.columnName</i>. In change event values, the values in the specified table columns are replaced with <i>length</i> 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.</p>
<code>column.mask.hash.hashAlgorithm.with.salt.salt</code>	<i>n/a</i>	<p>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 <i>schemaName.tableName.columnName</i>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. 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.</p> <p>In the following example, CzQMA0cB5K is a randomly selected salt.</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>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.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p>

Property	Default	Description
column.propagate.source.type	n/a	<p>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 <i>databaseName.tableName.columnName</i>, or <i>databaseName.schemaName.tableName.columnName</i>.</p> <p>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:</p> <p>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</p> <p>This property is useful for properly sizing corresponding columns in sink databases.</p>
datatype.propagate.source.type	n/a	<p>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 <i>databaseName.tableName.typeName</i>, or <i>databaseName.schemaName.tableName.typeName</i>.</p> <p>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:</p> <p>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</p> <p>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.</p> <p>See the list of PostgreSQL-specific data type names.</p>

Property	Default	Description
message.key.columns	<i>empty string</i>	<p>A semicolon separated list of tables with regular expressions that match table column names. The connector maps values in matching columns to key fields in change event records that it sends to Kafka topics. This is useful when a table does not have a primary key, or when you want to order change event records in a Kafka topic according to a field that is not a primary key.</p> <p>Separate entries with semicolons. Insert a colon between the fully-qualified table name and its regular expression. The format is:</p> <pre><i>schema-name.table-name:_regexp_;</i>...</pre> <p>For example,</p> <p>schemaA.table_a:regex_1;schemaB.table_b:regex_2;schemaC.table_c:regex_3</p> <p>If table_a has an id column, and regex_1 is ^i (matches any column that starts with i), the connector maps the value in table_a's id column to a key field in change events that the connector sends to Kafka.</p>

Property	Default	Description
publication.autocreate.mode	<i>all_tables</i>	<p>Applies only when streaming changes by using the pgoutput plug-in. The setting determines how creation of a publication should work. Possible settings are:</p> <p>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;</p> <p>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.</p> <p>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>.</p>
binary.handling.mode	bytes	<p>Specifies how binary (bytea) columns should be represented in change events:</p> <p>bytes represents binary data as byte array.</p> <p>base64 represents binary data as base64-encoded strings.</p> <p>hex represents binary data as hex-encoded (base16) strings.</p>

Property	Default	Description
truncate.handling.mode	bytes	<p>Specifies how whether TRUNCATE events should be propagated or not (only available when using the pgoutput plug-in with Postgres 11 or later):</p> <p>skip causes those event to be omitted (the default).</p> <p>include causes those events to be included.</p> <p>For information about the structure of <i>truncate</i> events and about their ordering semantics, see truncate events.</p>

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 7.20. Advanced connector configuration properties

Property	Default	Description
----------	---------	-------------

Property	Default	Description
snapshot.mode	initial	<p>Specifies the criteria for performing a snapshot when the connector starts:</p> <p>initial - The connector performs a snapshot only when no offsets have been recorded for the logical server name.</p> <p>always - The connector performs a snapshot each time the connector starts.</p> <p>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.</p> <p>initial_only - The connector performs an initial snapshot and then stops, without processing any subsequent changes.</p> <p>exported - The connector performs a snapshot based on the point in time when the replication slot was created. This is an excellent way to perform the snapshot in a lock-free way.</p> <p>The reference table for snapshot mode settings has more details.</p>
snapshot.include.collecton.list	All tables specified in table.include.list	An optional, comma-separated list of regular expressions that match names of schemas specified in table.include.list for which you want to take the snapshot when the snapshot.mode is not never
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		<p>Controls which table rows are included in snapshots. This property affects snapshots only. It does not affect events that are generated by the logical decoding plug-in. Specify a comma-separated list of fully-qualified table names in the form <i>databaseName.tableName</i>.</p> <p>For each table that you specify, also specify another configuration property: snapshot.select.statement.overrides.DB_NAME.TABLE_NAME, for example: snapshot.select.statement.overrides.customers.orders. Set this property to a SELECT statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this SELECT statement to retrieve data from that table.</p> <p>A possible use case for setting these properties is large, append-only tables. You can specify a SELECT statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.</p>
event.processing.failure.handling.mode	fail	<p>Specifies how the connector should react to exceptions during processing of events:</p> <p>fail propagates the exception, indicates the offset of the problematic event, and causes the connector to stop.</p> <p>warn logs the offset of the problematic event, skips that event, and continues processing.</p> <p>skip skips the problematic event and continues processing.</p>
max.queue.size	20240	<p>Positive integer value for the maximum size of the blocking queue. The connector places change events received from streaming replication in the blocking queue before writing them to Kafka. This queue can provide backpressure when, for example, writing records to Kafka is slower than it should be or Kafka is not available.</p>

Property	Default	Description
max.batch.size	10240	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
max.queue.size.in.bytes	0	Long value for the maximum size in bytes of the blocking queue. The feature is disabled by default, it will be active if it's set with a positive long value.
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
<p>include.unknown.datatypes</p>	<p>false</p>	<p>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.</p> <p>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.</p> <div data-bbox="922 689 1029 1348" style="border: 1px solid black; padding: 5px; width: fit-content;">  </div> <p>NOTE</p> <p>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.</p>

Property	Default	Description
database.initial.statement S		<p>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, ;;</p> <p>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.</p> <p>The connector does not execute these statements when it creates a connection for reading the transaction log.</p>

Property	Default	Description
heartbeat.interval.ms	0	<p>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</p> <p>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.</p> <p>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.</p>
heartbeat.topics.prefix	<code>__debezium-heartbeat</code>	<p>Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern:</p> <p><code><heartbeat.topics.prefix>.<server.name></code></p> <p>For example, if the database server name is fulfillment, the default topic name is __debezium-heartbeat.fulfillment.</p>

Property	Default	Description
heartbeat.action.query		<p>Specifies a query that the connector executes on the source database when the connector sends a heartbeat message.</p> <p>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:</p> <pre>INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')</pre> <p>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.</p>
schema.refresh.mode	columns_diff	<p>Specify the conditions that trigger a refresh of the in-memory schema for a table.</p> <p>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.</p> <p>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.</p> <p>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.</p>

Property	Default	Description
snapshot.delay.ms		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		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 .
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.

Property	Default	Description
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.
retriable.restart.connector.wait.ms	10000 (10 seconds)	The number of milliseconds to wait before restarting a connector after a retriable error occurs.

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=<database.server.name>`.

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	string[]	The list of tables that are monitored 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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

7.7.2. Monitoring Debezium PostgreSQL connector record streaming

The MBean is **debezium.postgres:type=connector-metrics,context=streaming,server=<database.server.name>**.

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 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	string[]	The list of tables that are monitored by the connector.

Attributes	Type	Description
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 incorporate 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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

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 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 8.2.5, “Descriptions of Debezium SQL Server connector data change events”](#)
- [Section 8.2.7, “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 8.2.3, “Default names of Kafka topics that receive Debezium SQL Server change event records”](#)
- [Section 8.2.4, “How the Debezium SQL Server connector uses the schema change topic”](#)
- [Section 8.2.5, “Descriptions of Debezium SQL Server connector data change events”](#)
- [Section 8.2.6, “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.

8.2.2. 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. Default names of Kafka topics that receive Debezium SQL Server change event records

The SQL Server connector writes events for all **INSERT**, **UPDATE**, and **DELETE** operations for a specific table to a single Kafka topic. By default, the Kafka topic name takes the form *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 SQL Server installation. The server has an **inventory** database with the schema name **dbo**, and the database contains tables with the names **products**, **products_on_hand**, **customers**, and **orders**. The connector would stream records to the following Kafka topics:

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

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 [Routing Debezium event records to topics that you specify](#).

8.2.4. 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 schema changes in a database history topic. This topic reflects an internal connector state and you should not use it directly. Application that require notifications about schema changes, should obtain the information from the public schema change topic. The connector writes all of these events to a Kafka topic named **<serverName>**, where **serverName** is the name of the connector that is specified in the `database.server.name` configuration property.



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](#).

A message to the schema change topic contains a logical representation of the table schema, for example:

```
{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "1.5.4.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", ❶
"schemaName": "dbo",
"ddl": null, ❷
"tableChanges": [ ❸
{
  "type": "CREATE", ❹
  "id": "\"testDB\".\"dbo\".\"customers\"", ❺
  "table": { ❻
    "defaultCharsetName": null,
    "primaryKeyColumnNames": [ ❼
      "id"
    ],
    "columns": [ ❽
      {
        "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,

```


Item	Field name	Description
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"
  }
}
```

8.2.5. 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 8.2. Overview of change event basic content

Item	Field name	Description
1	schema	<p>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.</p> <p>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.</p>
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 8.2.5.1, "About keys in Debezium SQL Server change events"](#)
- [Section 8.2.5.2, "About values in Debezium SQL Server change events"](#)

8.2.5.1. 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 8.3. 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	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 <i>connector-name.database-schema-name.table-name.Key</i> . In this example: <ul style="list-style-type: none"> ● 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 .

8.2.5.2. 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": "1.5.4.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 8.4. 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	<p>In the schema section, each name field specifies the schema for a field in the value's payload.</p> <p>server1.dbo.customers.Value is the schema for the payload's before and after fields. This schema is specific to the customers table.</p> <p>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.</p>
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	<p>The value's actual data. This is the information that the change event is providing.</p> <p>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.</p>

Item	Field name	Description
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.
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: <ul style="list-style-type: none"> ● 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: <ul style="list-style-type: none"> ● 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. <p>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.</p>

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": "1.5.4.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 8.5. 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 <i>update</i> 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	<p>Mandatory field that describes the source metadata for the event. The source field structure has the same fields as in <i>create</i> event, but some values are different, for example, the sample <i>update</i> event has a different offset. The source metadata includes:</p> <ul style="list-style-type: none"> • 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 <p>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:</p> <ul style="list-style-type: none"> • <i>update</i> 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 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 events. A <i>delete</i> event for the removal of the record with the old primary key value and a <i>create</i> 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	<p>Mandatory string that describes the type of operation. In an <i>update</i> event value, the op field value is u, signifying that this row changed because of an update.</p>
5	ts_ms	<p>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.</p> <p>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.</p>



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": "1.5.4.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 8.6. 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 <i>delete</i> 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 <i>delete</i> event value, the after field is null , signifying that the row no longer exists.

Item	Field name	Description
3	source	<p>Mandatory field that describes the source metadata for the event. In a <i>delete</i> event value, the source field structure is the same as for <i>create</i> and <i>update</i> events for the same table. Many source field values are also the same. In a <i>delete</i> event value, the ts_ms and pos field values, as well as other values, might have changed. But the source field in a <i>delete</i> event value provides the same metadata:</p> <ul style="list-style-type: none"> • 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	<p>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.</p> <p>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.</p>

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.

8.2.6. 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.

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
    }
  ]
}
```

The transaction events are written to the topic named `<database.server.name>.transaction`.

8.2.6.1. 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"
  }
}
```

8.2.7. 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.

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 8.7. 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
NTEXT	STRING	n/a
XML	STRING	io.debezium.data.Xml Contains the string representation of an XML document

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
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.
TIME(7)	INT64	io.debezium.time.NanoTime Represents the number of nanoseconds past midnight, and does not include timezone information.

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
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
DATE	INT32	org.apache.kafka.connect.data.Date Represents the number of days since the epoch.

SQL Server data type	Literal type (schema type)	Semantic type (schema name) and Notes
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 8.8. 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 8.9. Mappings when `decimal.handling.mode=double`

SQL Server type	Literal type	Semantic type
NUMERIC[(M[,D])]	FLOAT64	<i>n/a</i>
DECIMAL[(M[,D])]	FLOAT64	<i>n/a</i>
SMALLMONEY[(M[,D])]	FLOAT64	<i>n/a</i>
MONEY[(M[,D])]	FLOAT64	<i>n/a</i>

decimal.handling.mode=string

Table 8.10. Mappings when `decimal.handling.mode=string`

SQL Server type	Literal type	Semantic type
NUMERIC[(M[,D])]	STRING	<i>n/a</i>
DECIMAL[(M[,D])]	STRING	<i>n/a</i>
SMALLMONEY[(M[,D])]	STRING	<i>n/a</i>

SQL Server type	Literal type	Semantic type
MONEY[(M[,D])]	STRING	<i>n/a</i>

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 has not been tested with SQL Server on Azure.

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

To deploy a Debezium SQL Server connector, you add the connector files to Kafka Connect, create a custom container to run the connector, and then add connector configuration to your container. For details about deploying the Debezium SQL Server connector, see the following topics:

- [Section 8.4.1, “Deploying Debezium SQL Server connectors”](#)
- [Section 8.4.2, “Descriptions of Debezium SQL Server connector configuration properties”](#)

8.4.1. Deploying Debezium SQL Server connectors

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. Download the Debezium [SQL Server connector archive](#).
 - b. Extract the Debezium SQL Server connector archive to create a directory structure for the connector plug-in, for example:

```
./my-plugins/
├── debezium-connector-sqlserver
└── ...
```

- c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0** as the base image. For example, from a terminal window, enter the following, replacing **my-plugins** with the name of your plug-ins directory:

```
cat <<EOF >debezium-container-for-sqlserver.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:1.8.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ 2
USER 1001
EOF
```

1 1 1 1 1 1 You can specify any file name that you want.

2 2 2 2 2 1 2 Replace **my-plugins** with the name of your plug-ins directory.

The command creates a Docker file with the name **debezium-container-for-sqlserver.yaml** in the current directory.

- d. 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**.

- e. 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
```

- f. 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
```

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

- g. 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: fullfulment 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 8.11. 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.
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.

4. Verify that the connector was created and has started:
 - a. Display the Kafka Connect log output to verify that the connector was created and has started to capture changes in the specified database:

```
oc logs $(oc get pods -o name -l strimzi.io/cluster=my-connect-cluster)
```

- b. Review the log output to verify that Debezium performs the initial snapshot. The log displays output that is similar to the following messages:

```
... INFO Starting snapshot for ...  
... INFO Snapshot is using user 'debezium' ...
```

If the connector starts correctly without errors, it creates a topic for each table whose changes the connector is capturing. For the example CR, there would be a topic for the table specified in the **include.list** property. Downstream applications can subscribe to these topics.

- c. Verify that the connector created the topics by running the following command:

```
oc get kafkatopics
```

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.

8.4.2. 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		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.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		IP address or hostname of the SQL Server database server.
database.port	1433	Integer port number of the SQL Server database server.
database.user		Username to use when connecting to the SQL Server database server.
database.password		Password to use when connecting to the SQL Server database server.
database.dbname		The name of the SQL Server database from which to stream the changes
database.server.name		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 a prefix for all Kafka topic names emanating from this connector. Only alphanumeric characters and underscores should be used.

Property	Default	Description
table.include.list		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 <i>schemaName.tableName</i> . By default, the connector captures all non-system tables for the designated schemas. Must not be used with table.exclude.list .
table.exclude.list		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 <i>schemaName.tableName</i> . Must not be used with table.include.list .
column.include.list	<i>empty string</i>	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 <i>schemaName.tableName.columnName</i> . 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.
column.exclude.list	<i>empty string</i>	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 <i>schemaName.tableName.columnName</i> . 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.

Property	Default	Description
<p>column.mask.hash.hashAlgorithm.with.salt.salt</p>	<p>n/a</p>	<p>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 <i>schemaName.tableName.columnName</i>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. 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.</p> <p>In the following example, CzQMA0cB5K is a randomly selected salt.</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>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.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p>
<p>time.precision.mode</p>	<p>adaptive</p>	<p>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.</p>

Property	Default	Description
decimal.handling.mode	precise	<p>Specifies how the connector should handle values for DECIMAL and NUMERIC columns:</p> <p>precise (the default) represents them precisely using java.math.BigDecimal values represented in change events in a binary form.</p> <p>double represents them using double values, which may result in a loss of precision but is easier to use.</p> <p>string encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.</p>
include.schema.changes	true	<p>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.</p>
tombstones.on.delete	true	<p>Controls whether a <i>delete</i> event is followed by a tombstone event.</p> <p>true - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event.</p> <p>false - only a <i>delete</i> event is emitted.</p> <p>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.</p>

Property	Default	Description
<code>column.truncate.to._length_chars</code>	<i>n/a</i>	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 <i>schemaName.tableName.columnName</i> .
<code>column.mask.with._length_chars</code>	<i>n/a</i>	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 <i>schemaName.tableName.columnName</i> .
<code>column.propagate.source.type</code>	<i>n/a</i>	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 <i>schemaName.tableName.columnName</i> .

Property	Default	Description
datatype.propagate.source.type+	<i>n/a</i>	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 <code>__debezium.source.column.type</code> , <code>__debezium.source.column.length</code> and <code>__debezium.source.column.scale</code> 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 <code>schemaName.tableName.typeName</code> . See SQL Server data types for the list of SQL Server-specific data type names.
message.key.columns	<i>n/a</i>	A semi-colon list of regular expressions that match fully-qualified tables and columns to map a primary key. Each item (regular expression) must match the fully-qualified <fully-qualified table>:<a comma-separated list of columns> representing the custom key. Fully-qualified tables could be defined as <code>schemaName.tableName</code> .
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

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
snapshot.mode	<i>initial</i>	<p>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:</p> <ul style="list-style-type: none"> ● 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	<p>An optional, comma-separated list of regular expressions that match names of fully-qualified table names (<db-name>.<schema-name>.<name>) included in table.include.list for which you want to take the snapshot.</p>

Property	Default	Description
snapshot.isolation.mode	<i>repeatable_read</i>	<p>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:</p> <ul style="list-style-type: none"> ● 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). <p>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.</p> <p>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).</p>
event.processing.failure.handling.mode	fail	<p>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.</p> <p>warn will cause the problematic event to be skipped and the offset of the problematic event to be logged.</p> <p>skip will cause the problematic event to be skipped.</p>

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 size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the CDC table reader when, for example, writes to Kafka are slower or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified in the max.batch.size property.
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.
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> .

Property	Default	Description
snapshot.delay.ms		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		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.
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.
snapshot.select.statement.overrides		Controls which rows from tables are included in snapshot. This property contains a comma-separated list of fully-qualified tables (<i>SCHEMA_NAME.TABLE_NAME</i>). Select statements for the individual tables are specified in further configuration properties, one for each table, identified by the id snapshot.select.statement.overrides.[SCHEMA_NAME].[TABLE_NAME] . The value of those properties is the SELECT statement to use when retrieving data from the specific table during snapshotting. <i>A possible use case for large append-only tables is setting a specific point where to start (resume) snapshotting, in case a previous snapshotting was interrupted.</i> Note: This setting has impact on snapshots only. Events captured during log reading are not affected by it.

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.
database.server.timezone		<p>Timezone of the server.</p> <p>This property defines the timezone of the transaction timestamp (ts_ms) that is retrieved from the server (which is actually not zoned). By default, the value is unset. Set a value for the property only when running on SQL Server 2014 or older, and the database server and the JVM running the Debezium connector use different timezones.</p> <p>When unset, default behavior is to use the timezone of the VM running the Debezium connector. In this case, when running on on SQL Server 2014 or older and using different timezones on server and the connector, incorrect ts_ms values may be produced. Possible values include "Z", "UTC", offset values like "+02:00", short zone ids like "CET", and long zone ids like "Europe/Paris".</p>
provide.transaction.metadata	false	<p>When set to true Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata.</p> <p>See Transaction Metadata for additional details.</p>
retriable.restart.connector.wait.ms	10000 (10 seconds)	The number of milli-seconds to wait before restarting a connector after a retriable error occurs.

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 8.12. Connector database history configuration properties

Property	Default	Description
database.history.kafka.topic		The full name of the Kafka topic where the connector stores the database schema history.
database.history.kafka.bootstrap.servers		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.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 <i>Deprecated and scheduled for removal in a future release; use database.history.store.only.captured.tables.ddl instead.</i>	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 .
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.foofoo=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:

- [Section 8.5.1, “Running an offline update after a schema change”](#) . In offline schema updates, capture tables are updated after you stop the Debezium connector.
- [Section 8.5.2, “Running an online update after a schema change”](#) . In online schema updates, capture tables are updated 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

- 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, Q 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]
[jio.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]
[jio.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]
[jio.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,context=snapshot,server=<database.server.name>`.

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	string[]	The list of tables that are monitored 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. It will be enabled if max.queue.size.in.bytes is passed with a positive long value.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

8.6.2. Debezium SQL Server connector streaming metrics

The MBean is `debezium.sql_server:type=connector-metrics,context=streaming,server=<database.server.name>`.

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 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	string[]	The list of tables that are monitored 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 incorporate 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.
CurrentQueueSizeInBytes	long	The current data of records in the queue in bytes.

8.6.3. Debezium SQL Server connector schema history metrics

The MBean is `debezium.sql_server:type=connector-metrics,context=schema-history,server=<database.server.name>`.

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.

Attributes	Type	Description
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 [Zookeeper](#) and [Kafka](#) 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 OPENSIFT

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 [Setting up metrics and dashboards for AMQ Streams](#), 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. THE 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 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 2 The **stdout** appender writes log messages to the console (as opposed to a file).
- 3 3 The **stdout** appender uses a pattern matching algorithm to format the log messages.
- 4 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. However, you can change this configuration in the following ways:

- [Changing the logging level](#)
- [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

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. Adding Debezium 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'
[jio.debezium.connector.mysql.SnapshotReader]
2017-02-07 20:49:37,696 INFO  MySQL|dbserver1|snapshot Snapshot is using user
'debezium' with these MySQL grants: [jio.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'@%' [jio.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 OPENSIFT

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.

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/kafka.strimzi.io/v1beta2
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/kafka.strimzi.io/v1beta2
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
1	<p>topic.creation.default.replication.factor defines the replication factor for topics created by the default group.</p> <p>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.</p>

Item	Description
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/kafka.strimzi.io/v1beta2
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: 777600000

    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.applg- . 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.applg-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/kafka.strimzi.io/v1beta2
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/kafka.strimzi.io/v1beta2
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 [Red Hat Integration - Service Registry](#) documentation.

11.2.1. About the Service Registry

[Red Hat Integration - Service 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 Service Registry with Debezium, add Service Registry converters and their dependencies to the Kafka Connect container image that you are using for running a Debezium connector.

**NOTE**

The Service 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 Service 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 Integration - Service Registry instance by following the instructions in [Getting Started with Service Registry](#).
2. Install the Avro converter by downloading the [Service Registry Kafka Connectors](#) 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/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
```

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 Service Registry. See [Getting Started with Service Registry](#), which provides instructions for:
 - Installing Service 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 Integration download site](#) and download the Service 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-28-rhel8:1.8.0** 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-28-rhel8:1.8.0
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/kafka.strimzi.io/v1beta2
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&zeroDateBehavior=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",
  "source" : "/debezium/postgresql/test_server",
  "specversion" : "1.0",
  "type" : "io.debezium.postgresql.datachangeevent",
  "time" : "2020-01-13T13:55:39.738Z",
  "datacontenttype" : "application/json",
  "iodebeziumop" : "r",
  "iodebeziumversion" : "1.5.4.Final",
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578923739738",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "565",
  "iodebeziumlsn" : "29274832",
  "iodebeziumxmin" : null,
  "iodebeziumtxid" : "565",
  "iodebeziumtxtotalorder" : "1",
```

```

"iodebeziومتxdatacollectionorder": "1",
"data" : {
  "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 **iodebeziومت** 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 **iodebeziومتx** 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",
  "dataschema" : "http://my-registry/schemas/ids/1",
  "iodebeziومتop" : "r",
  "iodebeziومتversion" : "1.5.4.Final",
  "iodebeziومتconnector" : "postgresql",

```

```

"iodebeziumname" : "test_server",
"iodebeziumtsms" : "1578924258597",
"iodebeziumsnapshot" : "true",
"iodebeziumdb" : "postgres",
"iodebeziumschema" : "s1",
"iodebeziumtable" : "a",
"iodebeziumtxld" : "578",
"iodebeziumlsn" : "33227720",
"iodebeziumxmin" : null,
"iodebeziumtxid" : "578",
"iodebeziumtxtotalorder" : "1",
"iodebeziumtxdatacollectionorder" : "1",
"data" : "AAAAAAEAAgICAg=="
}

```

- 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",
"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
<code>serializer.type</code>	<code>json</code>	The encoding type to use for the CloudEvents envelope structure. The value can be <code>json</code> or <code>avro</code> .
<code>data.serializer.type</code>	<code>json</code>	The encoding type to use for the <code>data</code> attribute. The value can be <code>json</code> or <code>avro</code> .
<code>json. ...</code>	N/A	Any configuration options to be passed through to the underlying converter when using JSON. The <code>json.</code> prefix is removed.
<code>avro. ...</code>	N/A	Any configuration options to be passed through to the underlying converter when using Avro. The <code>avro.</code> prefix is removed. For example, for Avro <code>data</code> , you would specify the <code>avro.schema.registry.url</code> option.

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.

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 **RecordsTombstone** 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
 - RecordsIsTombstone
 - TopicNameMatches
- For example:

```
predicates.IsOutboxTable.type=org.apache.kafka.connect.predicates.TopicNameMatch
```

- 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.predicates.TopicNameMatch
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 `RecordsTombstone` 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.

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
<code>topic.regex</code>		Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
<code>topic.replacement</code>		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 <code>topic.regex</code> . To refer to a group, specify <code>\$1</code> , <code>\$2</code> , and so on.
<code>key.enforce.uniqueness</code>	<code>true</code>	<p>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.</p> <p>Specify <code>false</code> 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 <code>key.enforce.uniqueness=false</code> because unique keys are guaranteed in partitioned PostgreSQL tables.</p>

Option	Default	Description
key.field.name	<code>__dbz__physicalTableIdentifier</code>	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, and add to your Debezium connector plug-in directories, along any other JAR files used by the language implementation. For example, for Groovy 3, you can download its JSR 223 implementation from <https://groovy-lang.org/>. The JSR 223 implementation for GraalVM JavaScript is available at <https://github.com/graalvm/graaljs>.

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-1.5.4.Final.tar.gz**. To use the content-based routing SMT with a Debezium connector plug-in, you must explicitly add the SMT artifact to your Kafka Connect environment.



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.

Procedure

1. From a browser, open the [Red Hat Integration download site](#), and download the Debezium scripting SMT archive (**debezium-scripting-1.5.4.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.

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

Name	Description	Type
headers	<p>A Java map of message headers. The key field is the header name. The headers variable exposes the following properties:</p> <ul style="list-style-type: none"> • 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 disposes 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 fields by applying the [ExtractNewDocumentState](#) SMT.

You could also take the approach of using a JSON parser within the expression. 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	<p>Specifies how the transformation handles null (tombstone) messages. You can specify one of the following options:</p> <p>keep (Default) Pass the messages through.</p> <p>drop Remove the messages completely.</p> <p>evaluate Apply the condition logic to the messages.</p>
---------------------------	-------------	--

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, and add to your Debezium connector plug-in directories, along any other JAR files used by the language implementation. For example, for Groovy 3, you can download its JSR 223 implementation from <https://groovy-lang.org/>. The JSR223 implementation for GraalVM JavaScript is available at <https://github.com/graalvm/graaljs>.

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-1.5.4.Final.tar.gz**. To use the filter SMT with a Debezium connector plug-in, you must explicitly add the SMT artifact to your Kafka Connect

environment.



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.

Procedure

1. From a browser, open the [Red Hat Integration download site](#) , and download the Debezium scripting SMT archive (**debezium-scripting-1.5.4.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**.

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
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: <ul style="list-style-type: none"> ● 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 disposes 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 fields by applying the [ExtractNewDocumentState](#) SMT.

You could also take the approach of using a JSON parser within the expression. 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.

Similarly, 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 for 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.

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.header** 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
<code>drop.tombstones</code>	<code>true</code>	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 <code>drop.tombstones=false</code> .

Option	Default	Description
<code>delete.handling.mode</code>	<code>drop</code>	<p>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 <code>delete.handling.mode</code> to none or rewrite.</p> <p>Specify none to keep the change event record in the stream. The record contains only <code>"value": "null"</code>.</p> <p>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 <code>__deleted: true</code> to the value. This is another way to indicate that the record has been deleted.</p> <p>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.</p>

Option	Default	Description
route.by.field		<p>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.</p> <p>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.</p> <p>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.</p>
add.fields.prefix	__ (double-underscore)	Set this optional string to prefix a field.

Option	Default	Description
add.fields		<p>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.</p> <p>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.</p> <p>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.</p> <p>If you specify a field that is not in the change event record, the SMT still adds the field to the record's value.</p>
add.headers.prefix	__ (double-underscore)	Set this optional string to prefix a header.

Option	Default	Description
add.headers		<p>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.</p> <p>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.</p> <p>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.</p> <p>If you specify a field that is not in the change event record, the SMT does not add the field to the header.</p>

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.



IMPORTANT

The Debezium outbox event router 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](#).

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 does **not** support the MongoDB connector.

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 12.6.5, "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, "Options for configuring outbox event router transformation"](#)

12.6.1. Example of a Debezium outbox message

To learn about how to configure the Debezium outbox event router SMT, consider 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",
    "aggreatetype": "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": "1.5.4.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. You can customize the behavior of the SMT by modifying the values of the default 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
aggreatetype	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
--------	--------

Column	Effect
id	<p>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.</p> <p>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.</p>
aggregatetype	<p>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 <code>#{routedByValue}</code> variable in the route.topic.replacement SMT option.</p> <p>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 <code>outbox.event.#{routedByValue}</code>. 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 <code>outbox.event.customers</code> topic. The connector emits the second record to the <code>outbox.event.orders</code> topic.</p> <p>To obtain this value from a different outbox table column, set the route.by.field SMT option in the connector configuration.</p>
aggregateid	<p>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.</p> <p>To obtain the event key from a different outbox table column, set the table.field.event.key SMT option in the connector configuration.</p>
type	A user-defined value that helps categorize or organize events.
payload	<p>The representation of the event itself. The default structure is JSON. The content in this field becomes one of these:</p> <ul style="list-style-type: none"> • Part of the outbox message payload. • If other metadata, including eventType is delivered as headers, the payload becomes the message itself without encapsulation in an envelope. <p>To obtain the event payload from a different outbox table column, set the table.field.event.payload SMT option in the connector configuration.</p>

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
```

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 different 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**. 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=type:header:eventType
```

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=type:envelope:eventType
```

12.6.7. 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.field.event.id	id	Table	Specifies the outbox table column that contains the unique event ID.
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.

Option	Default	Group	Description
table.fields.additional.placement		Table, Envelope	<p>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:</p> <p>id:header,my-field:envelope</p> <p>To specify an alias for the column, specify a trio with the alias as the third value, for example:</p> <p>id:header,my-field:envelope:my-alias</p> <p>The second value is the placement and it must always be header or envelope.</p> <p>Configuration examples are in emitting additional fields in Debezium outbox messages.</p>
table.field.event.schema.version		Table, Schema	<p>When set, this value is used as the schema version as described in the Kafka Connect Schema Javadoc.</p>
route.by.field	aggregatetype	Router	<p>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.</p>
route.topic.regex	(? <routeByValue >.*)	Router	<p>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.</p> <p>The default behavior is that the SMT replaces the default #{routeByValue} variable in the setting of the route.topic.replacement SMT option with the setting of the route.by.field outbox SMT option.</p>

Option	Default	Group	Description
route.topic.replacement	outbox.event .\${routedByValue}	Router	<p>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.</p> <p>To change the topic name, you can:</p> <ul style="list-style-type: none"> ● 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	<p>Indicates whether an empty or null payload causes the connector to emit a tombstone event.</p>
debezium.op.invalid.behavior	warn	Debezium	<p>Determines the behavior of the SMT when there is an UPDATE operation on the outbox table. Possible settings are:</p> <ul style="list-style-type: none"> ● 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. <p>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.</p>

Revised on 2021-09-24 16:01:40 UTC

