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. DEBEZIUM CONNECTOR FOR MYSQL ........................................... 11
  2.1. HOW DEBEZIUM MYSQL CONNECTORS WORK ............................... 11
      2.1.1. MySQL topologies supported by Debezium connectors .......... 11
      2.1.2. How Debezium MySQL connectors handle database schema changes 12
      2.1.3. How Debezium MySQL connectors expose database schema changes 13
      2.1.4. How Debezium MySQL connectors perform database snapshots 16
      2.1.5. Default names of Kafka topics that receive Debezium MySQL change event records 18
  2.2. DESCRIPTIONS OF DEBEZIUM MYSQL CONNECTOR DATA CHANGE EVENTS 18
      2.2.1. About keys in Debezium MySQL change events .................. 20
      2.2.2. About values in Debezium MySQL change events ............... 21
  2.3. HOW DEBEZIUM MYSQL CONNECTORS MAP DATA TYPES ................. 32
  2.4. SETTING UP MYSQL TO RUN A DEBEZIUM CONNECTOR .................. 38
      2.4.1. Creating a MySQL user for a Debezium connector ............... 38
      2.4.2. Enabling the MySQL binlog for Debezium ......................... 39
      2.4.3. Enabling MySQL Global Transaction Identifiers for Debezium 40
      2.4.4. Configuring MySQL session timesouts for Debezium .......... 41
      2.4.5. Enabling query log events for Debezium MySQL connectors 42
  2.5. DEPLOYMENT OF DEBEZIUM MYSQL CONNECTORS ............................ 43
      2.5.1. Deploying Debezium MySQL connectors ............................... 43
      2.5.2. Description of Debezium MySQL connector configuration properties 47
  2.6. MONITORING DEBEZIUM MYSQL Connector PERFORMANCE ............. 65
      2.6.1. Monitoring Debezium during snapshots of MySQL databases .... 65
      2.6.2. Monitoring Debezium MySQL connector binlog reading .......... 67
      2.6.3. Monitoring Debezium MySQL connector schema history ........... 69
  2.7. HOW DEBEZIUM MYSQL CONNECTORS HANDLE FAULTS AND PROBLEMS 70

CHAPTER 3. DEBEZIUM CONNECTOR FOR POSTGRESQL ................................. 73
  3.1. OVERVIEW OF DEBEZIUM POSTGRESQL CONNECTOR ..................... 73
  3.2. HOW DEBEZIUM POSTGRESQL CONNECTORS WORK .......................... 74
      3.2.1. Security for PostgreSQL connector .................................. 75
      3.2.2. How Debezium PostgreSQL connectors perform database snapshots 75
      3.2.3. How Debezium PostgreSQL connectors stream change event records 76
      3.2.4. Default names of Kafka topics that receive Debezium PostgreSQL change event records 77
      3.2.5. Metadata in Debezium PostgreSQL change event records ......... 78
      3.2.6. Debezium PostgreSQL connector-generated events that represent transaction boundaries 79
  3.3. DESCRIPTIONS OF DEBEZIUM POSTGRESQL CONNECTOR DATA CHANGE EVENTS 80
      3.3.1. About keys in Debezium PostgreSQL change events .............. 82
      3.3.2. About values in Debezium PostgreSQL change events .......... 84
  3.4. HOW DEBEZIUM POSTGRESQL CONNECTORS MAP DATA TYPES ........ 95
  3.5. SETTING UP POSTGRESQL TO RUN A DEBEZIUM CONNECTOR ............. 106
      3.5.1. Configuring a replication slot for the Debezium pgoutput plug-in 106
      3.5.2. Setting up PostgreSQL permissions for the Debezium connector 106
      3.5.3. Setting privileges to enable Debezium to create PostgreSQL publications 107
      3.5.4. Configuring PostgreSQL to allow replication with the Debezium connector host 108
      3.5.5. Configuring PostgreSQL to manage Debezium WAL disk space consumption 108
3.6. DEPLOYMENT OF DEBEZIUM POSTGRESQL CONNECTORS
3.6.1. Deploying Debezium PostgreSQL connectors 109
3.6.2. Description of Debezium PostgreSQL connector configuration properties 113
3.7. MONITORING DEBEZIUM POSTGRESQL CONNECTOR PERFORMANCE
3.7.1. Monitoring Debezium during snapshots of PostgreSQL databases 133
3.7.2. Monitoring Debezium PostgreSQL connector record streaming 135
3.8. HOW DEBEZIUM POSTGRESQL CONNECTORS HANDLE FAULTS AND PROBLEMS 136

CHAPTER 4. DEBEZIUM CONNECTOR FOR MONGODB ........................................................................ 140
4.1. OVERVIEW 140
4.2. SETTING UP MONGODB 141
4.3. SUPPORTED MONGODB TOPOLOGIES 141
  4.3.1. MongoDB replica set 141
  4.3.2. MongoDB sharded cluster 141
  4.3.3. MongoDB standalone server 142
4.4. HOW THE MONGODB CONNECTOR WORKS 142
  4.4.1. Logical connector name 142
  4.4.2. Performing a snapshot 142
  4.4.3. Streaming changes 143
  4.4.4. Topics names 144
  4.4.5. Partitions 144
  4.4.6. Data change events 144
    4.4.6.1. Change event keys 146
    4.4.6.2. Change event values 148
    4.4.6.3. create events 148
    4.4.6.4. update events 152
    4.4.6.5. delete events 154
  4.4.7. Transaction Metadata 156
    4.4.7.1. Transaction boundaries 156
    4.4.7.2. Data events enrichment 157
4.5. DEPLOYING THE MONGODB CONNECTOR 158
  4.5.1. Example configuration 158
  4.5.2. Adding connector configuration 159
  4.5.3. Monitoring 161
    4.5.3.1. Snapshot Metrics 161
    4.5.3.2. Streaming Metrics 163
  4.5.4. Connector properties 164
4.6. MONGODB CONNECTOR COMMON ISSUES 171
  4.6.1. Configuration and startup errors 172
  4.6.2. MongoDB becomes unavailable 172
  4.6.3. Kafka Connect process stops gracefully 173
  4.6.4. Kafka Connect process crashes 173
  4.6.5. Kafka becomes unavailable 174
  4.6.6. Connector is stopped for a duration 174
  4.6.7. MongoDB loses writes 174

CHAPTER 5. DEBEZIUM CONNECTOR FOR SQL SERVER ...................................................................... 175
5.1. OVERVIEW OF DEBEZIUM SQL SERVER CONNECTOR 175
5.2. HOW DEBEZIUM SQL SERVER CONNECTORS WORK 176
  5.2.1. How Debezium SQL Server connectors perform database snapshots 176
  5.2.2. How Debezium SQL Server connectors read change data tables 177
  5.2.3. Default names of Kafka topics that receive Debezium SQL Server change event records 177
  5.2.4. How the Debezium SQL Server connector uses the schema change topic 178
5.2.5. Descriptions of Debezium SQL Server connector data change events
   5.2.5.1. About keys in Debezium SQL Server change events
   5.2.5.2. About values in Debezium SQL Server change events
   5.2.6. Debezium SQL Server connector-generated events that represent transaction boundaries
   5.2.6.1. Change data event enrichment
   5.2.7. How Debezium SQL Server connectors map data types
5.3. SETTING UP SQL SERVER TO RUN A DEBEZIUM CONNECTOR
   5.3.1. Enabling CDC on the SQL Server database
   5.3.2. Enabling CDC on a SQL Server table
   5.3.3. Verifying that the user has access to the CDC table
   5.3.4. SQL Server on Azure
   5.3.5. Effect of SQL Server capture job agent configuration on server load and latency
   5.3.6. SQL Server capture job agent configuration parameters
5.4. DEPLOYMENT OF DEBEZIUM SQL SERVER CONNECTORS
   5.4.1. Deploying Debezium SQL Server connectors
   5.4.2. Descriptions of Debezium SQL Server connector configuration properties
5.5. REFRESHING CAPTURE TABLES AFTER A SCHEMA CHANGE
   5.5.1. Running an offline update after a schema change
   5.5.2. Running an online update after a schema change
5.6. MONITORING DEBEZIUM SQL SERVER CONNECTOR PERFORMANCE
   5.6.1. Debezium SQL Server connector snapshot metrics
   5.6.2. Debezium SQL Server connector streaming metrics
   5.6.3. Debezium SQL Server connector schema history metrics

CHAPTER 6. DEBEZIUM CONNECTOR FOR DB2

6.1. OVERVIEW OF DEBEZIUM DB2 CONNECTOR
6.2. HOW DEBEZIUM DB2 CONNECTORS WORK
   6.2.1. How Debezium Db2 connectors perform database snapshots
   6.2.2. How Debezium Db2 connectors read change-data tables
   6.2.3. Default names of Kafka topics that receive Debezium Db2 change event records
   6.2.4. About the Debezium Db2 connector schema change topic
   6.2.5. Debezium Db2 connector-generated events that represent transaction boundaries
6.3. DESCRIPTIONS OF DEBEZIUM DB2 CONNECTOR DATA CHANGE EVENTS
   6.3.1. About keys in Debezium db2 change events
   6.3.2. About values in Debezium Db2 change events
6.4. HOW DEBEZIUM DB2 CONNECTORS MAP DATA TYPES
6.5. SETTING UP DB2 TO RUN A DEBEZIUM CONNECTOR
   6.5.1. Configuring Db2 tables for change data capture
   6.5.2. Effect of Db2 capture agent configuration on server load and latency
   6.5.3. Db2 capture agent configuration parameters
6.6. DEPLOYMENT OF DEBEZIUM DB2 CONNECTORS
   6.6.1. Deploying Debezium Db2 connectors
   6.6.2. Description of Debezium Db2 connector configuration properties
6.7. MONITORING DEBEZIUM DB2 CONNECTOR PERFORMANCE
   6.7.1. Monitoring Debezium during snapshots of Db2 databases
   6.7.2. Monitoring Debezium Db2 connector record streaming
   6.7.3. Monitoring Debezium Db2 connector schema history
6.8. MANAGING DEBEZIUM DB2 CONNECTORS
6.9. UPDATING SCHEMAS FOR DB2 TABLES IN CAPTURE MODE FOR DEBEZIUM CONNECTORS
   6.9.1. Performing offline schema updates for Debezium Db2 connectors
   6.9.2. Performing online schema updates for Debezium Db2 connectors

CHAPTER 7. MONITORING DEBEZIUM
7.1. METRICS FOR MONITORING DEBEZIUM CONNECTORS
7.2. ENABLING JMX IN LOCAL INSTALLATIONS
  7.2.1. Zookeeper JMX environment variables
  7.2.2. Kafka JMX environment variables
  7.2.3. Kafka Connect JMX environment variables
7.3. MONITORING DEBEZIUM ON OPENSIFT

CHAPTER 8. DEBEZIUM LOGGING
  8.1. DEBEZIUM LOGGING CONCEPTS
  8.2. THE DEFAULT DEBEZIUM LOGGING CONFIGURATION
  8.3. CONFIGURING DEBEZIUM LOGGING
    8.3.1. Changing the Debezium logging level
    8.3.2. Adding Debezium mapped diagnostic contexts
  8.4. DEBEZIUM LOGGING ON OPENSIFT

CHAPTER 9. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION
  9.1. ROUTING DEBEZIUM EVENT RECORDS TO TOPICS THAT YOU SPECIFY
    9.1.1. Use case for routing Debezium records to topics that you specify
    9.1.2. Example of routing Debezium records for multiple tables to one topic
    9.1.3. Ensuring unique keys across Debezium records routed to the same topic
    9.1.4. Options for configuring Debezium topic routing transformation
  9.2. ROUTING CHANGE EVENT RECORDS TO TOPICS ACCORDING TO EVENT CONTENT
    9.2.1. Setting up the Debezium content-based-routing SMT
    9.2.2. Example: Debezium basic content-based routing configuration
    9.2.3. Variables for use in Debezium content-based routing expressions
    9.2.4. Configuration of content-based routing conditions for other scripting languages
    9.2.5. Options for configuring the content-based routing transformation
  9.3. CUSTOMIZATION OF KAFKA CONNECT AUTOMATIC TOPIC CREATION
    9.3.1. Disabling automatic topic creation for the Kafka broker
    9.3.2. Configuring automatic topic creation in Kafka Connect
    9.3.3. Configuration of automatically created topics
      9.3.3.1. Topic creation groups
      9.3.3.2. Topic creation group configuration properties
      9.3.3.3. Specifying the configuration for the Debezium default topic creation group
      9.3.3.4. Specifying the configuration for Debezium custom topic creation groups
      9.3.3.5. Registering Debezium custom topic creation groups
  9.4. FILTERING DEBEZIUM CHANGE EVENT RECORDS
    9.4.1. Setting up the Debezium filter SMT
    9.4.2. Example: Debezium basic filter SMT configuration
    9.4.3. Variables for use in filter expressions
    9.4.4. Filter condition configuration for other scripting languages
    9.4.5. Options for configuring filter transformation
  9.5. EXTRACTING SOURCE RECORD AFTER STATE FROM DEBEZIUM CHANGE EVENTS
    9.5.1. Description of Debezium change event structure
    9.5.2. Behavior of Debezium event flattening transformation
    9.5.3. Configuration of Debezium event flattening transformation
    9.5.4. Example of adding Debezium metadata to the Kafka record
    9.5.5. Options for configuring Debezium event flattening transformation
  9.6. CONFIGURING DEBEZIUM CONNECTORS TO USE AVRO SERIALIZATION
    9.6.1. About the Service Registry
    9.6.2. Overview of deploying a Debezium connector that uses Avro serialization
    9.6.3. Deploying connectors that use Avro in Debezium containers
    9.6.4. About Avro name requirements
9.7. CONFIGURING DEBEZIUM CONNECTORS TO USE THE OUTBOX PATTERN

9.7.1. Example of aDebezium outbox message
9.7.2. Outbox table structure expected by Debezium outbox event router SMT
9.7.3. Basic Debezium outbox event router SMT configuration
9.7.4. Using Avro as the payload format in Debezium outbox messages
9.7.5. Emitting additional fields in Debezium outbox messages
9.7.6. Options for configuring outbox event router transformation

9.8. EMITTING DEBEZIUM CHANGE EVENT RECORDS IN CLOUDEVENTS FORMAT

9.8.1. Example Debezium change event records in CloudEvents format
9.8.2. Example of configuring Debezium CloudEvents converter
9.8.3. Debezium CloudEvents converter configuration options
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 connectors:

- Chapter 1, *High level overview of Debezium*
- Chapter 2, *Debezium connector for MySQL*
- Chapter 3, *Debezium connector for PostgreSQL*
- Chapter 4, *Debezium connector for MongoDB*
- Chapter 5, *Debezium connector for SQL Server*
- Chapter 6, *Debezium connector for Db2*
- Chapter 7, *Monitoring Debezium*
- Chapter 8, *Debezium logging*
- Chapter 9, *Configuring Debezium connectors for your application*

**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.
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 \texttt{INSERT}, \texttt{UPDATE}, and \texttt{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 \textit{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 2.1, "How Debezium MySQL connectors work"
- **Section 2.2, "Descriptions of Debezium MySQL connector data change events"
- **Section 2.3, "How Debezium MySQL connectors map data types"
- **Section 2.4, "Setting up MySQL to run a Debezium connector"
- **Section 2.5, "Deployment of Debezium MySQL connectors"
- **Section 2.6, "Monitoring Debezium MySQL connector performance"
- **Section 2.7, "How Debezium MySQL connectors handle faults and problems"

## 2.1. 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 2.1.1, “MySQL topologies supported by Debezium connectors”
- **Section 2.1.2, “How Debezium MySQL connectors handle database schema changes”
- **Section 2.1.3, “How Debezium MySQL connectors expose database schema changes”
- **Section 2.1.4, “How Debezium MySQL connectors perform database snapshots”
- **Section 2.1.5, “Default names of Kafka topics that receive Debezium MySQL change event records”

### 2.1.1. 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 (\textit{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.

### 2.1.2. 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 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.

### 2.1.3. 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:

```json
{
    "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": "1.4.2.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.

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.

### 2.1.4. 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 2.1. Workflow for performing an initial snapshot with a global read lock

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Grabs a global read lock that blocks writes by other database clients.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the consistent snapshot.</td>
</tr>
<tr>
<td>3</td>
<td>Reads the current binlog position.</td>
</tr>
<tr>
<td>4</td>
<td>Reads the schema of the databases and tables for which the connector is configured to capture changes.</td>
</tr>
<tr>
<td>5</td>
<td>Releases the global read lock. Other database clients can now write to the database.</td>
</tr>
</tbody>
</table>
If applicable, writes the DDL changes to the schema change topic, including all necessary DROP... and CREATE... DDL statements.

Scans the database tables. For each row, the connector emits CREATE events to the relevant table-specific Kafka topics.

Commits the transaction.

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Obtains table-level locks.</td>
</tr>
<tr>
<td>2</td>
<td>Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the consistent snapshot.</td>
</tr>
<tr>
<td>3</td>
<td>Reads and filters the names of the databases and tables.</td>
</tr>
<tr>
<td>4</td>
<td>Reads the current binlog position.</td>
</tr>
<tr>
<td>5</td>
<td>Reads the schema of the databases and tables for which the connector is configured to capture changes.</td>
</tr>
<tr>
<td>6</td>
<td>If applicable, writes the DDL changes to the schema change topic, including all necessary DROP... and CREATE... DDL statements.</td>
</tr>
<tr>
<td>7</td>
<td>Scans the database tables. For each row, the connector emits CREATE events to the relevant table-specific Kafka topics.</td>
</tr>
</tbody>
</table>
2.1.5. 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
```

2.2. 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:

```json
{
  "schema": { 1
    ...
  },
  "payload": { 2
    ...
  }
}
```
Table 2.3. Overview of change event basic content

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

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 2.2.1, “About keys in Debezium MySQL change events”
- Section 2.2.2, “About values in Debezium MySQL change events”

2.2.1. 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 ⑤
            }
        ]
    }
```
Table 2.4. Description of change event key

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s payload portion.</td>
</tr>
</tbody>
</table>
| 2    | mysql-server-1.inventory.customers.Key | Name of the schema that defines the structure of the key’s payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format connector-name.database-name.table-name.Key. In this example:  
  - mysql-server-1 is the name of the connector that generated this event.  
  - inventory is the database that contains the table that was changed.  
  - customers is the table that was updated. |
| 3    | optional   | Indicates whether the event key must contain a value in its payload field. In this example, a value in the key’s payload is required. A value in the key’s payload field is optional when a table does not have a primary key. |
| 4    | fields     | Specifies each field that is expected in the payload, including each field’s name, type, and whether it is required. |
| 5    | payload    | Contains the key for the row for which this change event was generated. In this example, the key, contains a single id field whose value is 1001. |

2.2.2. 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:

```json
{
  "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": "mysql-server-1.inventory.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": "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",
"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": {
"op": "c", 6
"ts_ms": 1465491411815, 7
"before": null, 8
"after": {
"id": 1004,
"first_name": "Anne",
"last_name": "Kretchmar",
"email": "annek@noanswer.org"
},
"source": {
"version": "1.4.2.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 2.5. Descriptions of create event value fields**

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>schema</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
| 2    | **name**   | In the **schema** section, each **name** field specifies the schema for a field in the value’s payload.  

*mysql-server-1.inventory.customers.Value* is the schema for the payload’s **before** and **after** fields. This schema is specific to the **customers** table.  

Names of schemas for **before** and **after** fields are of the form **logicalName.tableName.Value**, which ensures that the schema name is unique in the database. This means that when using the **Avro converter**, the resulting Avro schema for each table in each logical source has its own evolution and history. |
### Item 3: Field name - 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.

### Item 4: Field name - 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 5: Field name - payload
- The value’s actual data. This is the information that the change event is providing.

   It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the **Avro converter**, you can significantly decrease the size of the messages that the connector streams to Kafka topics.

### Item 6: Field name - op
- Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are:
  - **c** = create
  - **u** = update
  - **d** = delete
  - **r** = read (applies to only snapshots)

### Item 7: Field name - 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 8: Field name - 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.

### Item 9: Field name - 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.
source

Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes:

- Debezium version
- Connector name
- binlog name where the event was recorded
- binlog position
- Row within the event
- If the event was part of a snapshot
- Name of the database and table that contain the new row
- ID of the MySQL thread that created the event (non-snapshot only)
- MySQL server ID (if available)
- Timestamp for when the change was made in the database

If the `binlog_rows_query_log_events` MySQL configuration option is enabled and the connector configuration `include.query` property is enabled, the `source` field also provides the `query` field, which contains the original SQL statement that caused the change event.

update events

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

```json
{
  "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"
    }
  }
}
```
Table 2.6. Descriptions of update event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>An optional field that specifies the state of the row before the event occurred. In an update event value, the <strong>before</strong> field contains a field for each table column and the value that was in that column before the database commit. In this example, the <code>first_name</code> value is <strong>Anne</strong>.</td>
</tr>
<tr>
<td>2</td>
<td>after</td>
<td>An optional field that specifies the state of the row after the event occurred. You can compare the <strong>before</strong> and <strong>after</strong> structures to determine what the update to this row was. In the example, the <code>first_name</code> value is now <strong>Anne Marie</strong>.</td>
</tr>
</tbody>
</table>
### Item 3: source
Mandatory field that describes the source metadata for the event. The `source` field structure has the same fields as in a create event, but some values are different, for example, the sample `update` event is from a different position in the binlog. The source metadata includes:

- Debezium version
- Connector name
- Binlog name where the event was recorded
- Binlog position
- Row within the event
- If the event was part of a snapshot
- Name of the database and table that contain the updated row
- ID of the MySQL thread that created the event (non-snapshot only)
- MySQL server ID (if available)
- Timestamp for when the change was made in the database

If the `binlog_rows_query_log_events` MySQL configuration option is enabled and the connector configuration `include.query` property is enabled, the `source` field also provides the `query` field, which contains the original SQL statement that caused the change event.

### Item 4: op
Mandatory string that describes the type of operation. In an `update` event value, the `op` field value is `u`, signifying that this row changed because of an update.

### Item 5: ts_ms
Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.

In the `source` object, `ts_ms` indicates the time that the change was made in the database. By comparing the value for `payload.source.ts_ms` with the value for `payload.ts_ms`, you can determine the lag between the source database update and Debezium.

### NOTE
Updating the columns for a row’s primary/unique key changes the value of the row’s key. When a key changes, Debezium outputs three events: a `DELETE` event and a tombstone event with the old key for the row, followed by an event with the new key for the row. Details are in the next section.

### Primary key updates
An `UPDATE` operation that changes a row’s primary key field(s) is known as a primary key change. For a primary key change, in place of an `UPDATE` event record, the connector emits a `DELETE` event record.
for the old key and a CREATE event record for the new (updated) key. These events have the usual structure and content, and in addition, each one has a message header related to the primary key change:

- The DELETE event record has __debezium.newkey as a message header. The value of this header is the new primary key for the updated row.
- The CREATE event record has __debezium.oldkey as a message header. The value of this header is the previous (old) primary key that the updated row had.

**delete events**

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

```json
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,
    "source": {
      "version": "1.4.2.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",
    "ts_ms": 1465581902461
  }
}
```

**Table 2.7. Descriptions of delete event value fields**

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>Optional field that specifies the state of the row before the event occurred. In a delete event value, the before field contains the values that were in the row before it was deleted with the database commit.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>after</td>
<td>Optional field that specifies the state of the row after the event occurred. In a delete event value, the <strong>after</strong> field is <strong>null</strong>, signifying that the row no longer exists.</td>
</tr>
</tbody>
</table>
| 3 | source | Mandatory field that describes the source metadata for the event. In a delete event value, the **source** field structure is the same as for create and update events for the same table. Many **source** field values are also the same. In a delete event value, the **ts_ms** and **pos** field values, as well as other values, might have changed. But the **source** field in a delete event value provides the same metadata:  
  - Debezium version  
  - Connector name  
  - binlog name where the event was recorded  
  - binlog position  
  - Row within the event  
  - If the event was part of a snapshot  
  - Name of the database and table that contain the updated row  
  - ID of the MySQL thread that created the event (non-snapshot only)  
  - MySQL server ID (if available)  
  - Timestamp for when the change was made in the database  
  If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration include.query property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event. |
| 4 | op | Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted. |
| 5 | ts_ms | Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.  
In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium. |
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.

### 2.3. 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>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN, BOOL</td>
<td>BOOLEAN</td>
<td>n/a</td>
</tr>
<tr>
<td>BIT(1)</td>
<td>BOOLEAN</td>
<td>n/a</td>
</tr>
<tr>
<td>MySQL type</td>
<td>Literal type</td>
<td>Semantic type</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>BIT(&gt;1)</td>
<td>BYTES</td>
<td>io.debezium.data.Bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>length</code> schema parameter contains an integer that represents the number of bits. The <code>byte[]</code> contains the bits in little-endian form and is sized to contain the specified number of bits. For example, where <code>n</code> is bits: <code>numBytes = n/8 + (n%8== 0 ? 0 : 1)</code></td>
</tr>
<tr>
<td>TINYINT</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>SMALLINT[(M)]</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>MEDIUMINT[(M)]</td>
<td>INT32</td>
<td>n/a</td>
</tr>
<tr>
<td>INT, INTEGER[(M)]</td>
<td>INT32</td>
<td>n/a</td>
</tr>
<tr>
<td>BIGINT[(M)]</td>
<td>INT64</td>
<td>n/a</td>
</tr>
<tr>
<td>REAL[(M,D)]</td>
<td>FLOAT32</td>
<td>n/a</td>
</tr>
<tr>
<td>FLOAT[(M,D)]</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>DOUBLE[(M,D)]</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>CHAR(M)</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>VARCHAR(M)</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>BINARY(M)</td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</td>
</tr>
<tr>
<td>VARBINARY(M)</td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</td>
</tr>
<tr>
<td>TINYBLOB</td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</td>
</tr>
<tr>
<td>TINYTEXT</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>MySQL type</td>
<td>Literal type</td>
<td>Semantic type</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>BLOB</strong></td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><strong>TEXT</strong></td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>MEDIUMBLOB</strong></td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><strong>MEDIUMTEXT</strong></td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>LONGBLOB</strong></td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><strong>LONGTEXT</strong></td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>JSON</strong></td>
<td>STRING</td>
<td>io.debezium.data.Json</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a JSON document, array, or scalar.</td>
</tr>
<tr>
<td><strong>ENUM</strong></td>
<td>STRING</td>
<td>io.debezium.data.Enum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The allowed schema parameter contains the comma-separated list of allowed values.</td>
</tr>
<tr>
<td><strong>SET</strong></td>
<td>STRING</td>
<td>io.debezium.data.EnumSet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The allowed schema parameter contains the comma-separated list of allowed values.</td>
</tr>
<tr>
<td>**YEAR[(2</td>
<td>4)]**</td>
<td>INT32</td>
</tr>
<tr>
<td><strong>TIMESTAMP[(M)]</strong></td>
<td>STRING</td>
<td>io.debezium.time.ZonedTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In ISO 8601 format with microsecond precision. MySQL allows M to be in the range of 0-6.</td>
</tr>
</tbody>
</table>

**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`.

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>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DATE</code></td>
<td>INT32</td>
<td><code>io.debezium.time.Date</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td><code>TIME[(M)]</code></td>
<td>INT64</td>
<td><code>io.debezium.time.MicroTime</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the time value in microseconds and does not include time zone information. MySQL allows <code>M</code> to be in the range of <code>0-6</code>.</td>
</tr>
<tr>
<td><code>DATETIME</code>, <code>DATETIME(0)</code>, <code>DATETIME(1)</code>, <code>DATETIME(2)</code>, <code>DATETIME(3)</code></td>
<td>INT64</td>
<td><code>io.debezium.time.Timestamp</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past the epoch and does not include time zone information.</td>
</tr>
<tr>
<td>MySQL type</td>
<td>Literal type</td>
<td>Semantic type</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>DATETIME(4), DATETIME(5), DATETIME(6)</td>
<td>INT64</td>
<td>io.debezium.time.MicroTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of microseconds past the epoch and does not include time zone information.</td>
</tr>
</tbody>
</table>

**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 2.10. Mappings when time.precision.mode=connect**

<table>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>org.apache.kafka.connect.data.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME[(M)]</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the time value in microseconds since midnight and does not include time zone information.</td>
</tr>
<tr>
<td>DATETIME[(M)]</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds since the epoch, and does not include time zone information.</td>
</tr>
</tbody>
</table>

**Decimal types**

Debezium connectors handle decimals according to the setting of the **decimal.handling.mode** connector configuration property.

**decimal.handling.mode=precise**

**Table 2.11. Mappings when decimal.handling.mode=precise**

<table>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(M[,D])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The scale schema parameter contains an integer that represents how many digits the decimal point shifted.</td>
</tr>
<tr>
<td>DECIMAL[(M[,D])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The scale schema parameter contains an integer that represents how many digits the decimal point shifted.</td>
</tr>
</tbody>
</table>

**decimal.handling.mode=double**
### Table 2.12. Mappings when `decimal.handling.mode=double`

<table>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(M[,D])]</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>DECIMAL[(M[,D])]</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
</tbody>
</table>

**decimal.handling.mode=string**

### Table 2.13. Mappings when `decimal.handling.mode=string`

<table>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(M[,D])]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>DECIMAL[(M[,D])]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### 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:

```bash
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 2.14. Description of spatial type mappings

<table>
<thead>
<tr>
<th>MySQL type</th>
<th>Literal type</th>
<th>Semantic type</th>
</tr>
</thead>
</table>

37
### 2.4. 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 2.4.1, “Creating a MySQL user for a Debezium connector”
- Section 2.4.2, “Enabling the MySQL binlog for Debezium”
- Section 2.4.3, "Enabling MySQL Global Transaction Identifiers for Debezium”
- Section 2.4.4, “Configuring MySQL session timesouts for Debezium”
- Section 2.4.5, “Enabling query log events for Debezium MySQL connectors”

#### 2.4.1. Creating a MySQL user for a Debezium connector

A Debezium MySQL connector requires a MySQL user account. This MySQL user must have appropriate permissions on all databases for which the Debezium MySQL connector captures changes.

##### Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.

##### Procedure

1. Create the MySQL user:

   ```
   mysql> CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';
   ```

2. Grant the required permissions to the user:

   ```
   mysql> GRANT SELECT, RELOAD, SHOW DATABASES, REPLICATION SLAVE,
   REPLICATION CLIENT ON *.* TO 'user' IDENTIFIED BY 'password';
   ```

   The table below describes the permissions.
IMPORTANT

If using a hosted option such as Amazon RDS or Amazon Aurora that does not allow a global read lock, table-level locks are used to create the consistent snapshot. In this case, you need to also grant LOCK TABLES permissions to the user that you create. See snapshots for more details.

3. Finalize the user’s permissions:

    mysql> FLUSH PRIVILEGES;

Table 2.15. Descriptions of user permissions

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Enables the connector to select rows from tables in databases. This is used only when performing a snapshot.</td>
</tr>
<tr>
<td>RELOAD</td>
<td>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.</td>
</tr>
<tr>
<td>SHOW DATABASES</td>
<td>Enables the connector to see database names by issuing the SHOW DATABASE statement. This is used only when performing a snapshot.</td>
</tr>
<tr>
<td>REPLICATION SLAVE</td>
<td>Enables the connector to connect to and read the MySQL server binlog.</td>
</tr>
<tr>
<td>REPLICATION CLIENT</td>
<td>Enables the connector the use of the following statements:</td>
</tr>
<tr>
<td></td>
<td>• SHOW MASTER STATUS</td>
</tr>
<tr>
<td></td>
<td>• SHOW SLAVE STATUS</td>
</tr>
<tr>
<td></td>
<td>• SHOW BINARY LOGS</td>
</tr>
<tr>
<td></td>
<td>The connector always requires this.</td>
</tr>
<tr>
<td>ON</td>
<td>Identifies the database to which the permissions apply.</td>
</tr>
<tr>
<td>TO 'user'</td>
<td>Specifies the user to grant the permissions to.</td>
</tr>
<tr>
<td>IDENTIFIED BY 'password'</td>
<td>Specifies the user’s MySQL password.</td>
</tr>
</tbody>
</table>

2.4.2. 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:

```sql
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:

```properties
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:

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server-id</td>
<td>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.</td>
</tr>
<tr>
<td>log_bin</td>
<td>The value of log_bin is the base name of the sequence of binlog files.</td>
</tr>
<tr>
<td>binlog_format</td>
<td>The binlog-format must be set to ROW or row.</td>
</tr>
<tr>
<td>binlog_row_image</td>
<td>The binlog_row_image must be set to FULL or full.</td>
</tr>
<tr>
<td>expire_logs_days</td>
<td>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.</td>
</tr>
</tbody>
</table>

2.4.3. 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 2.17. Descriptions of GTID options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gtid_mode</code></td>
<td>Boolean that specifies whether GTID mode of the MySQL server is enabled or not.</td>
</tr>
<tr>
<td></td>
<td>- <strong>ON</strong> = enabled</td>
</tr>
<tr>
<td></td>
<td>- <strong>OFF</strong> = disabled</td>
</tr>
<tr>
<td><code>enforce_gtid_consistency</code></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>- <strong>ON</strong> = enabled</td>
</tr>
<tr>
<td></td>
<td>- <strong>OFF</strong> = disabled</td>
</tr>
</tbody>
</table>

**2.4.4. 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 2.18. Descriptions of MySQL session timeout options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interactive_timeout</code></td>
<td>The number of seconds the server waits for activity on an interactive connection before closing it. See MySQL's documentation for more details.</td>
</tr>
<tr>
<td><code>wait_timeout</code></td>
<td>The number of seconds the server waits for activity on a non-interactive connection before closing it. See MySQL's documentation for more details.</td>
</tr>
</tbody>
</table>

### 2.4.5. 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

## 2.5. 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 2.5.1, “Deploying Debezium MySQL connectors”
- Section 2.5.2, “Description of Debezium MySQL connector configuration properties”

### 2.5.1. 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 Installing Debezium 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:
c. Create a Docker file that uses `registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0` as the base image. For example, from a terminal window, enter the following, replacing `my-plugins` with the name of your plug-ins directory:

```bash
cat <<EOF >debezium-container-for-mysql.yaml
FROM registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/
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-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:

```bash
podman build -t debezium-container-for-mysql:latest .
grep 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:

```bash
podman push <myregistry.io>/debezium-container-for-mysql:latest
grep 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:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
type: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
```
spec:
  #...
  image: debezium-container-for-mysql

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**

```yaml
apiVersion: kafka.strimzi.io/v1beta1
class: io.debezium.connector.mysql.MySqlConnector
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.bootstraps.servers: my-cluster-kafka-bootstrap:9092
database.history.kafka.topic: schema-changes.inventory
database.include.list: inventory
database.history.kafka.bootstraps.servers: my-cluster-kafka-bootstrap:9092
```
Table 2.19. Descriptions of connector configuration settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The name of the connector.</td>
</tr>
<tr>
<td>2</td>
<td>Only one task should operate at any one time. Because the MySQL connector reads the MySQL server’s <strong>binlog</strong>, 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.</td>
</tr>
<tr>
<td>3</td>
<td>The connector’s configuration.</td>
</tr>
<tr>
<td>4</td>
<td>The database host, which is the name of the container running the MySQL server (<strong>mysql</strong>).</td>
</tr>
<tr>
<td>5</td>
<td>Unique ID of the connector.</td>
</tr>
<tr>
<td>6</td>
<td>Logical name of the MySQL server or cluster. This name is used as the prefix for all Kafka topics that receive change event records.</td>
</tr>
<tr>
<td>7</td>
<td>Changes in only the <strong>inventory</strong> database are captured.</td>
</tr>
<tr>
<td>8</td>
<td>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.</td>
</tr>
<tr>
<td>9</td>
<td>The name of the database history topic. This topic is for internal use only and should not be used by consumers.</td>
</tr>
</tbody>
</table>

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.

### 2.5.2. 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**
- **Pass-through configuration properties**

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

**Table 2.20. Required MySQL connector configuration properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td></td>
<td>Unique name for the connector. Attempting to register again with the same name fails. This property is required by all Kafka Connect connectors.</td>
</tr>
<tr>
<td><code>connector.class</code></td>
<td></td>
<td>The name of the Java class for the connector. Always specify <code>io.debezium.connector.mysql.MySqlConnector</code> for the MySQL connector.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tasks.max</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>database.hostname</td>
<td></td>
<td>IP address or host name of the MySQL database server.</td>
</tr>
<tr>
<td>database.port</td>
<td>3306</td>
<td>Integer port number of the MySQL database server.</td>
</tr>
<tr>
<td>database.user</td>
<td></td>
<td>Name of the MySQL user to use when connecting to the MySQL database server.</td>
</tr>
<tr>
<td>database.password</td>
<td></td>
<td>Password to use when connecting to the MySQL database server.</td>
</tr>
<tr>
<td>database.server.name</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>database.server.id</td>
<td>random</td>
<td>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.</td>
</tr>
<tr>
<td>database.history.kafka.topic</td>
<td></td>
<td>The full name of the Kafka topic where the connector stores the database schema history.</td>
</tr>
<tr>
<td>database.history.kafka.bootstrap.servers</td>
<td></td>
<td>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 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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>database.include.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>database.include.list</code>. By default, the connector captures changes in all databases. Do not also set the <code>database.exclude.list</code> connector configuration property.</td>
</tr>
<tr>
<td><code>database.exclude.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>database.exclude.list</code>. Do not also set the <code>database.include.list</code> connector configuration property.</td>
</tr>
<tr>
<td><code>table.include.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>table.include.list</code>. Each identifier is of the form <code>databaseName.tableName</code>. By default, the connector captures changes in every non-system table in each database whose changes are being captured. Do not also specify the <code>table.exclude.list</code> connector configuration property.</td>
</tr>
<tr>
<td><code>table.exclude.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>table.exclude.list</code>. Each identifier is of the form <code>databaseName.tableName</code>. Do not also specify the <code>table.include.list</code> connector configuration property.</td>
</tr>
<tr>
<td><code>column.exclude.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>databaseName.tableName.columnName</code>.</td>
</tr>
<tr>
<td><code>column.include.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>databaseName.tableName.columnName</code>.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>column.truncate.to.length.chars</td>
<td>n/a</td>
<td>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 <code>databaseName.tableName.columnName</code>.</td>
</tr>
<tr>
<td>column.mask.with.length.chars</td>
<td>n/a</td>
<td>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 <code>databaseName.tableName.columnName</code>.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>column.mask.hash._hashAlgorithm_.with.salt_.salt_</code></td>
<td>n/a</td>
<td>An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in the change event record values. Pseudonyms consist of the hashed value obtained by applying the algorithm <code>hashAlgorithm</code> and salt <code>salt</code>. Based on the hash function used, referential integrity is kept while data is pseudonymized. Supported hash functions are described in the <em><a href="https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html">MessageDigest section</a></em> of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash result is automatically shortened to the length of the column. 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 <code>databaseName.tableName.columnName</code>. For example: <code>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</code> CzQMA0cB5K is a randomly selected salt. Depending on the configured <code>hashAlgorithm</code>, the selected <code>salt</code>, and the actual data set, the resulting masked data set might not be completely anonymized.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>column.propagate.source.type</strong></td>
<td>n/a</td>
<td>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:</td>
</tr>
</tbody>
</table>
|                                              |         | __Debezium.source.column.type__  
|                                              |         | __Debezium.source.column.length__  
|                                              |         | __Debezium.source.column.scale__  
|                                              |         | are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of one of these forms:  |
|                                              |         | databaseName.tableName.columnName  
|                                              |         | databaseName.schemaName.tableName.columnName  
| **datatype.propagate.source.type**           | n/a     | An optional, comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change event records. These schema parameters:  |
|                                              |         | __debezium.source.column.type__  
|                                              |         | __debezium.source.column.length__  
|                                              |         | __debezium.source.column.scale__  
|                                              |         | are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of one of these forms:  |
|                                              |         | databaseName.tableName.typeName  
|                                              |         | databaseName.schemaName.tableName.typeName  
|                                              |         | See how MySQL connectors map data types for the list of MySQL-specific data type names.                                                                                                                     |
### Property: `time.precision.mode`

**Default**: `adaptive_time_microseconds`

Time, date, and timestamps can be represented with different kinds of precision, including:

- **adaptive_time_microseconds** (the default) captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column’s type, with the exception of TIME type fields, which are always captured as microseconds.

- **connect**: always represents time and timestamp values using Kafka Connect’s built-in representations for Time, Date, and Timestamp, which use millisecond precision regardless of the database columns’ precision.

### Property: `decimal.handling.mode`

**Default**: `precise`

Specifies how the connector should handle values for DECIMAL and NUMERIC columns:

- **precise** (the default) represents them precisely using `java.math.BigDecimal` values represented in change events in a binary form.

- **double** represents them using `double` values, which may result in a loss of precision but is easier to use.

- **string** encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.

### Property: `bigint.unsigned.handling.mode`

**Default**: `long`

Specifies how BIGINT UNSIGNED columns should be represented in change events. Possible settings are:

- **long** represents values by using Java’s `long`, which might not offer the precision but which is easy to use in consumers. `long` is usually the preferred setting.

- **precise** uses `java.math.BigDecimal` to represent values, which are encoded in the change events by using a binary representation and Kafka Connect’s `org.apache.kafka.connect.data.Decimal` type. Use this setting when working with values larger than $2^{63}$, because these values cannot be conveyed by using `long`. 

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| **time.precision.mode**      | `adaptive_time_microseconds` | Time, date, and timestamps can be represented with different kinds of precision, including:  
  - adaptive_time_microseconds (the default) captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column’s type, with the exception of TIME type fields, which are always captured as microseconds.  
  - connect always represents time and timestamp values using Kafka Connect’s built-in representations for Time, Date, and Timestamp, which use millisecond precision regardless of the database columns’ precision. |
| **decimal.handling.mode**    | `precise` | Specifies how the connector should handle values for DECIMAL and NUMERIC columns:  
  - precise (the default) represents them precisely using `java.math.BigDecimal` values represented in change events in a binary form.  
  - double represents them using `double` values, which may result in a loss of precision but is easier to use.  
  - string encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost. |
| **bigint.unsigned.handling.mode** | `long` | Specifies how BIGINT UNSIGNED columns should be represented in change events. Possible settings are:  
  - long represents values by using Java’s `long`, which might not offer the precision but which is easy to use in consumers. `long` is usually the preferred setting.  
  - precise uses `java.math.BigDecimal` to represent values, which are encoded in the change events by using a binary representation and Kafka Connect’s `org.apache.kafka.connect.data.Decimal` type. Use this setting when working with values larger than $2^{63}$, because these values cannot be conveyed by using `long`. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include.schema.changes</code></td>
<td><code>true</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>include.query</code></td>
<td><code>false</code></td>
<td>Boolean value that specifies whether the connector should include the original SQL query that generated the change event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you set this option to <code>true</code> then you must also configure MySQL with the <code>binlog_rows_query_log_events</code> option set to <code>ON</code>. When <code>include.query</code> is <code>true</code>, the query is not present for events that the snapshot process generates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Setting <code>include.query</code> to <code>true</code> 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 <code>false</code>.</td>
</tr>
<tr>
<td><code>event.deserialization.failure.handling.mode</code></td>
<td><code>fail</code></td>
<td>Specifies how the connector should react to exceptions during deserialization of binlog events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fail</code> propagates the exception, which indicates the problematic event and its binlog offset, and causes the connector to stop.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>warn</code> logs the problematic event and its binlog offset and then skips the event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>skip</code> passes over the problematic event and does not log anything.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>inconsistent.schema.handling.mode</td>
<td>fail</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>fail</strong> throws an exception that indicates the problematic event and its binlog offset, and causes the connector to stop.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>warn</strong> logs the problematic event and its binlog offset and skips the event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>skip</strong> passes over the problematic event and does not log anything.</td>
</tr>
<tr>
<td>max.queue.size</td>
<td>8192</td>
<td>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.</td>
</tr>
<tr>
<td>max.batch.size</td>
<td>2048</td>
<td>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.</td>
</tr>
<tr>
<td>max.queue.size.in.bytes</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>poll.interval.ms</td>
<td>1000</td>
<td>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.</td>
</tr>
<tr>
<td>connect.timeout.ms</td>
<td>30000</td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gtid.source.includes</td>
<td></td>
<td>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 <code>gtid.source.excludes</code>.</td>
</tr>
<tr>
<td>gtid.source.excludes</td>
<td></td>
<td>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 <code>gtid.source.includes</code>.</td>
</tr>
<tr>
<td>tombstones.on.delete</td>
<td>true</td>
<td>Controls whether a delete event is followed by a tombstone event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>true</strong> - a delete operation is represented by a delete event and a subsequent tombstone event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>false</strong> - only a delete event is emitted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

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:

```
database-name . table-name : regexp ; ...
```

For example:

```
dBA.table_a:regex_1;dBb.table_b:regex_2;dBc.table_c:regex_3
```

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.

**binary.handling.mode**

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>Specifies how binary columns, for example, blob, binary, varbinary, should be represented in change events. Possible settings:</td>
</tr>
<tr>
<td></td>
<td>bytes represents binary data as a byte array.</td>
</tr>
<tr>
<td></td>
<td>base64 represents binary data as a base64-encoded String.</td>
</tr>
<tr>
<td></td>
<td>hex represents binary data as a hex-encoded (base16) String.</td>
</tr>
</tbody>
</table>

**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 2.21. Descriptions of MySQL connector advanced configuration properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| message.key.columns    | n/a     | 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. 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: `database-name . table-name : regexp ; ...`
<p>| binary.handling.mode   | bytes   | Specifies how binary columns, for example, blob, binary, varbinary, should be represented in change events. Possible settings: bytes represents binary data as a byte array. base64 represents binary data as a base64-encoded String. hex represents binary data as a hex-encoded (base16) String. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect.keep.alive</td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>table.ignore.builtin</td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>database.history.kafka.recovery.poll.interval.ms</td>
<td>100</td>
<td>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.</td>
</tr>
<tr>
<td>database.history.kafka.recovery.attempts</td>
<td>4</td>
<td>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 \times recovery.poll.interval.ms.</td>
</tr>
<tr>
<td>database.history.skip.unparseable.ddl</td>
<td>false</td>
<td>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.</td>
</tr>
<tr>
<td>database.history.store.only.monitored.tables.ddl</td>
<td>false</td>
<td>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.</td>
</tr>
</tbody>
</table>
### database.ssl.mode

**Default:** disabled  
Specifies whether to use an encrypted connection. Possible settings are:

- **disabled** specifies the use of an unencrypted connection.
- **preferred** establishes an encrypted connection if the server supports secure connections. If the server does not support secure connections, falls back to an unencrypted connection.
- **required** establishes an encrypted connection or fails if one cannot be made for any reason.
- **verify_ca** behaves like **required** but additionally it verifies the server TLS certificate against the configured Certificate Authority (CA) certificates and fails if the server TLS certificate does not match any valid CA certificates.
- **verify_identity** behaves like **verify_ca** but additionally verifies that the server certificate matches the host of the remote connection.

### binlog.buffer.size

**Default:** 0  
The size of a look-ahead buffer used by the binlog reader. The default setting of 0 disables buffering.

Under specific conditions, it is possible that the MySQL binlog contains uncommitted data finished by a **ROLLBACK** statement. Typical examples are using savepoints or mixing temporary and regular table changes in a single transaction.

When a beginning of a transaction is detected then Debezium tries to roll forward the binlog position and find either **COMMIT** or **ROLLBACK** so it can determine whether to stream the changes from the transaction. The size of the binlog buffer defines the maximum number of changes in the transaction that Debezium can buffer while searching for transaction boundaries. If the size of the transaction is larger than the buffer then Debezium must rewind and re-read the events that have not fit into the buffer while streaming.

**NOTE:** This feature is incubating. Feedback is encouraged. It is expected that this feature is not completely polished.
**snapshot.mode** | **Default** | **Description**
--- | --- | ---
**initial** | Specifies the criteria for running a snapshot when the connector starts. Possible settings are:

- **initial** - the connector runs a snapshot only when no offsets have been recorded for the logical server name.

- **when_needed** - the connector runs a snapshot upon startup whenever it deems it necessary. That is, when no offsets are available, or when a previously recorded offset specifies a binlog location or GTID that is not available in the server.

- **never** - the connector never uses snapshots. Upon first startup with a logical server name, the connector reads from the beginning of the binlog. Configure this behavior with care. It is valid only when the binlog is guaranteed to contain the entire history of the database.

- **schema_only** - the connector runs a snapshot of the schemas and not the data. This setting is useful when you do not need the topics to contain a consistent snapshot of the data but need them to have only the changes since the connector was started.

- **schema_only_recovery** - this is a recovery setting for a connector that has already been capturing changes. When you restart the connector, this setting enables recovery of a corrupted or lost database history topic. You might set it periodically to "clean up" a database history topic that has been growing unexpectedly. Database history topics require infinite retention.
### Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| snapshot.locking.mode             | minimal | Controls whether and how long the connector holds the global MySQL read lock, which prevents any updates to the database, while the connector is performing a snapshot. Possible settings are:  
|                                   |         | **minimal** - the connector holds the global read lock for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global read lock is no longer held and other MySQL clients are updating the database.  
|                                   |         | **minimal_percona** - the connector holds the global backup lock for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global backup lock is no longer held and other MySQL clients are updating the database. This mode does not flush tables to disk, is not blocked by long-running reads, and is available only in Percona Server.  
|                                   |         | **extended** - blocks all writes for the duration of the snapshot. Use this setting if there are clients that are submitting operations that MySQL excludes from REPEATABLE READ semantics.  
<p>|                                   |         | <strong>none</strong> - prevents the connector from acquiring any table locks during the snapshot. While this setting is allowed with all snapshot modes, it is safe to use if and only if no schema changes are happening while the snapshot is running. For tables defined with MyISAM engine, the tables would still be locked despite this property being set as MyISAM acquires a table lock. This behavior is unlike InnoDB engine, which acquires row level locks. |
| 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 <em>table.include.list</em> for which you <em>want</em> to take the snapshot. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| snapshot.select.statementoverrides           |         | 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 `databaseName.tableName`.  
For each table that you specify, also specify another configuration property: `snapshot.select.statementoverrides.DB_NAME.TABLE_NAME`. For example, the name of the other configuration property might be: `snapshot.select.statementoverrides.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.  
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. |
| min.row.count.to.stream.results              | 1000    | During a snapshot, the connector queries each table for which the connector is configured to capture changes. The connector uses each query result to produce a read event that contains data for all rows in that table. This property determines whether the MySQL connector puts results for a table into memory, which is fast but requires large amounts of memory, or streams the results, which can be slower but work for very large tables. The setting of this property specifies the minimum number of rows a table must contain before the connector streams results.  
To skip all table size checks and always stream all results during a snapshot, set this property to 0. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heartbeat.interval.ms</td>
<td>0</td>
<td>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
| heartbeat.topics.prefix       | __debezium-heartbeat | Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern: heartbeats.topics.prefix.server.name  

For example, if the database server name is fulfillment, the default topic name is __debezium-heartbeat.fulfillment. |
| database.initial.statements   |         | A semicolon separated list of SQL statements to be executed when a JDBC connection, not the connection that is reading the transaction log, to the database is established. To specify a semicolon as a character in a SQL statement and not as a delimiter, use two semicolons, (; ;).  

The connector might establish JDBC connections at its own discretion, so this property is only for configuring session parameters. It is not for executing DML statements. |
<p>| 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           |         | During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snapshot.lock.timeout.ms</td>
<td>10000</td>
<td>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.</td>
</tr>
<tr>
<td>enable.time.adjuster</td>
<td>true</td>
<td>Boolean value that indicates whether the connector converts a 2-digit year specification to 4 digits. Set to false when conversion is fully delegated to the database. MySQL allows users to insert year values with either 2-digits or 4-digits. For 2-digit values, the value gets mapped to a year in the range 1970 - 2069. The default behavior is that the connector does the conversion.</td>
</tr>
<tr>
<td>sanitize.field.names</td>
<td>true if connector configuration sets the key.converter or value.converter property to the Avro converter. false if not.</td>
<td>Indicates whether field names are sanitized to adhere to Avro naming requirements.</td>
</tr>
<tr>
<td>skipped.operations</td>
<td></td>
<td>Comma-separated list of oplog operations to skip during streaming. Values that you can specify are: c for inserts/create, u for updates, d for deletes. By default, no operations are skipped.</td>
</tr>
</tbody>
</table>

**Pass-through configuration properties**

The MySQL connector also supports pass-through configuration properties that are used when creating the Kafka producer and consumer. Specifically, all connector configuration properties that begin with the `database.history.producer.` prefix are used (without the prefix) when creating the Kafka producer that writes to the database history. All properties that begin with the prefix `database.history.consumer.` are used (without the prefix) when creating the Kafka consumer that reads the database history upon connector start-up.

For example, the following connector configuration properties can be used to secure connections to the Kafka broker:

```
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

See the Kafka documentation for more details about pass-through properties.

Pass-through properties for database drivers

In addition to the pass-through properties for the Kafka producer and consumer, there are pass-through properties for database drivers. These properties have the database. prefix. For example, database.tinyInt1isBit=false is passed to the JDBC URL.

2.6. 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.

2.6.1. Monitoring Debezium during snapshots of MySQL databases

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last snapshot event that the connector has read.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>TotalTableCount</td>
<td>int</td>
<td>The total number of tables that are being included in the snapshot.</td>
</tr>
<tr>
<td>RemainingTableCount</td>
<td>int</td>
<td>The number of tables that the snapshot has yet to copy.</td>
</tr>
<tr>
<td>SnapshotRunning</td>
<td>boolean</td>
<td>Whether the snapshot was started.</td>
</tr>
<tr>
<td>SnapshotAborted</td>
<td>boolean</td>
<td>Whether the snapshot was aborted.</td>
</tr>
<tr>
<td>SnapshotCompleted</td>
<td>boolean</td>
<td>Whether the snapshot completed.</td>
</tr>
<tr>
<td>SnapshotDurationInSeconds</td>
<td>long</td>
<td>The total number of seconds that the snapshot has taken so far, even if not complete.</td>
</tr>
<tr>
<td>RowsScanned</td>
<td>Map&lt;String, Long&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>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.</td>
</tr>
</tbody>
</table>
The Debezium MySQL connector also provides the `HoldingGlobalLock` custom snapshot metric. This metric is set to a Boolean value that indicates whether the connector currently holds a global or table write lock.

### 2.6.2. Monitoring Debezium MySQL connector binlog reading

The MBean is `debezium.mysql:type=connector-metrics,context=binlog,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.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CurrentQueueSizeInBytes</code></td>
<td><code>long</code></td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LastEvent</code></td>
<td><code>string</code></td>
<td>The last streaming event that the connector has read.</td>
</tr>
<tr>
<td><code>MilliSecondsSinceLastEvent</code></td>
<td><code>long</code></td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td><code>TotalNumberOfEventsSeen</code></td>
<td><code>long</code></td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td><code>NumberOfEventsFiltered</code></td>
<td><code>long</code></td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td><code>MonitoredTables</code></td>
<td><code>string[]</code></td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td><code>QueueTotalCapacity</code></td>
<td><code>int</code></td>
<td>The length the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td><code>QueueRemainingCapacity</code></td>
<td><code>int</code></td>
<td>The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Connected</td>
<td>boolean</td>
<td>Flag that denotes whether the connector is currently connected to the database server.</td>
</tr>
<tr>
<td>MilliSecondsBehindSource</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfCommittedTransactions</td>
<td>long</td>
<td>The number of processed transactions that were committed.</td>
</tr>
<tr>
<td>SourceEventPosition</td>
<td>Map&lt;String, String&gt;</td>
<td>The coordinates of the last received event.</td>
</tr>
<tr>
<td>LastTransactionId</td>
<td>string</td>
<td>Transaction identifier of the last processed transaction.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>The maximum buffer of the queue in bytes.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

The Debezium MySQL connector also provides the following custom binlog metrics:

**Table 2.22. Descriptions of custom binlog metrics**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinlogFilename</td>
<td>string</td>
<td>The name of the binlog file that the connector has most recently read.</td>
</tr>
<tr>
<td>BinlogPosition</td>
<td>long</td>
<td>The most recent position (in bytes) within the binlog that the connector has read.</td>
</tr>
<tr>
<td>IsGtidModeEnabled</td>
<td>boolean</td>
<td>Flag that denotes whether the connector is currently tracking GTIDs from MySQL server.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GtidSet</td>
<td>string</td>
<td>The string representation of the most recent GTID set processed by the connector when reading the binlog.</td>
</tr>
<tr>
<td>NumberOfSkippedEvents</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfDisconnects</td>
<td>long</td>
<td>The number of disconnects by the MySQL connector.</td>
</tr>
<tr>
<td>NumberOfRolledBackTransactions</td>
<td>long</td>
<td>The number of processed transactions that were rolled back and not streamed.</td>
</tr>
<tr>
<td>NumberOfNotWellFormedTransactions</td>
<td>long</td>
<td>The number of transactions that have not conformed to the expected protocol of <code>BEGIN + COMMIT/ROLLBACK</code>. This value should be 0 under normal conditions.</td>
</tr>
<tr>
<td>NumberOfLargeTransactions</td>
<td>long</td>
<td>The number of transactions that have not fit into the look-ahead buffer. For optimal performance, this value should be significantly smaller than <code>NumberOfCommittedTransactions</code> and <code>NumberOfRolledBackTransactions</code>.</td>
</tr>
</tbody>
</table>

### 2.6.3. Monitoring Debezium MySQL connector schema history

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>string</td>
<td>One of <code>STOPPED</code>, <code>RECOVERING</code> (recovering history from the storage), <code>RUNNING</code> describing the state of the database history.</td>
</tr>
<tr>
<td>RecoveryStartTime</td>
<td>long</td>
<td>The time in epoch seconds at what recovery has started.</td>
</tr>
<tr>
<td>ChangesRecovered</td>
<td>long</td>
<td>The number of changes that were read during recovery phase.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ChangesApplied</td>
<td>long</td>
<td>the total number of schema changes applied during recovery and runtime.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastRecoveredChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was recovered from the history store.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastAppliedChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was applied.</td>
</tr>
<tr>
<td>LastRecoveredChange</td>
<td>string</td>
<td>The string representation of the last change recovered from the history store.</td>
</tr>
<tr>
<td>LastAppliedChange</td>
<td>string</td>
<td>The string representation of the last applied change.</td>
</tr>
</tbody>
</table>

2.7. 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 3. DEBEZIUM CONNECTOR FOR POSTGRESQL

Debezium’s PostgreSQL connector captures row-level changes in the schemas of a PostgreSQL database. PostgreSQL versions 10, 11, and 12 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 3.1, “Overview of Debezium PostgreSQL connector”
- Section 3.2, “How Debezium PostgreSQL connectors work”
- Section 3.3, “Descriptions of Debezium PostgreSQL connector data change events”
- Section 3.4, “How Debezium PostgreSQL connectors map data types”
- Section 3.5, “Setting up PostgreSQL to run a Debezium connector”
- Section 3.6, “Deployment of Debezium PostgreSQL connectors”
- Section 3.7, “Monitoring Debezium PostgreSQL connector performance”
- Section 3.8, “How Debezium PostgreSQL connectors handle faults and problems”

3.1. 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.

### 3.2. 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 3.2.2, “How Debezium PostgreSQL connectors perform database snapshots”
- Section 3.2.3, “How Debezium PostgreSQL connectors stream change event records”
- Section 3.2.4, “Default names of Kafka topics that receive Debezium PostgreSQL change event records”
- Section 3.2.5, “Metadata in Debezium PostgreSQL change event records”
3.2.1. 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.

3.2.2. 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 3.1. Settings for `snapshot.mode` connector configuration property

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
| **always** | The connector always performs a snapshot when it starts. After the snapshot completes, the connector continues streaming changes from step 3 in the above sequence. This mode is useful in these situations:  
  - It is known that some WAL segments have been deleted and are no longer available.  
  - After a cluster failure, a new primary has been promoted. The `always` snapshot mode ensures that the connector does not miss any changes that were made after the new primary had been promoted but before the connector was restarted on the new primary. |
| **never**  | The connector never performs snapshots. When a connector is configured this way, its behavior when it starts is as follows. If there is a previously stored LSN in the Kafka offsets topic, the connector continues streaming changes from that position. If no LSN has been stored, the connector starts streaming changes from the point in time when the PostgreSQL logical replication slot was created on the server. The `never` snapshot mode is useful only when you know all data of interest is still reflected in the WAL. |
| **initial only** | The connector performs a database snapshot and stops before streaming any change event records. If the connector had started but did not complete a snapshot before stopping, the connector restarts the snapshot process and stops when the snapshot completes. |
| **exported** | 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. |

3.2.3. 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.

### 3.2.4. 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`

### 3.2.5. 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:

```json
"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.

### 3.2.6.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**

```json
{
  "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:

```json
{
    "before": null,
    "after": {
        "pk": "2",
        "aa": "1"
    },
    "source": {
        ...
    },
    "op": "c",
    "ts_ms": "1580390884335",
    "transaction": {
        "id": "571",
        "total_order": "1",
        "data_collection_order": "1"
    }
}
```

### 3.3. 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:

```json
{
    "schema": {  
    },
    "payload": {  
    }
}
```
Table 3.2. Overview of change event basic content

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

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

3.3.1. 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": {
        "type": "struct",
        "name": "PostgreSQL_server.public.customers.Key",
        "optional": false,
        "fields": [
            {
                "name": "id",
                "type": "int64",
                "optional": false
            },
            {
                "name": "first_name",
                "type": "string",
                "optional": false
            },
            {
                "name": "last_name",
                "type": "string",
                "optional": false
            },
            {
                "name": "email",
                "type": "string",
                "optional": false
            }
        ]
    }
}
```
Table 3.3. Description of change event key

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s <code>payload</code> portion.</td>
</tr>
</tbody>
</table>
| 2    | PostgreSQL_server.inventory.customers.Key | Name of the schema that defines the structure of the key’s payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format `connector-name.database-name.table-name.Key`. In this example:  
- `PostgreSQL_server` is the name of the connector that generated this event.  
- `inventory` is the database that contains the table that was changed.  
- `customers` is the table that was updated. |
| 3    | optional   | Indicates whether the event key must contain a value in its `payload` field. In this example, a value in the key’s `payload` is required. A value in the key’s `payload` field is optional when a table does not have a primary key. |
| 4    | fields     | Specifies each field that is expected in the `payload`, including each field’s name, index, and schema. |
| 5    | payload    | Contains the key for the row for which this change event was generated. In this example, the key, contains a single `id` field whose value is 1. |

**NOTE**

Although the `column.exclude.list` and `column.include.list` connector configuration properties allow you to capture only a subset of table columns, all columns in a primary or unique key are always included in the event’s key.
### 3.3.2. 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:

```sql
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": {
    "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",
        "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"
      }
    ],
    "optional": false,
    "name": "PostgreSQL_server.inventory.customers.Value",
    "field": "after"
  },
  "type": "create",
  "op": "create",
  "oid": 1
}
```
"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",
"field": "source"
},
{
  "type": "string",
  "optional": false,
  "field": "op"
},
{
  "type": "int64",
  "optional": true,
  "field": "ts_ms"
}],
"optional": false,
"name": "PostgreSQL_server.inventory.customers.Envelope",
"source": {
"before": null,
"after": {
  "id": 1,
  "first_name": "Anne",
  "last_name": "Kretchmar",
  "email": "annek@noanswer.org"
},
"source": {
Table 3.4. Descriptions of create event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>In the schema section, each name field specifies the schema for a field in the value’s payload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PostgreSQL_server.inventory.customers.Value</strong> is the schema for the payload’s before and after fields. This schema is specific to the customers table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Names of schemas for before and after fields are of the form <strong>logicalName.tableName.Value</strong>, 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.</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td><strong>PostgreSQL_server.inventory.customers.Envelope</strong> 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.</td>
</tr>
</tbody>
</table>
The value's actual data. This is the information that the change event is providing.

It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.

An optional field that specifies the state of the row before the event occurred. When the op field is `c` for create, as it is in this example, the `before` field is `null` since this change event is for new content.

**NOTE**

Whether or not this field is available is dependent on the REPLICA IDENTITY setting for each table.

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.

Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes:

- Debezium version
- Connector type and name
- Database and table that contains the new row
- 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 op

Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are:

- **c** = create
- **u** = update
- **d** = delete
- **r** = read (applies to only snapshots)

### Item ts_ms

Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.

In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

### update events

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

```json
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1
    },
    "after": { 2
      "id": 1,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 3
      "version": "1.4.2.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": null,
      "db": "postgres",
      "schema": "public",
      "table": "customers",
      "txId": 556,
    }
  }
}
```
Table 3.5. Descriptions of update event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>An optional field that contains values that were in the row before the database commit. In this example, only the primary key column, id, is present because the table’s REPLICA IDENTITY setting is, by default, DEFAULT. + For an update event to contain the previous values of all columns in the row, you would have to change the customers table by running ALTER TABLE customers REPLICA IDENTITY FULL.</td>
</tr>
<tr>
<td>2</td>
<td>after</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 3    | source     | Mandatory field that describes the source metadata for the event. The source field structure has the same fields as in a create event, but some values are different. The source metadata includes:  
  - Debezium version  
  - Connector type and name  
  - Database and table that contains the new row  
  - Schema name  
  - If the event was part of a snapshot  
  - ID of the transaction in which the operation was performed  
  - Offset of the operation in the database log  
  - Timestamp for when the change was made in the database |
| 4    | op         | Mandatory string that describes the type of operation. In an update event value, the op field value is u, signifying that this row changed because of an update. |
| 5    | ts_ms      | Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the source object, ts_ms indicates the time that the change was made in the database. By comparing the value for payload.source.ts_ms with the value for payload.ts_ms, you can determine the lag between the source database update and Debezium. |
NOTE

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

Primary key updates

An UPDATE operation that changes a row’s primary key field(s) is known as a primary key change. For a primary key change, in place of sending an UPDATE event record, the connector sends a DELETE event record for the old key and a CREATE event record for the new (updated) key. These events have the usual structure and content, and in addition, each one has a message header related to the primary key change:

- The DELETE event record has __debezium.newkey as a message header. The value of this header is the new primary key for the updated row.

- The CREATE event record has __debezium.oldkey as a message header. The value of this header is the previous (old) primary key that the updated row had.

delete events

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

```json
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1
    },
    "after": null,
    "source": {
      "version": "1.4.2.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": null,
      "db": "postgres",
      "schema": "public",
      "table": "customers",
      "txId": 556,
      "lsn": 46523128,
      "xmin": null
    },
    "op": "d",
    "ts_ms": 1465581902461
  }
}
```

Table 3.6. Descriptions of delete event value fields
<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>Optional field that specifies the state of the row before the event occurred. In a delete event value, the <code>before</code> field contains the values that were in the row before it was deleted with the database commit. In this example, the <code>before</code> field contains only the primary key column because the table’s <code>REPLICA IDENTITY</code> setting is <code>DEFAULT</code>.</td>
</tr>
<tr>
<td>2</td>
<td>after</td>
<td>Optional field that specifies the state of the row after the event occurred. In a delete event value, the <code>after</code> field is <code>null</code>, signifying that the row no longer exists.</td>
</tr>
<tr>
<td>3</td>
<td>source</td>
<td>Mandatory field that describes the source metadata for the event. In a delete event value, the <code>source</code> field structure is the same as for create and update events for the same table. Many <code>source</code> field values are also the same. In a delete event value, the <code>ts_ms</code> and <code>lsn</code> field values, as well as other values, might have changed. But the <code>source</code> field in a delete event value provides the same metadata:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debezium version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Connector type and name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Database and table that contains the new row</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Schema name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the event was part of a snapshot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ID of the transaction in which the operation was performed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Offset of the operation in the database log</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Timestamp for when the change was made in the database</td>
</tr>
<tr>
<td>4</td>
<td>op</td>
<td>Mandatory string that describes the type of operation. The <code>op</code> field value is <code>d</code>, signifying that this row was deleted.</td>
</tr>
<tr>
<td>5</td>
<td>ts_ms</td>
<td>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 <code>source</code> object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and Debezium.</td>
</tr>
</tbody>
</table>

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:

```json
{
   "schema": { ... },
   "payload": {
      "source": {
         "version": "1.4.2.Final",
         "connector": "postgresql",
         "name": "PostgreSQL_server",
         "ts_ms": 155903904863,
         "snapshot": false,
         "db": "postgres",
         "schema": "public",
         "table": "customers",
         "txId": 556,
         "lsn": 46523128,
         "xmin": null
      },
      "op": "t",
      "ts_ms": 155903904961
   }
}
```

Table 3.7. Descriptions of truncate event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 1    | source     | Mandatory field that describes the source metadata for the event. In a *truncate* event value, the `source` field structure is the same as for *create*, *update*, and *delete* events for the same table, provides this metadata:  
  - Debezium version  
  - Connector type and name  
  - Database and table that contains the new row  
  - Schema name  
  - If the event was part of a snapshot (always **false** for *delete* events)  
  - ID of the transaction in which the operation was performed  
  - Offset of the operation in the database log  
  - Timestamp for when the change was made in the database |
| 2    | op         | Mandatory string that describes the type of operation. The `op` field value is **t**, signifying that this table was truncated. |
| 3    | ts_ms      | Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.  
In the `source` object, `ts_ms` indicates the time that the change was made in the database. By comparing the value for `payload.source.ts_ms` with the value for `payload.ts_ms`, you can determine the lag between the source database update and Debezium. |

In case a single **TRUNCATE** statement applies to multiple tables, one *truncate* change event record for each truncated table will be emitted.

Note that since *truncate* events represent a change made to an entire table and don’t have a message key, unless you’re working with topics with a single partition, there are no ordering guarantees for the change events pertaining to a table (**create**, **update**, etc.) and *truncate* events for that table. For instance a consumer may receive an **update** event only after a *truncate* event for that table, when those events are read from different partitions.

### 3.4. 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 3.8. Mappings for PostgreSQL basic data types**

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>n/a</td>
</tr>
<tr>
<td>BIT(1)</td>
<td>BOOLEAN</td>
<td>n/a</td>
</tr>
<tr>
<td>BIT( &gt; 1)</td>
<td>BYTES</td>
<td>io.debezium.data.Bits</td>
</tr>
</tbody>
</table>

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, \( \text{numBytes} = \frac{n}{8} + (n \ % \ 8 == 0 \ ? \ 0 : 1) \) where \( n \) is the number of bits.
<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT VARYING[(M)]</td>
<td>BYTES</td>
<td>io.debezium.data.Bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>SMALLINT, SMALLSERIAL</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>INTEGER, SERIAL</td>
<td>INT32</td>
<td>n/a</td>
</tr>
<tr>
<td>BIGINT, BIGSERIAL</td>
<td>INT64</td>
<td>n/a</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT32</td>
<td>n/a</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>CHAR[(M)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>VARCHAR[(M)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>CHARACTER[(M)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>CHARACTER VARYING[(M)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>TIMESTAMPTZ, TIMESTAMP WITH TIME ZONE</td>
<td>STRING</td>
<td>io.debezium.time.ZonedTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A string representation of a timestamp with timezone information, where the timezone is GMT.</td>
</tr>
<tr>
<td>TIMETZ, TIME WITH TIME ZONE</td>
<td>STRING</td>
<td>io.debezium.time.ZonedTime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A string representation of a time value with timezone information, where the timezone is GMT.</td>
</tr>
<tr>
<td>INTERVAL [P]</td>
<td>INT64</td>
<td>io.debezium.time.MicroDuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(default) The approximate number of microseconds for a time interval using the 365.25 / 12.0 formula for days per month average.</td>
</tr>
<tr>
<td>PostgreSQL data type</td>
<td>Literal type (schema type)</td>
<td>Semantic type (schema name) and Notes</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>INTERVAL [P]</td>
<td>STRING</td>
<td>io.debezium.time.Interval</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(when interval.handling.mode is set to string)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The string representation of the interval value that follows the pattern P&lt;years&gt;Y&lt;months&gt;M&lt;days&gt;DT&lt;hours&gt;M&lt;minutes&gt;s&gt;M&lt;seconds&gt;s, for example, P1Y2M3DT4H5M6.78S.</td>
</tr>
<tr>
<td>BYTEA</td>
<td>BYTES or STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Either the raw bytes (the default), a base64-encoded string, or a hex-encoded string, based on the connector’s binary handling mode setting.</td>
</tr>
<tr>
<td>JSON, JSONB</td>
<td>STRING</td>
<td>io.debezium.data.Json</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a JSON document, array, or scalar.</td>
</tr>
<tr>
<td>XML</td>
<td>STRING</td>
<td>io.debezium.data.Xml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of an XML document.</td>
</tr>
<tr>
<td>UUID</td>
<td>STRING</td>
<td>io.debezium.data.Uuid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a PostgreSQL UUID value.</td>
</tr>
<tr>
<td>POINT</td>
<td>STRUCT</td>
<td>io.debezium.data.geometry.Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains a structure with two FLOAT64 fields, (x, y). Each field represents the coordinates of a geometric point.</td>
</tr>
<tr>
<td>LTREE</td>
<td>STRING</td>
<td>io.debezium.data.Ltree</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a PostgreSQL LTREE value.</td>
</tr>
<tr>
<td>CITEXT</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>INET</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>INT4RANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range of integer.</td>
</tr>
<tr>
<td>PostgreSQL data type</td>
<td>Literal type (schema type)</td>
<td>Semantic type (schema name) and Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>INT8RANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range of bigint.</td>
</tr>
<tr>
<td>NUMRANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Range of numeric.</td>
</tr>
<tr>
<td>TSRANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a timestamp range without a time zone.</td>
</tr>
<tr>
<td>TSTZRANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a timestamp range with the local system time zone.</td>
</tr>
<tr>
<td>DATERANGE</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of a date range. It always has an exclusive upper-bound.</td>
</tr>
<tr>
<td>ENUM</td>
<td>STRING</td>
<td>io.debezium.data.Enum</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of the PostgreSQL ENUM value. The set of allowed values is maintained in the allowed schema parameter.</td>
</tr>
</tbody>
</table>

**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 3.9. Mappings when time.precision.mode is adaptive**
<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>io.debezium.time.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME(1), TIME(2), TIME(3)</td>
<td>INT32</td>
<td>io.debezium.time.Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past midnight, and does not include timezone information.</td>
</tr>
<tr>
<td>TIME(4), TIME(5), TIME(6)</td>
<td>INT64</td>
<td>io.debezium.time.MicroTime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of microseconds past midnight, and does not include timezone information.</td>
</tr>
<tr>
<td>TIMESTAMP(1), TIMESTAMP(2), TIMESTAMP(3)</td>
<td>INT64</td>
<td>io.debezium.time.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds since the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>TIMESTAMP(4), TIMESTAMP(5), TIMESTAMP(6), TIMESTAMP</td>
<td>INT64</td>
<td>io.debezium.time.MicroTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of microseconds since the epoch, and does not include timezone information.</td>
</tr>
</tbody>
</table>

**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 3.10. Mappings when `time.precision.mode` is `adaptive_time_microseconds`

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>io.debezium.time.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME</td>
<td>INT64</td>
<td>io.debezium.time.MicroTime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number value in microseconds and does not include timezone information. PostgreSQL allows precision <code>P</code> to be in the range 0–6 to store up to microsecond precision.</td>
</tr>
</tbody>
</table>
**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 3.11. Mappings when **time.precision.mode** is **connect**

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
</table>
**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 timezone 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.

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 3.12. Mappings when decimal.handling.mode is precise

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(M[,D])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The scale schema parameter contains an integer representing how many digits the decimal point was shifted.</td>
</tr>
<tr>
<td>DECIMAL[(M[,D])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The scale schema parameter contains an integer representing how many digits the decimal point was shifted.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC</td>
<td>STRUCT</td>
<td>io.debezium.data.VariableScaleDecimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>STRUCT</td>
<td>io.debezium.data.VariableScaleDecimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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 3.14. Mappings when `decimal.handling.mode` is `double`

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NUMERIC[(M[,D])]</code></td>
<td>FLOAT64</td>
<td></td>
</tr>
<tr>
<td><code>DECIMAL[(M[,D])]</code></td>
<td>FLOAT64</td>
<td></td>
</tr>
</tbody>
</table>

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 3.15. Mappings when `decimal.handling.mode` is `string`

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NUMERIC[(M[,D])]</code></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td><code>DECIMAL[(M[,D])]</code></td>
<td>STRING</td>
<td></td>
</tr>
</tbody>
</table>

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 3.16. Mappings for `HSTORE` data type

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HSTORE</code></td>
<td>STRING</td>
<td><code>io.debezium.data.Json</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: output representation using the JSON converter is <code>{&quot;key&quot;: &quot;val&quot;}</code></td>
</tr>
<tr>
<td><code>HSTORE</code></td>
<td>MAP</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: output representation using the JSON converter is <code>{&quot;key&quot; : &quot;val&quot;}</code></td>
</tr>
</tbody>
</table>

**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 3.17. Mappings for network address types

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INET</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPv4 and IPv6 networks</td>
</tr>
<tr>
<td>CIDR</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IPv4 and IPv6 hosts and networks</td>
</tr>
<tr>
<td>MACADDR</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC addresses</td>
</tr>
<tr>
<td>MACADDR8</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAC addresses in EUI-64 format</td>
</tr>
</tbody>
</table>

PostGIS types

The PostgreSQL connector supports all PostGIS data types.

Table 3.18. Mappings of PostGIS data types

<table>
<thead>
<tr>
<th>PostGIS data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostGIS data type</td>
<td>Literal type (schema type)</td>
<td>Semantic type (schema name) and Notes</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>GEOMETRY (planar)</td>
<td>STRUCT</td>
<td>io.debezium.data.geometry.Geometry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains a structure with two fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- srid (INT32) - Spatial Reference System Identifier that defines what type of geometry object is stored in the structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- wkb (BYTES) - A binary representation of the geometry object encoded in the Well-Known-Binary format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For format details, see Open Geospatial Consortium Simple Features Access specification.</td>
</tr>
<tr>
<td>GEOGRAPHY (spherical)</td>
<td>STRUCT</td>
<td>io.debezium.data.geometry.Geography</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains a structure with two fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- srid (INT32) - Spatial Reference System Identifier that defines what type of geography object is stored in the structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- wkb (BYTES) - A binary representation of the geometry object encoded in the Well-Known-Binary format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For format details, see Open Geospatial Consortium Simple Features Access specification.</td>
</tr>
</tbody>
</table>

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 link:https://www.postgresql.org/docs/current/storage-toast.html[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.
3.5. 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 3.5.1, “Configuring a replication slot for the Debezium pgoutput plug-in”
- Section 3.5.2, “Setting up PostgreSQL permissions for the Debezium connector”
- Section 3.5.3, “Setting privileges to enable Debezium to create PostgreSQL publications”
- Section 3.5.4, “Configuring PostgreSQL to allow replication with the Debezium connector host”
- Section 3.5.5, “Configuring PostgreSQL to manage Debezium WAL disk space consumption”

3.5.1. 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.

3.5.2. Setting up PostgreSQL permissions for the Debezium connector
Setting up a PostgreSQL server to run a Debezium connector requires a database user that can perform replications. Replication can be performed only by a database user that has appropriate permissions and only for a configured number of hosts.

Although, by default, superusers have the necessary **REPLICATION** and **LOGIN** roles, as mentioned in Security, it is best not to provide the Debezium replication user with elevated privileges. Instead, create a Debezium user that has the minimum required privileges.

**Prerequisites**
- PostgreSQL administrative permissions.

**Procedure**

1. To provide a user with replication permissions, define a PostgreSQL role that has at least the **REPLICATION** and **LOGIN** permissions, and then grant that role to the user. For example:

   ```sql
   CREATE ROLE <name> REPLICATION LOGIN;
   ```

**3.5.3. 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`.

3.5.4. Configuring PostgreSQL to allow replication with the Debezium connector host

To enable [prodname] 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,

```
local replication <youruser> trust
host replication <youruser> 127.0.0.1/32 trust
host replication <youruser> ::1/128 trust
```

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

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

3. Instructs the server to allow `<youruser>` on `localhost` to receive replication changes using `IPV6`.

**NOTE**

For more information about network masks, see the PostgreSQL documentation.

3.5.5. 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.

### 3.6. DEPLOYMENT OF DEBEZIUM POSTGRESQL 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:


### 3.6.1. 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 configures your Kafka Connector and that specifies the name of the image that you created to run your Debezium connector. You apply this CR to the OpenShift Kafka instance.

A KafkaConnector CR that configures your Debezium PostgreSQL connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed.

Prerequisites

- PostgreSQL is running and you performed the steps to set up PostgreSQL to run a Debezium connector.

- Red Hat AMQ Streams was used to set up and start running Apache Kafka and Kafka Connect on OpenShift. AMQ Streams offers operators and images that bring Kafka to 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-26-rhel7:1.6.0 as the base image. For example, from a terminal window, enter the following:
      ```
      cat <<EOF >debezium-container-for-postgresql.yaml
      FROM {DockerKafkaConnect}
      USER root:root
      COPY ./my-plugins/ /opt/kafka/plugins/
      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, run the following command:
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 any internal container registry. Ensure that this registry is reachable from your OpenShift instance. For example:

   ```bash
   podman push 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:

   ```yaml
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnect
   metadata:
     name: my-connect-cluster
     annotations: strimzi.io/use-connector-resources: "true"  
   spec:
     image: debezium-container-for-postgresql
   ```

   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:

   ```bash
   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 sets connector configuration properties. A 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. See the complete list of PostgreSQL connector properties that can be specified in these configurations.

   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.
The name of the connector.

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.

The connector’s configuration.

The name of the database host that is running the PostgreSQL server. In this example, the database host name is 192.168.99.100.

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.

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.

The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server. While the only supported value for PostgreSQL 10 and later is pgoutput, you must explicitly set plugin.name to pgoutput.

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 the initial snapshot has been executed. You should see something like this:
   
   ```
   ... 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.

### 3.6.2. 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 3.19. Required connector configuration properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Unique name for the connector. Attempting to register again with the same name will fail. This property is required by all Kafka Connect connectors.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>connector.class</strong></td>
<td></td>
<td>The name of the Java class for the connector. Always use a value of <code>io.debezium.connector.postgresql.PostgresConnector</code> for the PostgreSQL connector.</td>
</tr>
<tr>
<td><strong>tasks.max</strong></td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td><strong>plugin.name</strong></td>
<td>decoderbufs</td>
<td>The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server. The only supported value is <code>pgoutput</code>. You must explicitly set <code>plugin.name</code> to <code>pgoutput</code>.</td>
</tr>
<tr>
<td><strong>slot.name</strong></td>
<td>debezium</td>
<td>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: &quot;Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character.&quot;</td>
</tr>
<tr>
<td><strong>slot.drop.on.stop</strong></td>
<td>false</td>
<td>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.</td>
</tr>
</tbody>
</table>
The name of the PostgreSQL publication created for streaming changes when using `pgoutput`.

This publication is created at start-up if it does not already exist and it includes all tables. Debezium then applies its own include/exclude list filtering, if configured, to limit the publication to change events for the specific tables of interest. The connector user must have superuser permissions to create this publication, so it is usually preferable to create the publication before starting the connector for the first time.

If the publication already exists, either for all tables or configured with a subset of tables, Debezium uses the publication as it is defined.

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>publication.name</td>
<td><code>dbz_publication</code></td>
<td>The name of the PostgreSQL publication created for streaming changes when using <code>pgoutput</code>. This publication is created at start-up if it does not already exist and it includes all tables. Debezium then applies its own include/exclude list filtering, if configured, to limit the publication to change events for the specific tables of interest. The connector user must have superuser permissions to create this publication, so it is usually preferable to create the publication before starting the connector for the first time. If the publication already exists, either for all tables or configured with a subset of tables, Debezium uses the publication as it is defined.</td>
</tr>
<tr>
<td>database.hostname</td>
<td></td>
<td>IP address or hostname of the PostgreSQL database server.</td>
</tr>
<tr>
<td>database.port</td>
<td>5432</td>
<td>Integer port number of the PostgreSQL database server.</td>
</tr>
<tr>
<td>database.user</td>
<td></td>
<td>Name of the PostgreSQL database user for connecting to the PostgreSQL database server.</td>
</tr>
<tr>
<td>database.password</td>
<td></td>
<td>Password to use when connecting to the PostgreSQL database server.</td>
</tr>
<tr>
<td>database.dbname</td>
<td></td>
<td>The name of the PostgreSQL database from which to stream the changes.</td>
</tr>
<tr>
<td>database.server.name</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>schema.include.list</td>
<td></td>
<td>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 <code>schema.include.list</code> is excluded from having its changes captured. By default, all non-system schemas have their changes captured. Do not also set the <code>schema.exclude.list</code> property.</td>
</tr>
<tr>
<td>schema.exclude.list</td>
<td></td>
<td>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 <code>schema.exclude.list</code> has its changes captured, with the exception of system schemas. Do not also set the <code>schema.include.list</code> property.</td>
</tr>
<tr>
<td>table.include.list</td>
<td></td>
<td>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 <code>table.include.list</code> does not have its changes captured. Each identifier is of the form <code>schemaName$tableName</code>. By default, the connector captures changes in every non-system table in each schema whose changes are being captured. Do not also set the <code>table.exclude.list</code> property.</td>
</tr>
<tr>
<td>table.exclude.list</td>
<td></td>
<td>An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you do not want to capture. Any table not included in <code>table.exclude.list</code> has its changes captured. Each identifier is of the form <code>schemaName$tableName</code>. Do not also set the <code>table.include.list</code> property.</td>
</tr>
<tr>
<td>column.include.list</td>
<td></td>
<td>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 <code>schemaName$tableName$columnName</code>. Do not also set the <code>column.exclude.list</code> property.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>column.exclude.list</td>
<td></td>
<td>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 <code>schemaName.tableName.columnName</code>. Do not also set the <code>column.include.list</code> property.</td>
</tr>
<tr>
<td>time.precision.mode</td>
<td>adaptive</td>
<td>Time, date, and timestamps can be represented with different kinds of precision:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>adaptive</strong> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>adaptive_time_microseconds</strong> 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 <strong>TIME</strong> type fields, which are always captured as microseconds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>connect</strong> always represents time and timestamp values by using Kafka Connect’s built-in representations for <strong>Time</strong>, <strong>Date</strong>, and <strong>Timestamp</strong>, which use millisecond precision regardless of the database columns’ precision. See temporal values.</td>
</tr>
<tr>
<td>decimal.handling.mode</td>
<td>precise</td>
<td>Specifies how the connector should handle values for <strong>DECIMAL</strong> and <strong>NUMERIC</strong> columns:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>precise</strong> represents values by using <code>java.math.BigDecimal</code> to represent values in binary form in change events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>double</strong> represents values by using <strong>double</strong> values, which might result in a loss of precision but which is easier to use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>string</strong> encodes values as formatted strings, which are easy to consume but semantic information about the real type is lost. See Decimal types.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| hstore.handling.mode   | map     | Specifies how the connector should handle values for hstore columns:  
|                        |         | map represents values by using MAP.  
|                        |         | json represents values by using json string. This setting encodes values as formatted strings such as "key" : "val". See PostgreSQL HSTORE type. |
| interval.handling.mode | numeric | Specifies how the connector should handle values for interval columns:  
|                        |         | numeric represents intervals using approximate number of microseconds.  
|                        |         | string represents intervals exactly by using the string pattern representation  
|                        |         | P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S. For example: P1Y2M3DT4H5M6.78S. See PostgreSQL basic types. |
| database.sslmode       | disable | Whether to use an encrypted connection to the PostgreSQL server. Options include:  
|                        |         | disable uses an unencrypted connection.  
|                        |         | require uses a secure (encrypted) connection, and fails if one cannot be established.  
|                        |         | verify-ca behaves like require but also verifies the server TLS certificate against the configured Certificate Authority (CA) certificates, or fails if no valid matching CA certificates are found.  
<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. |
| 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. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>database.sslpassword</code></td>
<td></td>
<td>The password to access the client private key from the file specified by <code>database.ssslkey</code>. See the PostgreSQL documentation for more information.</td>
</tr>
<tr>
<td><code>database.ssslrootcert</code></td>
<td></td>
<td>The path to the file that contains the root certificate(s) against which the server is validated. See the PostgreSQL documentation for more information.</td>
</tr>
<tr>
<td><code>database.tcpKeepAlive</code></td>
<td><code>true</code></td>
<td>Enable TCP keep-alive probe to verify that the database connection is still alive. See the PostgreSQL documentation for more information.</td>
</tr>
<tr>
<td><code>tombstones.on.delete</code></td>
<td><code>true</code></td>
<td>Controls whether a tombstone event should be generated after a <code>delete</code> event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>true</strong> - delete operations are represented by a <code>delete</code> event and a subsequent tombstone event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>false</strong> - only a <code>delete</code> event is sent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>After a <code>delete</code> operation, emitting a tombstone event enables Kafka to delete all change event records that have the same key as the deleted row.</td>
</tr>
<tr>
<td><code>column.truncate.to_length.chars</code></td>
<td><code>n/a</code></td>
<td>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 <code>schemaName.tableName.columnName</code>. In change event records, values in these columns are truncated if they are longer than the number of characters specified by <code>length</code> in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, <code>+column.truncate.to.20.chars</code>.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>column.mask.with.<em>length</em>.chars</td>
<td>n/a</td>
<td>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 <code>schemaName.tableName.columnName</code>. In change event values, the values in the specified table columns are replaced with length number of asterisk (*) characters. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer or zero. When you specify zero, the connector replaces a value with an empty string.</td>
</tr>
<tr>
<td>column.mask.hash.<em>hash.hashAlgorithm</em>.with.salt_.salt</td>
<td>n/a</td>
<td>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 <code>schemaName.tableName.columnName</code>. In change event values, the values in the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specified <code>hashAlgorithm</code> and <code>salt</code>. Based on the hash function that is used, referential integrity is kept 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. If necessary, the pseudonym is automatically shortened to the length of the column. You can specify multiple properties with different hash algorithms and salts in a single configuration. In the following example, <code>CzQMA0cB5K</code> is a randomly selected salt.</td>
</tr>
<tr>
<td>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K</td>
<td></td>
<td>=inventory.orders.customerName,inventory.shipment.customerName</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Depending on the <code>hashAlgorithm</code> used, the <code>salt</code> selected, and the actual data set, the resulting masked data set might not be completely masked.</td>
</tr>
</tbody>
</table>
### column.propagate.source.type

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n/a</td>
<td>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 <code>databaseName.tableName.columnName</code>, or <code>databaseName.schemaName.tableName.columnName</code>. 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This property is useful for properly sizing corresponding columns in sink databases.</td>
</tr>
</tbody>
</table>

### datatype.propagate.source.type

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n/a</td>
<td>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 <code>databaseName.tableName.typeName</code>, or <code>databaseName.schemaName.tableName.typeName</code>. 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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. This property is useful for properly sizing corresponding columns in sink databases.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See the list of PostgreSQL-specific data type names.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| message.key.columns   | empty string  | 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. Separate entries with semicolons. Insert a colon between the fully-qualified table name and its regular expression. The format is:  

  `schema-name.table-name:_regexp_;...`  

For example,  

  `schemaA.table_a:regex_1;schemaB.table_b:regex_2;schemaC.table_c:regex_3`  

If `table_a` has a `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. |
### publication.autocreate.mode

- **Default:** `all_tables`

Applies only when streaming changes by using the **pgoutput** plug-in. The setting determines how creation of a publication should work. Possible settings are:

- **all_tables** - If a publication exists, the connector uses it. If a publication does not exist, the connector creates a publication for all tables in the database for which the connector is capturing changes. This requires that the database user that has permission to perform replications also has permission to create a publication. This is granted with `CREATE PUBLICATION <publication_name> FOR ALL TABLES;`.

- **disabled** - The connector does not attempt to create a publication. A database administrator or the user configured to perform replications must have created the publication before running the connector. If the connector cannot find the publication, the connector throws an exception and stops.

- **filtered** - If a publication exists, the connector uses it. If no publication exists, the connector creates a new publication for tables that match the current filter configuration as specified by the `database.exclude.list`, `schema.include.list`, `schema.exclude.list`, and `table.include.list` connector configuration properties. For example: `CREATE PUBLICATION <publication_name> FOR TABLE <tbl1, tbl2, tbl3>`.

### binary.handling.mode

- **Default:** `bytes`

Specifies how binary (bytea) columns should be represented in change events:

- **bytes** represents binary data as byte array.

- **base64** represents binary data as base64-encoded strings.

- **hex** represents binary data as hex-encoded (base16) strings.
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 3.20. Advanced connector configuration properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>truncate.handling.mode</code></td>
<td>bytes</td>
<td>Specifies how whether TRUNCATE events should be propagated or not (only available when using the <code>pgoutput</code> plug-in with Postgres 11 or later): skip causes those event to be omitted (the default). include causes hos events to be included. + Please see <code>truncate events</code> for the structure of <code>truncate</code> events and their ordering semantics.</td>
</tr>
</tbody>
</table>

+ Please see `truncate events` for the structure of `truncate` events and their ordering semantics.
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snapshot.mode</td>
<td>initial</td>
<td>Specifies the criteria for performing a snapshot when the connector starts:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>initial</strong> - The connector performs a snapshot only when no offsets have been recorded for the logical server name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>always</strong> - The connector performs a snapshot each time the connector starts.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>never</strong> - 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 <strong>never</strong> snapshot mode is useful only when you know all data of interest is still reflected in the WAL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>initial_only</strong> - The connector performs an initial snapshot and then stops, without processing any subsequent changes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>exported</strong> - 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.</td>
</tr>
<tr>
<td>snapshot.include.collection.list</td>
<td>All tables specified in table.include.list</td>
<td>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</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>snapshot.lock.timeout.ms</td>
<td>10000</td>
<td>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.</td>
</tr>
<tr>
<td>snapshot.select.statenameoverrides</td>
<td></td>
<td>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 <code>databaseName.tableName</code>. For each table that you specify, also specify another configuration property: <code>snapshot.select.statenameoverrides.DB_NAME.TABLE_NAME</code>, for example: <code>snapshot.select.statenameoverrides.customers.orders</code>. Set this property to a <code>SELECT</code> statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this <code>SELECT</code> statement to retrieve data from that table. A possible use case for setting these properties is large, append-only tables. You can specify a <code>SELECT</code> statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.</td>
</tr>
<tr>
<td>event.processing.failure.handling.mode</td>
<td>fail</td>
<td>Specifies how the connector should react to exceptions during processing of events: <code>fail</code> propagates the exception, indicates the offset of the problematic event, and causes the connector to stop. <code>warn</code> logs the offset of the problematic event, skips that event, and continues processing. <code>skip</code> skips the problematic event and continues processing.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>max.queue.size</td>
<td>20240</td>
<td>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 that it should be or Kafka is not available.</td>
</tr>
<tr>
<td>max.batch.size</td>
<td>10240</td>
<td>Positive integer value that specifies the maximum size of each batch of events that the connector processes.</td>
</tr>
<tr>
<td>max.queue.size.in.bytes</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>poll.interval.ms</td>
<td>1000</td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>include.unknown.datatypes</td>
<td>false</td>
<td>Specifies connector behavior when the connector encounters a field whose data type is unknown. The default behavior is that the connector omits the field from the change event and logs a warning. Set this property to <strong>true</strong> 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 <strong>binary handling mode</strong> property.</td>
</tr>
</tbody>
</table>

**NOTE**

Consumers risk backward compatibility issues when **include.unknown.datatypes** is set to **true**. Not only may the database-specific binary representation change between releases, but if the data type is eventually supported by Debezium, the data type will be sent downstream in a logical type, which would require adjustments by consumers. In general, when encountering unsupported data types, create a feature request so that support can be added.
**database.initialstatements**

A semicolon separated list of SQL statements that the connector executes when it establishes a JDBC connection to the database. To use a semicolon as a character and not as a delimiter, specify two consecutive semicolons, `;`.

The connector may establish JDBC connections at its own discretion. Consequently, this property is useful for configuration of session parameters only, and not for executing DML statements.

The connector does not execute these statements when it creates a connection for reading the transaction log.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>database.initialstatements</strong></td>
<td></td>
<td>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, <code>;</code>. The connector may establish JDBC connections at its own discretion. Consequently, this property is useful for configuration of session parameters only, and not for executing DML statements. The connector does not execute these statements when it creates a connection for reading the transaction log.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>heartbeat.interval.ms</td>
<td>0</td>
<td>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are needed when there are many updates in a database that is being tracked but only a tiny number of updates are related to the table(s) and schema(s) for which the connector is capturing changes. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that no offset updates are committed to Kafka and the connector does not have an opportunity to send the latest retrieved LSN to the database. The database retains WAL files that contain events that have already been processed by the connector. Sending heartbeat messages enables the connector to send the latest retrieved LSN to the database, which allows the database to reclaim disk space being used by no longer needed WAL files.</td>
</tr>
</tbody>
</table>
| heartbeat.topics.prefix | __debezium-heartbeat | Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern: 

<heartbeat.topics.prefix>.<server.name> 

For example, if the database server name is fulfillment, the default topic name is __debezium-heartbeat.fulfillment. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heartbeat.action.query</td>
<td></td>
<td>Specifies a query that the connector executes on the source database when the connector sends a heartbeat message.</td>
</tr>
</tbody>
</table>
|                          |               | This is useful for resolving the situation described in WAL disk space consumption, where capturing changes from a low-traffic database on the same host as a high-traffic database prevents Debezium from processing WAL records and thus acknowledging WAL positions with the database. To address this situation, create a heartbeat table in the low-traffic database, and set this property to a statement that inserts records into that table, for example:  
|                          |               | **INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')**                                                                                                                                 |
|                          |               | This allows the connector to receive changes from the low-traffic database and acknowledge their LSNs, which prevents unbounded WAL growth on the database host.                                              |
| schema.refresh.mode      | columns_diff | Specify the conditions that trigger a refresh of the in-memory schema for a table. Adamantly, **columns_diff** is the safest mode. It ensures that the in-memory schema stays in sync with the database table’s schema at all times. |
|                          |               | **columns_diff** exclude unchanged TOAST instructs the connector to refresh the in-memory schema cache if there is a discrepancy with the schema derived from the incoming message, unless unchanged TOASTable data fully accounts for the discrepancy. This setting can significantly improve connector performance if there are frequently-updated tables that have TOASTed data that are rarely part of updates. However, it is possible for the in-memory schema to become outdated if TOASTable columns are dropped from the table. |


<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snapshot.delay.ms</td>
<td></td>
<td>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 rebalancing of connectors.</td>
</tr>
<tr>
<td>snapshot.fetch.size</td>
<td>10240</td>
<td>During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.</td>
</tr>
<tr>
<td>slot.stream.params</td>
<td></td>
<td>Semicolon separated list of parameters to pass to the configured logical decoding plug-in. For example, <code>add-tables=public.table,public.table2;include-lsn=true</code>.</td>
</tr>
<tr>
<td>sanitize.field.names</td>
<td>true if connector configuration sets the <code>key.converter</code> or <code>value.converter</code> property to the Avro converter. false if not.</td>
<td>Indicates whether field names are sanitized to adhere to Avro naming requirements.</td>
</tr>
<tr>
<td>slot.max.retries</td>
<td>6</td>
<td>If connecting to a replication slot fails, this is the maximum number of consecutive attempts to connect.</td>
</tr>
<tr>
<td>slot.retry.delay.ms</td>
<td>10000 (10 seconds)</td>
<td>The number of milliseconds to wait between retry attempts when the connector fails to connect to a replication slot.</td>
</tr>
<tr>
<td>toasted.value.placeholder</td>
<td>__debezium_unavailable_value</td>
<td>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 <code>toasted.value.placeholder</code> starts with the <code>hex:</code> prefix it is expected that the rest of the string represents hexadecimally encoded octets. See toasted values for additional details.</td>
</tr>
</tbody>
</table>
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.

### 3.7. 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.

#### 3.7.1. Monitoring Debezium during snapshots of PostgreSQL databases

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last snapshot event that the connector has read.</td>
</tr>
<tr>
<td>MillisecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>TotalTableCount</td>
<td>int</td>
<td>The total number of tables that are being included in the snapshot.</td>
</tr>
<tr>
<td>RemainingTableCount</td>
<td>int</td>
<td>The number of tables that the snapshot has yet to copy.</td>
</tr>
<tr>
<td>SnapshotRunning</td>
<td>boolean</td>
<td>Whether the snapshot was started.</td>
</tr>
<tr>
<td>SnapshotAborted</td>
<td>boolean</td>
<td>Whether the snapshot was aborted.</td>
</tr>
<tr>
<td>SnapshotCompleted</td>
<td>boolean</td>
<td>Whether the snapshot completed.</td>
</tr>
<tr>
<td>SnapshotDurationInSeconds</td>
<td>long</td>
<td>The total number of seconds that the snapshot has taken so far, even if not complete.</td>
</tr>
</tbody>
</table>
### Attributes

| **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 PostgreSQL connector record streaming

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

<table>
<thead>
<tr>
<th><strong>Attributes</strong></th>
<th><strong>Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LastEvent</strong></td>
<td><strong>string</strong></td>
<td>The last streaming event that the connector has read.</td>
</tr>
<tr>
<td><strong>MilliSecondsSinceLastEvent</strong></td>
<td><strong>long</strong></td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td><strong>TotalNumberOfEventsSeen</strong></td>
<td><strong>long</strong></td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td><strong>NumberOfEventsFiltered</strong></td>
<td><strong>long</strong></td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td><strong>MonitoredTables</strong></td>
<td><strong>string[]</strong></td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>Connected</td>
<td>boolean</td>
<td>Flag that denotes whether the connector is currently connected to the database server.</td>
</tr>
<tr>
<td>MilliSecondsBehindSource</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfCommittedTransactions</td>
<td>long</td>
<td>The number of processed transactions that were committed.</td>
</tr>
<tr>
<td>SourceEventPosition</td>
<td>Map&lt;String, String&gt;</td>
<td>The coordinates of the last received event.</td>
</tr>
<tr>
<td>LastTransactionId</td>
<td>string</td>
<td>Transaction identifier of the last processed transaction.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>The maximum buffer of the queue in bytes.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

### 3.8. 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 \textit{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

\section*{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.

\section*{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.

\section*{Cluster failures}

As of release 12, PostgreSQL allows logical replication slots \textit{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 \texttt{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 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.

4.1. OVERVIEW

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 a 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 the 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
4.2. SETTING UP MONGODB

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.

4.3. SUPPORTED MONGODB TOPOLOGIES

The MongoDB connector can be used with a variety of MongoDB topologies.

4.3.1. MongoDB replica set

The 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.

4.3.2. 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.

### 4.3.3. 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.

### 4.4. HOW THE MONGODB CONNECTOR WORKS

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.

### 4.4.1. Logical connector name

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.

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

### 4.4.2. Performing a snapshot
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.

4.4.3. Streaming changes

Once the connector task for a replica set has an offset, it uses the offset to determine the position in the oplog where it should start streaming changes. The task will then connect to the replica set’s primary node and start streaming changes from that position, processing all of the create, insert, and delete operations and converting them into Debezium change events. Each change event includes the position in the oplog where the operation was found, and the connector periodically records this as its most recent offset. The interval at which the offset is recorded is governed by offset.flush.interval.ms, which is a Kafka Connect worker configuration property.

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

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

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

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

### 4.4.4. Topics names

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.

### 4.4.5. Partitions

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

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.

### 4.4.6. 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:

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

### Table 4.1. Overview of change event basic content

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>payload</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td>payload</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
4.4.6.1. Change event keys

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

```json
{
  "_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:

```json
{
  "schema": {
    "type": "struct",
    "name": "fulfillment.inventory.customers.Key",
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "string",
        "optional": false
      }
    ]
  }
}
```
Table 4.2. Description of change event key

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s payload portion.</td>
</tr>
</tbody>
</table>
| 2    | fulfillment.inventory.customers. Key | Name of the schema that defines the structure of the key’s payload. This schema describes the structure of the key for the document that was changed. Key schema names have the format `connector-name.database-name.collection-name`. In this example:  
  - `fulfillment` is the name of the connector that generated this event.  
  - `inventory` is the database that contains the collection that was changed.  
  - `customers` is the collection that contains the document that was updated. |
| 3    | optional   | Indicates whether the event key must contain a value in its payload field. In this example, a value in the key’s payload is required. A value in the key’s payload field is optional when a document does not have a key. |
| 4    | fields     | Specifies each field that is expected in the payload, including each field’s name, type, and whether it is required. |
| 5    | payload    | Contains the key for the document for which this change event was generated. In this example, the key contains a single `id` field of type `string` whose value is `1004`. |

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

Table 4.3. Examples of representing document _id fields in event key payloads

<table>
<thead>
<tr>
<th>Type</th>
<th>MongoDB _id Value</th>
<th>Key’s payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1234</td>
<td>{ &quot;id&quot; : &quot;1234&quot; }</td>
</tr>
<tr>
<td>Float</td>
<td>12.34</td>
<td>{ &quot;id&quot; : &quot;12.34&quot; }</td>
</tr>
</tbody>
</table>
### 4.4.6.2. Change event values

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**

```json
{
    "_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**

#### 4.4.6.3. 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:

```json
{
    "schema": {
        "type": "struct",
        "fields": [
```
CHAPTER 4. DEBEZIUM CONNECTOR FOR MONGODB

```json
{
  "type": "string",
  "optional": true,
  "name": "io.debezium.data.Json",
  "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": "collection"
    }
  ]
}
```
"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",
"field": "source"
},
{
"type": "string",
"optional": true,
"field": "op"
},
{
"type": "int64",
"optional": true,
"field": "ts_ms"
}
],
"optional": false,
"name": "dbserver1.inventory.customers.Envelope",
"payload": {
"after": {
"_id" : {"$numberLong" : "1004"},
"first_name" : "Anne",
"last_name" : "Kretchmar",
"email" : "annek@noanswer.org"}
},
"patch": null,
"source": {
"version": "1.4.2.Final",
"connector": "mongodb",
"name": "fulfillment",
"ts_ms": 1558965508000,
"snapshot": false,
"db": "inventory",
"rs": "rs0",
"collection": "customers",
"ord": 31,
"h": 1546547425148721999
},
"op": "c"}
Table 4.4. Descriptions of create event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>In the schema section, each name field specifies the schema for a field in the value’s payload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>io.debezium.data.Json is the schema for the payload’s after, patch, and filter fields. This schema is specific to the customers collection. A create event is the only kind of event that contains an after field. An update event contains a filter field and a patch field. A delete event contains a filter field, but not an after field nor a patch field.</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td>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.</td>
</tr>
<tr>
<td>5</td>
<td>payload</td>
<td>The value’s actual data. This is the information that the change event is providing. 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.</td>
</tr>
<tr>
<td>6</td>
<td>after</td>
<td>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 <em>create</em> events; in other words, a create event is the only kind of event that contains an after field.</td>
</tr>
</tbody>
</table>
7 source

Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes:

- Debezium version.
- Name of the connector that generated the event.
- Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes.
- Names of the collection and database that contain the new document.
- If the event was part of a snapshot.
- Timestamp for when the change was made in the database and ordinal of the event within the timestamp.
- Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the $h$ field in the oplog event, or a field named $stxnid$, which represents the $lsid$ and $txnNumber$ fields from the oplog event.

8 op

Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, $c$ indicates that the operation created a document. Valid values are:

- $c$ = create
- $u$ = update
- $d$ = delete
- $r$ = read (applies to only snapshots)

9 ts_ms

Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.

In the source object, ts_ms indicates the time that the change was made in the database. By comparing the value for payload.source.ts_ms with the value for payload.ts_ms, you can determine the lag between the source database update and Debezium.

### 4.4.6.4. 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:

```json
{
  "schema": { ... },
  "payload": {
    "op": "u",  
    "ts_ms": 1465491461815,  
    "patch": "{"$set":{"first_name":"Anne Marie"}}",
    "filter": "{"_id" : {"$numberLong" : "1004"}}",
    "source": {
      "version": "1.4.2.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

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>op</td>
<td>Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, <strong>u</strong> indicates that the operation updated a document.</td>
</tr>
<tr>
<td>2</td>
<td>ts_ms</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In the <strong>source</strong> object, <strong>ts_ms</strong> indicates the time that the change was made in the database. By comparing the value for <strong>payload.source.ts_ms</strong> with the value for <strong>payload.ts_ms</strong>, you can determine the lag between the source database update and Debezium.</td>
</tr>
<tr>
<td>3</td>
<td>patch</td>
<td>Contains the JSON string representation of the actual MongoDB idempotent change to the document. In this example, the update changed the <strong>first_name</strong> field to a new value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An update event value does not contain an <strong>after</strong> field.</td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>4</td>
<td>filter</td>
<td>Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be updated.</td>
</tr>
<tr>
<td>5</td>
<td>source</td>
<td>Mandatory field that describes the source metadata for the event. This field contains the same information as a create event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debezium version.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name of the connector that generated the event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Names of the collection and database that contain the updated document.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the event was part of a snapshot.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Timestamp for when the change was made in the database and ordinal of the event within the timestamp.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
</tbody>
</table>

**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.

### 4.4.6.5. 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": "1.4.2.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

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>op</td>
<td>Mandatory string that describes the type of operation. The op field value is d, signifying that this document was deleted.</td>
</tr>
<tr>
<td>2</td>
<td>ts_ms</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>filter</td>
<td>Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be deleted.</td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>4</td>
<td>source</td>
<td>Mandatory field that describes the source metadata for the event. This field contains the same information as a create or update event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debezium version.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name of the connector that generated the event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Names of the collection and database that contained the deleted document.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the event was part of a snapshot.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Timestamp for when the change was made in the database and ordinal of the event within the timestamp.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 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.</td>
</tr>
</tbody>
</table>

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.

#### 4.4.7. Transaction Metadata

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

##### 4.4.7.1. Transaction boundaries

Debezium generates events for every transaction **BEGIN** and **END**. Every event contains

- **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

Following is an example of what a message looks like:

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

{
  "status": "END",
  "id": "1462833718356672513",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "rs0.testDB.tablea",
      "event_count": 1
    },
    {
      "data_collection": "rs0.testDB.tableb",
      "event_count": 1
    }
  ]
}

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

### 4.4.7.2. Data events 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:

```json
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
    ...
  },
  "op": "c",
}
```
4.5. DEPLOYING THE MONGODB CONNECTOR

To deploy a Debezium MongoDB connector, install the Debezium MongoDB connector archive, configure the connector, and start the connector by adding its configuration to Kafka Connect.

To install the MongoDB connector, follow the procedures in Installing Debezium on OpenShift. The main steps are:

1. Use Red Hat AMQ Streams to set up Apache Kafka and Kafka Connect on OpenShift. AMQ Streams offers operators and images that bring Kafka to OpenShift.
2. Download the Debezium MongoDB connector.
3. Extract the connector files into your Kafka Connect environment.
4. Add the connector plug-in's parent directory to your Kafka Connect plugin.path, for example:

   ```
   plugin.path=/kafka/connect
   ```

   The above example assumes that you extracted the Debezium MongoDB connector to the /kafka/connect/Debezium-connector-mongodb path.
5. Restart your Kafka Connect process to ensure that the new JAR files are picked up.

You also need to set up MongoDB to run a Debezium connector.

**Additional resources**

For more information about the deployment process, and deploying connectors with AMQ Streams, see the Debezium installation guides.

- Installing Debezium on OpenShift
- Installing Debezium on RHEL

4.5.1. Example configuration

To use the connector to produce change events for a particular MongoDB replica set or sharded cluster, create a configuration file in JSON. When the connector starts, it will perform a snapshot of the collections in your MongoDB replica sets and start reading the replica sets’ oplogs, producing events for every inserted, updated, and deleted document. Optionally filter out collections that are not needed.

Following is an example of the configuration for a MongoDB connector that monitors a MongoDB replica set rs0 at port 27017 on 192.168.99.100, which we logically name `fulfillment`. Typically, you configure the Debezium MongoDB connector in a .yaml file using the configuration properties available for the connector.
The name of our connector when we register it with a Kafka Connect service.

The name of the MongoDB connector class.

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

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

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

See the complete list of connector properties that can be specified in these configurations.

This configuration can be sent via POST to a running Kafka Connect service, which will then record the configuration and start up the one connector task that will connect to the MongoDB replica set or sharded cluster, assign tasks for each replica set, perform a snapshot if necessary, read the oplog, and record events to Kafka topics.

### 4.5.2. Adding connector configuration

You can use a provided Debezium container to deploy a Debezium MongoDB connector. In this procedure, you build a custom Kafka Connect container image for Debezium, configure the Debezium connector as needed, and then add your connector configuration to your Kafka Connect environment.

#### Prerequisites

- Podman or Docker is installed and you have sufficient rights to create and manage containers.
- You installed the Debezium MongoDB connector archive.

#### Procedure

1. Extract the Debezium MongoDB connector archive to create a directory structure for the connector plug-in, for example:

   ```
   tree ./my-plugins/
   ./my-plugins/
   ├── debezium-connector-mongodb
   │    ├── ...
   ```
2. Create and publish a custom image for running your Debezium connector:

   a. Create a new Dockerfile by using registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0 as the base image. In the following example, you would replace my-plugins with the name of your plug-ins directory:

   ```
   FROM registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0
   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 container image. For example, if you saved the Dockerfile that you created in the previous step as debezium-container-for-mongodb, and if the Dockerfile is in the current directory, then you would run the following command:

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

   c. Push your custom image to your container registry, for example:

   ```
   podman push debezium-container-for-mongodb:latest
   ```

   d. Point to the new container image. Do one of the following:

   - Edit the spec.image property of the KafkaConnector custom resource. If set, this property overrides the STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE variable in the Cluster Operator. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnector
   metadata:
     name: my-connect-cluster
   spec:
     #...
     image: debezium-container-for-mongodb
   ```

   - 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 must apply it to your OpenShift cluster.

3. Create a KafkaConnector custom resource that defines your Debezium MongoDB connector instance. See the connector configuration example.

4. 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.

5. Verify that the connector was created and has started to capture 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:

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

Results
When the connector starts, it performs a consistent snapshot of the MongoDB databases that the connector is configured for. The connector then starts generating data change events for document-level operations and streaming change event records to Kafka topics.

4.5.3. Monitoring
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; for monitoring the connector when performing snapshots
- streaming metrics; for monitoring the connector when processing oplog events

Please refer to the monitoring documentation for details of how to expose these metrics via JMX.

4.5.3.1. Snapshot Metrics
The MBean is debezium.mongodb:type=connector-metrics,context=snapshot,server=<mongodb.name>.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last snapshot event that the connector has read.</td>
</tr>
<tr>
<td>MillisecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Type</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>QueueTotalCapacity</code></td>
<td><code>int</code></td>
<td>The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td><code>QueueRemainingCapacity</code></td>
<td><code>int</code></td>
<td>The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td><code>TotalTableCount</code></td>
<td><code>int</code></td>
<td>The total number of tables that are being included in the snapshot.</td>
</tr>
<tr>
<td><code>RemainingTableCount</code></td>
<td><code>int</code></td>
<td>The number of tables that the snapshot has yet to copy.</td>
</tr>
<tr>
<td><code>SnapshotRunning</code></td>
<td><code>boolean</code></td>
<td>Whether the snapshot was started.</td>
</tr>
<tr>
<td><code>SnapshotAborted</code></td>
<td><code>boolean</code></td>
<td>Whether the snapshot was aborted.</td>
</tr>
<tr>
<td><code>SnapshotCompleted</code></td>
<td><code>boolean</code></td>
<td>Whether the snapshot completed.</td>
</tr>
<tr>
<td><code>SnapshotDurationInSeconds</code></td>
<td><code>long</code></td>
<td>The total number of seconds that the snapshot has taken so far, even if not complete.</td>
</tr>
<tr>
<td><code>RowsScanned</code></td>
<td><code>Map&lt;String, Long&gt;</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>MaxQueueSizeInBytes</code></td>
<td><code>long</code></td>
<td>The maximum buffer of the queue in bytes. It will be enabled if <code>max.queue.size.in.bytes</code> is passed with a positive long value.</td>
</tr>
<tr>
<td><code>CurrentQueueSizeInBytes</code></td>
<td><code>long</code></td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>
The Debezium MongoDB connector also provides the following custom snapshot metrics:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfDisconnects</td>
<td>long</td>
<td>Number of database disconnects.</td>
</tr>
</tbody>
</table>

### 4.5.3.2. Streaming Metrics

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last streaming event that the connector has read.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>Connected</td>
<td>boolean</td>
<td>Flag that denotes whether the connector is currently connected to the database server.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MilliSecondsBehindSource</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfCommittedTransactions</td>
<td>long</td>
<td>The number of processed transactions that were committed.</td>
</tr>
<tr>
<td>SourceEventPosition</td>
<td>Map&lt;String, String&gt;</td>
<td>The coordinates of the last received event.</td>
</tr>
<tr>
<td>LastTransactionId</td>
<td>string</td>
<td>Transaction identifier of the last processed transaction.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>The maximum buffer of the queue in bytes.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfDisconnects</td>
<td>long</td>
<td>Number of database disconnects.</td>
</tr>
<tr>
<td>NumberOfPrimaryElections</td>
<td>long</td>
<td>Number of primary node elections.</td>
</tr>
</tbody>
</table>

4.5.4. Connector properties

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>connector.class</td>
<td></td>
<td>The name of the Java class for the connector. Always use a value of <code>io.debezium.connector.mongodb.MongoDbConnector</code> for the MongoDB connector.</td>
</tr>
<tr>
<td>mongodb.hosts</td>
<td></td>
<td>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 <code>mongodb.members.auto.discover</code> is set to <code>false</code>, then the host and port pair should be prefixed with the replica set name (e.g., <code>rs0/localhost:27017</code>).</td>
</tr>
<tr>
<td>mongodb.name</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>mongodb.user</td>
<td></td>
<td>Name of the database user to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.</td>
</tr>
<tr>
<td>mongodb.password</td>
<td></td>
<td>Password to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.</td>
</tr>
<tr>
<td>mongodb.authsource</td>
<td>admin</td>
<td>Database (authentication source) containing MongoDB credentials. This is required only when MongoDB is configured to use authentication with another authentication database than <code>admin</code>.</td>
</tr>
<tr>
<td>mongodb.ssl.enabled</td>
<td>false</td>
<td>Connector will use SSL to connect to MongoDB instances.</td>
</tr>
<tr>
<td>mongodb.ssl.invalid.host.name.allowed</td>
<td>false</td>
<td>When SSL is enabled this setting controls whether strict hostname checking is disabled during connection phase. If <code>true</code> the connection will not prevent man-in-the-middle attacks.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>database.include.list</td>
<td>empty string</td>
<td>An optional comma-separated list of regular expressions that match database names to be monitored; any database name not included in <code>database.include.list</code> is excluded from monitoring. By default all databases are monitored. Must not be used with <code>database.exclude.list</code>.</td>
</tr>
<tr>
<td>database.exclude.list</td>
<td>empty string</td>
<td>An optional comma-separated list of regular expressions that match database names to be excluded from monitoring; any database name not included in <code>database.exclude.list</code> is monitored. Must not be used with <code>database.include.list</code>.</td>
</tr>
<tr>
<td>collection.include.list</td>
<td>empty string</td>
<td>An optional comma-separated list of regular expressions that match fully-qualified namespaces for MongoDB collections to be monitored; any collection not included in <code>collection.include.list</code> is excluded from monitoring. Each identifier is of the form <code>databaseName.collectionName</code>. By default the connector will monitor all collections except those in the <code>local</code> and <code>admin</code> databases. Must not be used with <code>collection.exclude.list</code>.</td>
</tr>
<tr>
<td>collection.exclude.list</td>
<td>empty string</td>
<td>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 <code>collection.exclude.list</code> is monitored. Each identifier is of the form <code>databaseName.collectionName</code>. Must not be used with <code>collection.include.list</code>.</td>
</tr>
<tr>
<td>snapshot.mode</td>
<td>initial</td>
<td>Specifies the criteria for running a snapshot upon startup of the connector. The default is <code>initial</code>, and specifies the connector reads a snapshot when either no offset is found or if the oplog no longer contains the previous offset. The <code>never</code> option specifies that the connector should never use snapshots, instead the connector should proceed to tail the log.</td>
</tr>
<tr>
<td>snapshot.include.collection.list</td>
<td>All collections specified in <code>collection.include.list</code></td>
<td>An optional, comma-separated list of regular expressions that match names of schemas specified in <code>collection.include.list</code> for which you want to take the snapshot.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>field.exclude.list</td>
<td>empty string</td>
<td>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 <code>databaseName.collectionName.fieldName.nestedFieldNam</code>, where <code>databaseName</code> and <code>collectionName</code> may contain the wildcard (*) which matches any characters.</td>
</tr>
<tr>
<td>field.renames</td>
<td>empty string</td>
<td>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 <code>databaseName.collectionName.fieldName.nestedFieldNam: newFieldName</code>, where <code>databaseName</code> and <code>collectionName</code> 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.</td>
</tr>
<tr>
<td>tasks.max</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>snapshot.max.threads</td>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tombstones.on.delete</strong></td>
<td><strong>true</strong></td>
<td>Controls whether a tombstone event should be generated after a delete event. When <strong>true</strong> the delete operations are represented by a delete event and a subsequent tombstone event. When <strong>false</strong> only a delete event is sent. Emitting the tombstone event (the default behavior) allows Kafka to completely delete all events pertaining to the given key once the source record got deleted.</td>
</tr>
<tr>
<td><strong>snapshot.delay.ms</strong></td>
<td><strong>0</strong></td>
<td>An interval in milliseconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.</td>
</tr>
<tr>
<td><strong>snapshot.fetch.size</strong></td>
<td><strong>0</strong></td>
<td>Specifies the maximum number of documents that should be read in one go from each collection while taking a snapshot. The connector will read the collection contents in multiple batches of this size. Defaults to <strong>0</strong>, which indicates that the server chooses an appropriate fetch size.</td>
</tr>
<tr>
<td><strong>max.queue.size</strong></td>
<td><strong>8192</strong></td>
<td>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 <strong>8192</strong>, and should always be larger than the maximum batch size specified in the <strong>max.batch.size</strong> property.</td>
</tr>
<tr>
<td><strong>max.batch.size</strong></td>
<td><strong>2048</strong></td>
<td>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 <strong>2048</strong>.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>max.queue.size.in.bytes</code></td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td><code>poll.interval.ms</code></td>
<td>1000</td>
<td>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.</td>
</tr>
<tr>
<td><code>connect.backoff.initial.delay.ms</code></td>
<td>1000</td>
<td>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).</td>
</tr>
<tr>
<td><code>connect.backoff.max.delay.ms</code></td>
<td>1000</td>
<td>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).</td>
</tr>
<tr>
<td><code>connect.max.attempts</code></td>
<td>16</td>
<td>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.</td>
</tr>
<tr>
<td><code>mongodb.members.auto.discover</code></td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>heartbeat.interval.ms</td>
<td>0</td>
<td>Controls how frequently heartbeat messages are sent. This property contains an interval in milliseconds that defines how frequently the connector sends messages into a heartbeat topic. This can be used to monitor whether the connector is still receiving change events from the database. You also should leverage heartbeat messages in cases where only records in non-captured collections are changed for a longer period of time. In such situation the connector would proceed to read the oplog from the database but never emit any change messages into Kafka, which in turn means that no offset updates are committed to Kafka. This will cause the oplog files to be rotated out but connector will not notice it so on restart some events are no longer available which leads to the need of re-execution of the initial snapshot. Set this parameter to 0 to not send heartbeat messages at all. Disabled by default.</td>
</tr>
<tr>
<td>heartbeat.topics.prefix</td>
<td>__debezium-heartbeat</td>
<td>Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern <code>&lt;heartbeat.topics.prefix&gt;.&lt;server.name&gt;</code>.</td>
</tr>
<tr>
<td>sanitize.field.names</td>
<td>true</td>
<td>Whether field names are sanitized to adhere to Avro naming requirements. When connector configuration explicitly specifies the <code>key.converter</code> or <code>value.converter</code> parameters to use Avro, otherwise defaults to false.</td>
</tr>
<tr>
<td>skipped.operations</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>snapshot.collection.filter.overrides</td>
<td></td>
<td>Controls which collection items are included in snapshot. This property affects snapshots only. Specify a comma-separated list of collection names in the form databaseName.collectionName. For each collection that you specify, also specify another configuration property: snapshot.collection.filter.overrides.dataBaseName.collectionName. For example, the name of the other configuration property might be: snapshot.collection.filter.overrides.customers.orders. Set this property to a valid filter expression that retrieves only the items that you want in the snapshot. When the connector performs a snapshot, it retrieves only the items that matches the filter expression.</td>
</tr>
<tr>
<td>provide.transaction.metadata</td>
<td>false</td>
<td>When set to true Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.</td>
</tr>
<tr>
<td>retrievable.restart.connector.wait.ms</td>
<td>10000 (10 seconds)</td>
<td>The number of milliseconds to wait before restarting a connector after a retrievable error occurs.</td>
</tr>
<tr>
<td>mongodb.poll.interval.ms</td>
<td>30000</td>
<td>The interval in which the connector polls for new, removed, or changed replica sets.</td>
</tr>
<tr>
<td>mongodb.connect.timeout.ms</td>
<td>10000 (10 seconds)</td>
<td>The number of milliseconds the driver will wait before a new connection attempt is aborted.</td>
</tr>
<tr>
<td>mongodb.socket.timeout.ms</td>
<td>0</td>
<td>The number of milliseconds before a send/receive on the socket can take before a timeout occurs. A value of 0 disables this behavior.</td>
</tr>
<tr>
<td>mongodb.server.select.timeout.ms</td>
<td>30000 (30 seconds)</td>
<td>The number of milliseconds the driver will wait to select a server before it times out and throws an error.</td>
</tr>
</tbody>
</table>

### 4.6. MONGODB CONNECTOR COMMON ISSUES

Debezium is a distributed system that captures all changes in multiple upstream databases, and will never miss or lose an event. Of course, when the system is operating nominally or being administered
carefully, then Debezium provides exactly once delivery of every change event. However, if a fault does happen then the system will still not lose any events, although while it is recovering from the fault it may repeat some change events. Thus, 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.

4.6.1. Configuration and startup errors

The connector will fail upon startup, report an error/exception in the log, and stop running when the connector’s configuration is invalid, or when the connector repeatedly fails to connect to MongoDB using the specified connectivity parameters. Reconnection is done using exponential backoff, and the maximum number of attempts is configurable.

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.

4.6.2. 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.

<table>
<thead>
<tr>
<th>Reconnection attempt number</th>
<th>Delay before attempt, in seconds</th>
<th>Total delay before attempt, in minutes and seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>00:01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>00:03</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>00:07</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>00:15</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>00:31</td>
</tr>
</tbody>
</table>
### 4.6.3. Kafka Connect process stops gracefully

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

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

### 4.6.4. 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.

<table>
<thead>
<tr>
<th>Reconnection attempt number</th>
<th>Delay before attempt, in seconds</th>
<th>Total delay before attempt, in minutes and seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>32</td>
<td>01:03</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>02:07</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>04:07</td>
</tr>
<tr>
<td>9</td>
<td>120</td>
<td>06:07</td>
</tr>
<tr>
<td>10</td>
<td>120</td>
<td>08:07</td>
</tr>
<tr>
<td>11</td>
<td>120</td>
<td>10:07</td>
</tr>
<tr>
<td>12</td>
<td>120</td>
<td>12:07</td>
</tr>
<tr>
<td>13</td>
<td>120</td>
<td>14:07</td>
</tr>
<tr>
<td>14</td>
<td>120</td>
<td>16:07</td>
</tr>
<tr>
<td>15</td>
<td>120</td>
<td>18:07</td>
</tr>
<tr>
<td>16</td>
<td>120</td>
<td>20:07</td>
</tr>
</tbody>
</table>
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.

4.6.5. 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.

4.6.6. Connector is stopped for a duration

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.

4.6.7. MongoDB loses writes

It is possible for MongoDB to lose commits in specific failure situations. For example, if the primary applies a change and records it in its oplog before it then crashes unexpectedly, the secondary nodes may not have had a chance to read those changes from the primary’s oplog before the primary crashed. If one such secondary is then elected as primary, its oplog is missing the last changes that the old primary had recorded and no longer has those changes.

In these cases where MongoDB loses changes recorded in a primary’s oplog, it is possible that the MongoDB connector may or may not capture these lost changes. At this time, there is no way to prevent this side effect of MongoDB.
CHAPTER 5. 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 5.1, “Overview of Debezium SQL Server connector”
- Section 5.2, “How Debezium SQL Server connectors work”
- Section 5.2.5, “Descriptions of Debezium SQL Server connector data change events”
- Section 5.2.7, “How Debezium SQL Server connectors map data types”
- Section 5.3, “Setting up SQL Server to run a Debezium connector”
- Section 5.4, “Deployment of Debezium SQL Server connectors”
- Section 5.5, “Refreshing capture tables after a schema change”
- Section 5.6, “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.

5.1. 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 Numbers).
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.

### 5.2. 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 5.2.1, “How Debezium SQL Server connectors perform database snapshots”
- Section 5.2.2, “How Debezium SQL Server connectors read change data tables”
- Section 5.2.3, “Default names of Kafka topics that receive Debezium SQL Server change event records”
- Section 5.2.4, “How the Debezium SQL Server connector uses the schema change topic”
- Section 5.2.5, “Descriptions of Debezium SQL Server connector data change events”
- Section 5.2.6, “Debezium SQL Server connector-generated events that represent transaction boundaries”

#### 5.2.1. 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 the **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.

5.2.2. 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.

5.2.3. 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.

5.2.4. 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:

```json
{
   "schema": {
      ...,
   },
   "payload": {
      "source": {
         "version": "1.4.2.Final",
         "connector": "sqlserver",
      },
```
<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>server1</td>
<td></td>
</tr>
<tr>
<td>ts_ms</td>
<td>1588252618953</td>
</tr>
<tr>
<td>snapshot</td>
<td>true</td>
</tr>
<tr>
<td>db</td>
<td>testDB</td>
</tr>
<tr>
<td>schema</td>
<td>dbo</td>
</tr>
<tr>
<td>table</td>
<td>customers</td>
</tr>
<tr>
<td>change_lsn</td>
<td>null</td>
</tr>
<tr>
<td>commit_lsn</td>
<td>000000025:00000d98:00a2</td>
</tr>
<tr>
<td>event_serial_no</td>
<td>null</td>
</tr>
<tr>
<td>databaseName</td>
<td>testDB</td>
</tr>
<tr>
<td>schemaName</td>
<td>dbo</td>
</tr>
<tr>
<td>ddl</td>
<td>null</td>
</tr>
<tr>
<td>tableChanges</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>type: &quot;CREATE&quot;,</td>
</tr>
</tbody>
</table>
|     |   id: ""testDB".dbo."customers"",
|     |   table: {                     |
|     |     defaultCharsetName: null, |
|     |     primaryKeyColumnNames: [    |
|     |     "id"                      |
|     |   },                          |
|     |   columns: [                  |
|     |     {                         |
|     |       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,       |
|     |   }                           |
|   } |                              |
Table 5.1. Descriptions of fields in messages emitted to the schema change topic

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>databaseName</code> <code>schemaName</code></td>
<td>Identifies the database and the schema that contain the change.</td>
</tr>
<tr>
<td>2</td>
<td><code>ddl</code></td>
<td>Always <strong>null</strong> for the SQL Server connector. For other connectors, this field contains the DDL responsible for the schema change. This DDL is not available to SQL Server connectors.</td>
</tr>
<tr>
<td>3</td>
<td><code>tableChanges</code></td>
<td>An array of one or more items that contain the schema changes generated by a DDL command.</td>
</tr>
<tr>
<td>4</td>
<td><code>type</code></td>
<td>Describes the kind of change. The value is one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>CREATE</strong> - table created</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>ALTER</strong> - table modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>DROP</strong> - table deleted</td>
</tr>
</tbody>
</table>
### 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:

```json
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "string",
                "optional": false,
                "field": "databaseName"
            }
        ],
        "optional": false,
        "name": "io.debezium.connector.sqlserver.SchemaChangeKey"
    },
    "payload": {
        "databaseName": "testDB"
    }
}
```
Table 5.2. Overview of change event basic content

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

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 5.2.5.1, "About keys in Debezium SQL Server change events"
- Section 5.2.5.2, "About values in Debezium SQL Server change events"

5.2.5.1. 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

```sql
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:

```json
{"schema": {  
  "type": "struct",
  "fields": [{  
    "type": "int32",
    "optional": false,
    "field": "id"
  }]}
```
Table 5.3. Description of change event key

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s payload portion.</td>
</tr>
<tr>
<td>2</td>
<td>fields</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>optional</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 4    | server1.dbo.customers.Key | Name of the schema that defines the structure of the key’s payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format connector-name.database-schema-name.table-name.Key. In this example:  
  - server1 is the name of the connector that generated this event.  
  - dbo is the database schema for the table that was changed.  
  - customers is the table that was updated. |
| 5    | payload    | Contains the key for the row for which this change event was generated. In this example, the key, contains a single id field whose value is 1004. |

5.2.5.2. 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:

```sql
CREATE TABLE customers (  
id INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,  
first_name VARCHAR(255) NOT NULL,
```
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:

```json
{
  "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.dbo.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          }
        ],
        "optional": false,
        "name": "server1.dbo.customers.Value",
        "field": "after"
      }
    ]
  }
}
```

```sql
last_name VARCHAR(255) NOT NULL,
email VARCHAR(255) NOT NULL UNIQUE
);```

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:
```json
{
    "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",
"field": "source"
},
{
"type": "string",
"optional": false,
"field": "op"
},
{
"type": "int64",
"optional": true,
"field": "ts_ms"
}
],
"optional": false,
"name": "server1.dbo.customers.Envelope"
},
"payload": {
"before": null,
"after": {
"id": 1005,
"first_name": "john",
"last_name": "doe",
"email": "john.doe@example.org"
}
},
"source": {
"version": "1.4.2.Final",
"connector": "sqlserver",
"name": "server1"}
Table 5.4. Descriptions of create event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>In the schema section, each name field specifies the schema for a field in the value’s payload. server1.dbo.customers.Value is the schema for the payload’s before and after fields. This schema is specific to the customers table. Names of schemas for before and after fields are of the form logicalName.database-schemaName.tableName.Value, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 5    | payload    | The value’s actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>before</td>
<td>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.</td>
</tr>
<tr>
<td>7</td>
<td>after</td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>source</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debezium version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Connector type and name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Database and schema names</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Timestamp for when the change was made in the database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the event was part of a snapshot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name of the table that contains the new row</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Server log offsets</td>
</tr>
<tr>
<td>9</td>
<td>op</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- c = create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- u = update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- d = delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- r = read (applies to only snapshots)</td>
</tr>
<tr>
<td>10</td>
<td>ts_ms</td>
<td>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.</td>
</tr>
</tbody>
</table>

In the source object, ts_ms indicates the time when a change was committed in the database. By comparing the value for payload.source.ts_ms with the value for payload.ts_ms, you can determine the lag between the source database update and Debezium.
**update events**

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

```json
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "after": {
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "source": {
      "version": "1.4.2.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",
    "ts_ms": 1559729998706
  }
}
```

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>before</strong></td>
<td>An optional field that specifies the state of the row before the event occurred. In an <em>update</em> event value, the <strong>before</strong> field contains a field for each table column and the value that was in that column before the database commit. In this example, the <strong>email</strong> value is <code>john.doe@example.org</code>.</td>
</tr>
<tr>
<td>2</td>
<td><strong>after</strong></td>
<td>An optional field that specifies the state of the row after the event occurred. You can compare the <strong>before</strong> and <strong>after</strong> structures to determine what the update to this row was. In the example, the <strong>email</strong> value is now <code>noreply@example.org</code>.</td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 3    | **source** | Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a create event, but some values are different, for example, the sample update event has a different offset. The source metadata includes:  
  - Debezium version  
  - Connector type and name  
  - Database and schema names  
  - Timestamp for when the change was made in the database  
  - If the event was part of a snapshot  
  - Name of the table that contains the new row  
  - Server log offsets  
  The **event_serial_no** field differentiates events that have the same commit and change LSN. Typical situations for when this field has a value other than **1**:  
  - **update** events have the value set to **2** because the update generates two events in the CDC change table of SQL Server ([see the source documentation for details](#)). The first event contains the old values and the second contains contains new values. The connector uses values in the first event to create the second event. The connector drops the first event.  
  - When a primary key is updated SQL Server emits two events. A **delete** event for the removal of the record with the old primary key value and a **create** event for the addition of the record with the new primary key. Both operations share the same commit and change LSN and their event numbers are **1** and **2**, respectively. |
| 4    | **op**     | Mandatory string that describes the type of operation. In an update event value, the **op** field value is **u**, signifying that this row changed because of an update. |
| 5    | **ts_ms**  | Optional field that displays the time at which the connector processed the event. In the event message envelope, the time is based on the system clock in the JVM running the Kafka Connect task.  
  In the **source** object, **ts_ms** indicates the time when the change was committed to the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium. |
NOTE

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

delete events

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

```json
{
  "schema": { ... },
  "payload": {
    "before": { <>
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "after": null,
    "source": {
      "version": "1.4.2.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",
    "ts_ms": 1559730450205
  }
}
```

Table 5.6. Descriptions of delete event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>Optional field that specifies the state of the row before the event occurred. In a delete event value, the <strong>before</strong> field contains the values that were in the row before it was deleted with the database commit.</td>
</tr>
<tr>
<td>2</td>
<td>after</td>
<td>Optional field that specifies the state of the row after the event occurred. In a delete event value, the <strong>after</strong> field is <strong>null</strong>, signifying that the row no longer exists.</td>
</tr>
</tbody>
</table>
**Item** | **Field name** | **Description**
--- | --- | ---
3 | **source** | Mandatory field that describes the source metadata for the event. In a delete event value, the source field structure is the same as for create and update events for the same table. Many source field values are also the same. In a delete event value, the ts_ms and pos field values, as well as other values, might have changed. But the source field in a delete event value provides the same metadata:
- Debezium version
- Connector type and name
- Database and schema names
- Timestamp for when the change was made in the database
- If the event was part of a snapshot
- Name of the table that contains the new row
- Server log offsets

4 | **op** | Mandatory string that describes the type of operation. The op field value is d, signifying that this row was deleted.

5 | **ts_ms** | Optional field that displays the time at which the connector processed the event. In the event message envelope, the time is based on the system clock in the JVM running the Kafka Connect task.

In the source object, ts_ms indicates the time that the change was made in the database. By comparing the value for payload.source.ts_ms with the value for payload.ts_ms, you can determine the lag between the source database update and Debezium.

---

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

**Tombstone events**

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

**5.2.6. 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.

The following example shows a typical transaction boundary message:

**Example: SQL Server connector transaction boundary event**

```json
{
   "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`.

### 5.2.6.1. 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:

```json
{
  "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"
  }
}
```

5.2.7. 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
- Table 5.8, “Decimal values”
- Timestamp values

Basic types

The following table shows how the connector maps basic SQL Server data types.
<table>
<thead>
<tr>
<th>SQL Server data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>BOOLEAN</td>
<td>n/a</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>INT</td>
<td>INT32</td>
<td>n/a</td>
</tr>
<tr>
<td>BIGINT</td>
<td>INT64</td>
<td>n/a</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT32</td>
<td>n/a</td>
</tr>
<tr>
<td>FLOAT[(N)]</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>CHAR[(N)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>VARCHAR[(N)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>TEXT</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>NCHAR[(N)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>NVARCHAR[(N)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>NTEXT</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>XML</td>
<td>STRING</td>
<td>io.debezium.data.Xml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains the string representation of an XML document</td>
</tr>
<tr>
<td>DATETIMEOFFSET[(P)]</td>
<td>STRING</td>
<td>io.debezium.time.ZonedTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A string representation of a timestamp with timezone information, where the timezone is GMT</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>SQL Server data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>io.debezium.time.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME(0), TIME(1), TIME(2), TIME(3)</td>
<td>INT32</td>
<td>io.debezium.time.Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past midnight, and does not include timezone information.</td>
</tr>
<tr>
<td>TIME(4), TIME(5), TIME(6)</td>
<td>INT64</td>
<td>io.debezium.time.MicroTime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of microseconds past midnight, and does not include timezone information.</td>
</tr>
<tr>
<td>TIME(7)</td>
<td>INT64</td>
<td>io.debezium.time.NanoTime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of nanoseconds past midnight, and does not include timezone information.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>INT64</td>
<td>io.debezium.time.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>INT64</td>
<td>io.debezium.time.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>DATETIME2(0), DATETIME2(1), DATETIME2(2), DATETIME2(3)</td>
<td>INT64</td>
<td>io.debezium.time.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>SQL Server data type</td>
<td>Literal type (schema type)</td>
<td>Semantic type (schema name) and Notes</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DATETIME2(4), DATETIME2(5),</td>
<td>INT64</td>
<td>io.debezium.time.MicroTimestamp</td>
</tr>
<tr>
<td>DATETIME2(6)</td>
<td></td>
<td>Represents the number of microseconds past the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>DATETIME2(7)</td>
<td>INT64</td>
<td>io.debezium.time.NanoTimestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of nanoseconds past the epoch, and does not include timezone information.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>SQL Server data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>org.apache.kafka.connect.data.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME([P])</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 &gt; 3.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds since the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds past the epoch, and does not include timezone information.</td>
</tr>
</tbody>
</table>
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.

Table 5.8. Decimal values

<table>
<thead>
<tr>
<th>SQL Server data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(P[,S])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td>DECIMAL[(P[,S])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td>MONEY</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
</tbody>
</table>

The **scale** schema parameter contains an integer that represents how many digits the decimal point was shifted. The **connect.decimal.precision** schema parameter contains an integer that represents the precision of the given decimal value.

5.3. 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 5.3.1, “Enabling CDC on the SQL Server database”
- Section 5.3.2, “Enabling CDC on a SQL Server table”
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.

### 5.3.1. 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
```
5.3.2. 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`:

   ```sql
   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.

5.3.3. 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**

```sql
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.

### 5.3.4. SQL Server on Azure

The Debezium SQL Server connector has not been tested with SQL Server on Azure.

### 5.3.5. 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.

5.3.6. 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.

5.4. 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 5.4.1, “Deploying Debezium SQL Server connectors”
- Section 5.4.2, “Descriptions of Debezium SQL Server connector configuration properties”

5.4.1. 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 Installing Debezium 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:
c. Create a Docker file that uses `registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0` as the base image. For example, from a terminal window, enter the following, replacing `my-plugins` with the name of your plug-ins directory:

```bash
cat <<EOF >debezium-container-for-sqlserver.yaml
FROM registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/
USER 1001
EOF
```

You can specify any file name that you want. 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:

```bash
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:

```bash
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:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
category: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
```
metadata.annotations indicates to the Cluster Operator that KafkaConnector resources are used to configure connectors in this Kafka Connect cluster.

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**

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: fulfillment-connector
labels:
  strimzi.io/cluster: my-connect-cluster
annotations:
  strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.sqlserver.SqlServerConnector
  config:
    database.hostname: 192.168.99.100
    database.port: 1433
    database.user: debezium
    database.password: dbz
    database.dbname: testDB
    database.include.list: dbo.customers
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092
    database.history.kafka.topic: dbhistory.fulfillment
```
Table 5.9. Descriptions of connector configuration settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The name of our connector when we register it with a Kafka Connect service.</td>
</tr>
<tr>
<td>2</td>
<td>The name of this SQL Server connector class.</td>
</tr>
<tr>
<td>3</td>
<td>The address of the SQL Server instance.</td>
</tr>
<tr>
<td>4</td>
<td>The port number of the SQL Server instance.</td>
</tr>
<tr>
<td>5</td>
<td>The name of the SQL Server user.</td>
</tr>
<tr>
<td>6</td>
<td>The password for the SQL Server user.</td>
</tr>
<tr>
<td>7</td>
<td>The name of the database to capture changes from.</td>
</tr>
<tr>
<td>8</td>
<td>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.</td>
</tr>
<tr>
<td>9</td>
<td>A list of all tables whose changes Debezium should capture.</td>
</tr>
<tr>
<td>10</td>
<td>The list of Kafka brokers that this connector will use to write and recover DDL statements to the database history topic.</td>
</tr>
<tr>
<td>11</td>
<td>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.</td>
</tr>
</tbody>
</table>

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:

```bash
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:

```bash
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:
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.

### 5.4.2. 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.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)</td>
</tr>
<tr>
<td>connector.class</td>
<td></td>
<td>The name of the Java class for the connector. Always use a value of <code>io.debezium.connector.sqlserver.SqlServerConnector</code> for the SQL Server connector.</td>
</tr>
<tr>
<td>tasks.max</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>database.hostname</td>
<td></td>
<td>IP address or hostname of the SQL Server database server.</td>
</tr>
<tr>
<td>database.port</td>
<td>1433</td>
<td>Integer port number of the SQL Server database server.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>database.user</td>
<td></td>
<td>Username to use when connecting to the SQL Server database server.</td>
</tr>
<tr>
<td>database.password</td>
<td></td>
<td>Password to use when connecting to the SQL Server database server.</td>
</tr>
<tr>
<td>database.dbname</td>
<td></td>
<td>The name of the SQL Server database from which to stream the changes</td>
</tr>
<tr>
<td>database.server.name</td>
<td></td>
<td>Logical name that identifies and provides a namespace for the SQL Server database server that you want Debezium to capture. The logical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>alphanumeric characters and underscores should be used.</td>
</tr>
<tr>
<td>database.history.kafka.topic</td>
<td></td>
<td>The full name of the Kafka topic where the connector will store the database schema history.</td>
</tr>
<tr>
<td>database.history.kafka.bootstrap.servers</td>
<td></td>
<td>A list of host and port pairs that the connector will use for establishing an initial connection to the Kafka cluster. This connection is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>used for retrieving database schema history previously stored by the connector, and for writing each DDL statement read from the source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>database. This should point to the same Kafka cluster used by the Kafka Connect process.</td>
</tr>
<tr>
<td>table.include.list</td>
<td></td>
<td>An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables that you want Debezium to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>capture; any table that is not included in table.include.list is excluded from capture. Each identifier is of the form</td>
</tr>
<tr>
<td></td>
<td></td>
<td>schemaName.tableName. By default, the connector captures all non-system tables for the designated schemas. Must not be used with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>table.exclude.list.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>table.exclude.list</td>
<td></td>
<td>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 <code>table.exclude.list</code>. Each identifier is of the form <code>schemaName.tableName</code>. Must not be used with <code>table.include.list</code>.</td>
</tr>
<tr>
<td>column.include.list</td>
<td>empty string</td>
<td>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 <code>schemaName.tableName.columnName</code>. 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 <code>column.exclude.list</code> property.</td>
</tr>
<tr>
<td>column.exclude.list</td>
<td>empty string</td>
<td>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 <code>schemaName.tableName.columnName</code>. Note that primary key columns are always included in the event’s key, also if excluded from the value. Do not also set the <code>column.include.list</code> property.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| column.mask.hash._hash Algorithm_.with.salt._salt | n/a | An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in the change event message values with a field value consisting of the hashed value using the algorithm hashAlgorithm and salt salt. Based on the used hash function referential integrity is kept while data is pseudonymized. Supported hash functions are described in the MessageDigest section of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash is automatically shortened to the length of the column. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form schemaName(tableName).columnName. Example:  
  ```
column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = dbo.orders.customerName, dbo.shipment.customerName
  ```  
  where CzQMA0cB5K is a randomly selected salt. Note: Depending on the hashAlgorithm used, the salt selected and the actual data set, the resulting masked data set may not be completely anonymized. |
<p>| time.precision.mode | adaptive | Time, date, and timestamps can be represented with different kinds of precision, including adaptive (the default) captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column’s type; or connect always represents time and timestamp values using Kafka Connect’s built-in representations for Time, Date, and Timestamp, which uses millisecond precision regardless of the database columns’ precision. See temporal values. |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include.schema.changes</td>
<td>true</td>
<td>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.</td>
</tr>
<tr>
<td>tombstones.on.delete</td>
<td>true</td>
<td>Controls whether a tombstone event should be generated after a delete event. When true the delete operations are represented by a delete event and a subsequent tombstone event. When false only a delete event is sent. Emitting the tombstone event (the default behavior) allows Kafka to completely delete all events pertaining to the given key once the source record got deleted.</td>
</tr>
<tr>
<td>column.truncate.to.length.chars</td>
<td>n/a</td>
<td>An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event message values if the field values are longer than the specified number of characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer. Fully-qualified names for columns are of the form schemaName.tableName.columnName.</td>
</tr>
<tr>
<td>column.mask.with.length.chars</td>
<td>n/a</td>
<td>An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*) characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form schemaName.tableName.columnName.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>column.propagate.source.type</td>
<td>n/a</td>
<td>An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters __debezium.source.column.type, __debezium.source.column.length and __debezium.source.column.scale is used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of the form schemaName.tableName.columnName.</td>
</tr>
<tr>
<td>datatype.propagate.source.type+</td>
<td>n/a</td>
<td>An optional comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters __debezium.source.column.type, __debezium.source.column.length and __debezium.source.column.scale will be used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of the form schemaName.tableName.typeName. See SQL Server data types for the list of SQL Server-specific data type names.</td>
</tr>
<tr>
<td>message.key.columns</td>
<td>n/a</td>
<td>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 <code>&lt;fully-qualified table&gt;:</code>&lt;a comma-separated list of columns&gt; representing the custom key. Fully-qualified tables could be defined as schemaName.tableName.</td>
</tr>
</tbody>
</table>
#### binary.handling.mode

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bytes</td>
<td>Specifies how binary (binary, varbinary) columns should be represented in change events, including: <code>bytes</code> represents binary data as byte array (default), <code>base64</code> represents binary data as base64-encoded String, <code>hex</code> represents binary data as hex-encoded (base16) String</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>snapshot.mode</td>
<td>initial</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>initial</strong>: 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>initial_only</strong>: Takes a snapshot of structure and data like <code>initial</code> but instead does not transition into streaming changes once the snapshot has completed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <strong>schema_only</strong>: Takes a snapshot of the structure of captured tables only; useful if only changes happening from now onwards should be propagated to topics.</td>
</tr>
<tr>
<td>snapshot.include.collection.list</td>
<td>All tables specified in table.include.list</td>
<td>An optional, comma-separated list of regular expressions that match names of schemas specified in <code>table.include.list</code> for which you want to take the snapshot.</td>
</tr>
</tbody>
</table>
Mode to control which transaction isolation level is used and how long the connector locks tables that are designated for capture. The following values are supported:

- `read_uncommitted`
- `read_committed`
- `repeatable_read`
- `snapshot`
- `exclusive` (exclusive mode uses repeatable read isolation level, however, it takes the exclusive lock on all tables to be read).

The `snapshot`, `read_committed` and `read_uncommitted` modes do not prevent other transactions from updating table rows during initial snapshot. The `exclusive` and `repeatable_read` modes do prevent concurrent updates.

Mode choice also affects data consistency. Only `exclusive` and `snapshot` modes guarantee full consistency, that is, initial snapshot and streaming logs constitute a linear history. In case of `repeatable_read` and `read_committed` modes, it might happen that, for instance, a record added appears twice - once in initial snapshot and once in streaming phase. Nonetheless, that consistency level should do for data mirroring. For `read_uncommitted` there are no data consistency guarantees at all (some data might be lost or corrupted).
### Property: `source.timestamp.mode`

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>commit</code></td>
<td>String that represents the criteria of the attached timestamp within the source record (ts_ms).</td>
</tr>
</tbody>
</table>

- **commit** (default) sets the source timestamp to the time when the record was committed to the database.
- **processing** sets the source timestamp to the time when Debezium accesses the record in the change table. Use the `processing` option if you want [prodname] to set the top level `ts_ms` value, or if you want to avoid the additional cost of Debezium querying the database to extract the LSN timestamps.

### Property: `event.processing.failurehandling.mode`

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fail</code></td>
<td>Specifies how the connector should react to exceptions during processing of events. <code>fail</code> will propagate the exception (indicating the offset of the problematic event), causing the connector to stop. <code>warn</code> will cause the problematic event to be skipped and the offset of the problematic event to be logged. <code>skip</code> will cause the problematic event to be skipped.</td>
</tr>
</tbody>
</table>

### Property: `poll.interval.ms`

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>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.</td>
</tr>
</tbody>
</table>

### Property: `max.queue.size`

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>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 <code>max.batch.size</code> property.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>max.batch.size</td>
<td>2048</td>
</tr>
<tr>
<td>heartbeat.interval.ms</td>
<td>0</td>
</tr>
<tr>
<td>heartbeat.topics.prefix</td>
<td>__debezium-heartbeat</td>
</tr>
<tr>
<td>snapshot.delay.ms</td>
<td></td>
</tr>
<tr>
<td>snapshot.fetch.size</td>
<td>2000</td>
</tr>
<tr>
<td>query.fetch.size</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>snapshot.lock.timeout.ms</td>
<td>10000</td>
</tr>
<tr>
<td>snapshot.select.statementoverrides</td>
<td></td>
</tr>
<tr>
<td>sanitize.field.names</td>
<td>true</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>database.server.timezone</td>
<td></td>
</tr>
<tr>
<td>provide.transaction.meta data</td>
<td>false</td>
</tr>
<tr>
<td>retrievable.restart.connector.wait.ms</td>
<td>10000 (10 seconds)</td>
</tr>
</tbody>
</table>

The connector also supports pass-through configuration properties that are used when creating the Kafka producer and consumer. Specifically, all connector configuration properties that begin with the database.history.producer. prefix are used (without the prefix) when creating the Kafka producer that writes to the database history, and all those that begin with the prefix database.history.consumer. are used (without the prefix) when creating the Kafka consumer that reads the database history upon connector startup.

For example, the following connector configuration properties can be used to secure connections to the Kafka broker:

In addition to the pass-through to the Kafka producer and consumer, the properties starting with database., e.g. database.applicationName=debezium are passed to the JDBC URL.

- database.history.producer.security.protocol=SSL
- database.history.producer.ssl.key.password=test1234
- database.history.producer.ssl.truststore.password=test1234
- database.history.consumer.security.protocol=SSL
Be sure to consult the Kafka documentation for all of the configuration properties for Kafka producers and consumers. (The SQL Server connector does use the new consumer.)

5.5. 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 5.5.1, “Running an offline update after a schema change”. In offline schema updates, capture tables are updated after you stop the Debezium connector.

- Section 5.5.2, “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.
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`.

### 5.5.1. 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 the `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.

### 5.5.2. 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:

   ```sql
   ALTER TABLE customers ADD phone_number VARCHAR(32);
   ```

2. Create the new capture instance by running the `sys.sp_cdc_enable_table` stored procedure.

   ```sql
   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:

   ```sql
   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:

```plaintext
connect_1  | 2019-01-17 10:11:14,924 INFO  ||  Multiple capture instances present for the same table: Capture instance "dbo_customers" [sourceTableId=testDB.dbo.customers, changeTableId=testDB.cdc.dbo_customers_CT, startLsn=00000024:00000d98:0036, changeTableObjectId=1525580473, stopLsn=00000025:00000ef8:0048] and Capture instance "dbo_customers_v2" [sourceTableId=testDB.dbo.customers, changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048, changeTableObjectId=1749581271, stopLsn=NULL] [io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource] connect_1  | 2019-01-17 10:11:14,924 INFO  ||  Schema will be changed for ChangeTable [captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers, changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048, changeTableObjectId=1749581271, stopLsn=NULL] [io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
```
Eventually, the **phone_number** field is added to the schema and its value appears in messages written to the Kafka topic.

```json
...
   "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.

```sql
EXEC sys.sp_cdc_disable_table @source_schema = 'dbo', @source_name = 'dbo_customers', @capture_instance = 'dbo_customers';
GO
```

### 5.6. 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.

#### 5.6.1. Debezium SQL Server connector snapshot metrics

The MBean is `debezium.sql_server:type=connector-metrics,context=snapshot,server=<database.server.name>`.
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last snapshot event that the connector has read.</td>
</tr>
<tr>
<td>MillisecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>TotalTableCount</td>
<td>int</td>
<td>The total number of tables that are being included in the snapshot.</td>
</tr>
<tr>
<td>RemainingTableCount</td>
<td>int</td>
<td>The number of tables that the snapshot has yet to copy.</td>
</tr>
<tr>
<td>SnapshotRunning</td>
<td>boolean</td>
<td>Whether the snapshot was started.</td>
</tr>
<tr>
<td>SnapshotAborted</td>
<td>boolean</td>
<td>Whether the snapshot was aborted.</td>
</tr>
<tr>
<td>SnapshotCompleted</td>
<td>boolean</td>
<td>Whether the snapshot completed.</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnapshotDurationInSeconds</td>
<td>long</td>
<td>The total number of seconds that the snapshot has taken so far, even if not complete.</td>
</tr>
<tr>
<td>RowsScanned</td>
<td>Map&lt;String, Long&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>The maximum buffer of the queue in bytes. It will be enabled if <code>max.queue.size.in.bytes</code> is passed with a positive long value.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

### 5.6.2. Debezium SQL Server connector streaming metrics

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last streaming event that the connector has read.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>Connected</td>
<td>boolean</td>
<td>Flag that denotes whether the connector is currently connected to the database server.</td>
</tr>
<tr>
<td>MilliSecondsBehindSource</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfCommittedTransactions</td>
<td>long</td>
<td>The number of processed transactions that were committed.</td>
</tr>
<tr>
<td>SourceEventPosition</td>
<td>Map&lt;String, String&gt;</td>
<td>The coordinates of the last received event.</td>
</tr>
<tr>
<td>LastTransactionId</td>
<td>string</td>
<td>Transaction identifier of the last processed transaction.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>The maximum buffer of the queue in bytes.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

5.6.3. Debezium SQL Server connector schema history metrics

The MBean is `debezium.sql_server:type=connector-metrics,context=schema-history,server=<database.server.name>`. 
<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>string</td>
<td>One of <strong>STOPPED</strong>, <strong>RECOVERING</strong> (recovering history from the storage), <strong>RUNNING</strong> describing the state of the database history.</td>
</tr>
<tr>
<td>RecoveryStartTime</td>
<td>long</td>
<td>The time in epoch seconds at what recovery has started.</td>
</tr>
<tr>
<td>ChangesRecovered</td>
<td>long</td>
<td>The number of changes that were read during recovery phase.</td>
</tr>
<tr>
<td>ChangesApplied</td>
<td>long</td>
<td>the total number of schema changes applied during recovery and runtime.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastRecoveredChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was recovered from the history store.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastAppliedChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was applied.</td>
</tr>
<tr>
<td>LastRecoveredChange</td>
<td>string</td>
<td>The string representation of the last change recovered from the history store.</td>
</tr>
<tr>
<td>LastAppliedChange</td>
<td>string</td>
<td>The string representation of the last applied change.</td>
</tr>
</tbody>
</table>
CHAPTER 6. 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 6.1, “Overview of Debezium Db2 connector”
- Section 6.2, “How Debezium Db2 connectors work”
- Section 6.3, “Descriptions of Debezium Db2 connector data change events”
- Section 6.4, “How Debezium Db2 connectors map data types”
- Section 6.5, “Setting up Db2 to run a Debezium connector”
- Section 6.6, “Deployment of Debezium Db2 connectors”
- Section 6.7, “Monitoring Debezium Db2 connector performance”
- Section 6.8, “Managing Debezium Db2 connectors”
- Section 6.9, “Updating schemas for Db2 tables in capture mode for Debezium connectors”

6.1. 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.

### 6.2. 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 6.2.1, “How Debezium Db2 connectors perform database snapshots”**
- **Section 6.2.2, “How Debezium Db2 connectors read change-data tables”**
- **Section 6.2.3, “Default names of Kafka topics that receive Debezium Db2 change event records”**
- **Section 6.2.4, “About the Debezium Db2 connector schema change topic”**
6.2.1. 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.

6.2.2. 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.

### 6.2.3. 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 `MYSCHEMA` schema. The connector would emit events to these four Kafka topics:

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

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

### 6.2.4. 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.
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:

```json
{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "1.4.2.Final",
      "connector": "db2",
      "name": "db2",
      "ts_ms": "1588252618953",
      "snapshot": "true",
      "db": "testdb",
      "schema": "DB2INST1",
      "table": "CUSTOMERS",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "databaseName": "TESTDB",
    "schemaName": "DB2INST1",
    "ddl": null,
    "tableChanges": [
      {
        "type": "CREATE",
        "id": ""DB2INST1"."CUSTOMERS!",
        "table": {
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ "ID"
        },
        "columns": [
          {
            "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 6.1. Descriptions of fields in messages emitted to the schema change topic

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>databaseName</code>&lt;br&gt;schemaName</td>
<td>Identifies the database and the schema that contain the change.</td>
</tr>
<tr>
<td>2</td>
<td><code>ddl</code></td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td><code>tableChanges</code></td>
<td>An array of one or more items that contain the schema changes generated by a DDL command.</td>
</tr>
</tbody>
</table>
| 4    | `type` | Describes the kind of change. The value is one of the following:  
  - `CREATE` - table created  
  - `ALTER` - table modified  
  - `DROP` - table deleted |
| 5    | `id` | Full identifier of the table that was created, altered, or dropped. |
| 6    | `table` | Represents table metadata after the applied change. |
| 7    | `primaryKeyColumnNames` | List of columns that compose the table’s primary key. |
| 8    | `columns` | Metadata for each column in the changed table. |

In messages 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:

```json
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "databaseName"
      }
    ],
    "optional": false,
    "name": "io.debezium.connector.db2.SchemaChangeKey"
}
```
6.2.5. 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

```json
{
    "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:

```json
{
  "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"
  }
}
```

6.3. 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:

```json
{
  "schema": { 1
    ...
  },
  "payload": { 2
    ...
  },
  "schema": { 3
    ...
}
```
Table 6.2. Overview of change event basic content

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

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 6.3.1, "About keys in Debezium db2 change events"
- Section 6.3.2, "About values in Debezium Db2 change events"

### 6.3.1. 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

```sql
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:

```json
{
  "schema": {
    "type": "struct",
    "fields": [
      
```
Table 6.3. Description of change event key

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>The schema portion of the key specifies a Kafka Connect schema that describes what is in the key’s payload portion.</td>
</tr>
<tr>
<td>2</td>
<td>fields</td>
<td>Specifies each field that is expected in the payload, including each field’s name, type, and whether it is required.</td>
</tr>
<tr>
<td>3</td>
<td>optional</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 4    | mydatabase.MYSHEMA.CUSTOMERS.Key | Name of the schema that defines the structure of the key’s payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format connector-name.database-name.table-name.Key. In this example:  
  - mydatabase is the name of the connector that generated this event.  
  - MYSHEMA 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. |

6.3.2. 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

```sql
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:

```json
{
    "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": "mydatabase.MYSHEMA.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": "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",  
"field": "source"},  
{  "type": "string",  "optional": false,  "field": "op"},  
{  "type": "int64",  "optional": true,  "field": "ts_ms"},  
"optional": false,  "name": "mydatabase.MYSHEMA.CUSTOMERS.Envelope"},  
"payload": {  "before": null,  "after": {  "ID": 1005,  "FIRST_NAME": "john",  "LAST_NAME": "doe",  "EMAIL": "john.doe@example.org"},  "source": {  "version": "1.4.2.Final",  "connector": "db2",}
### Table 6.4. Descriptions of create event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>schema</td>
<td>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.</td>
</tr>
<tr>
<td>2</td>
<td>name</td>
<td>In the <code>schema</code> section, each <code>name</code> field specifies the schema for a field in the value’s payload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>mydatabase.MYSHEMA.CUSTOMERS.Value</strong> is the schema for the payload’s <code>before</code> and <code>after</code> fields. This schema is specific to the <code>customers</code> table. The connector uses this schema for all rows in the <code>MYSHEMA.CUSTOMERS</code> table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Names of schemas for <code>before</code> and <code>after</code> fields are of the form <code>logicalName.schemaName.tableName.Value</code>, 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.</td>
</tr>
<tr>
<td>3</td>
<td>name</td>
<td><strong>io.debezium.connector.db2.Source</strong> is the schema for the payload’s <code>source</code> field. This schema is specific to the Db2 connector. The connector uses it for all events that it generates.</td>
</tr>
<tr>
<td>4</td>
<td>name</td>
<td><strong>mydatabase.MYSHEMA.CUSTOMERS.Envelope</strong> is the schema for the overall structure of the payload, where <code>mydatabase</code> is the database, <code>MYSHEMA</code> is the schema, and <code>CUSTOMERS</code> is the table.</td>
</tr>
<tr>
<td>5</td>
<td>payload</td>
<td>The value’s actual data. This is the information that the change event is providing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>6</td>
<td>before</td>
<td>An optional field that specifies the state of the row before the event occurred. When the <code>op</code> field is <code>c</code> for create, as it is in this example, the <code>before</code> field is <code>null</code> since this change event is for new content.</td>
</tr>
<tr>
<td>7</td>
<td>after</td>
<td>An optional field that specifies the state of the row after the event occurred. In this example, the <code>after</code> field contains the values of the new row’s <code>ID</code>, <code>FIRST_NAME</code>, <code>LAST_NAME</code>, and <code>EMAIL</code> columns.</td>
</tr>
<tr>
<td>8</td>
<td>source</td>
<td>Mandatory field that describes the source metadata for the event. The <code>source</code> 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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Debezium version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Connector type and name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Timestamp for when the change was made in the database</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Whether the event is part of an ongoing snapshot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Name of the database, schema, and table that contain the new row</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Change LSN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Commit LSN (omitted if this event is part of a snapshot)</td>
</tr>
<tr>
<td>9</td>
<td>op</td>
<td>Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, <code>c</code> indicates that the operation created a row. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>c</code> = create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>u</code> = update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>d</code> = delete</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>r</code> = read (applies to only snapshots)</td>
</tr>
<tr>
<td>10</td>
<td>ts_ms</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In the <code>source</code> object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and Debezium.</td>
</tr>
</tbody>
</table>

**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": "1.4.2.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
    },
    "op": "u", \4
    "ts_ms": 1559729998706  \5
  }
}
```

**Table 6.5. Descriptions of update event value fields**

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>before</strong></td>
<td>An optional field that specifies the state of the row before the event occurred. In an update event value, the <strong>before</strong> 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 <strong>EMAIL</strong> value is <strong><a href="mailto:john.doe@example.com">john.doe@example.com</a></strong>.</td>
</tr>
<tr>
<td>2</td>
<td><strong>after</strong></td>
<td>An optional field that specifies the state of the row after the event occurred. You can compare the <strong>before</strong> and <strong>after</strong> structures to determine what the update to this row was. In the example, the <strong>EMAIL</strong> value is now <strong><a href="mailto:noreply@example.com">noreply@example.com</a></strong>.</td>
</tr>
<tr>
<td>Item</td>
<td>Field name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 3    | source     | Mandatory field that describes the source metadata for the event. The `source` field structure contains the same fields as in a `create` event, but some values are different, for example, the sample `update` event has different LSNs. You can use this information to compare this event to other events to know whether this event occurred before, after, or as part of the same commit as other events. The source metadata includes:  
  - Debezium version  
  - Connector type and name  
  - Timestamp for when the change was made in the database  
  - Whether the event is part of an ongoing snapshot  
  - Name of the database, schema, and table that contain the new row  
  - Change LSN  
  - Commit LSN (omitted if this event is part of a snapshot) |
| 4    | op         | Mandatory string that describes the type of operation. In an `update` event value, the `op` field value is `u`, signifying that this row changed because of an update. |
| 5    | ts_ms      | Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.  
  In the `source` object, `ts_ms` indicates the time that the change was made in the database. By comparing the value for `payload.source.ts_ms` with the value for `payload.ts_ms`, you can determine the lag between the source database update and Debezium. |

**NOTE**

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

**delete events**

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

```json
{
  "schema": { ... },
},
"payload": {
  "before": { ... }
}
```
Table 6.6. Descriptions of delete event value fields

<table>
<thead>
<tr>
<th>Item</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>before</td>
<td>Optional field that specifies the state of the row before the event occurred. In a delete event value, the <code>before</code> field contains the values that were in the row before it was deleted with the database commit.</td>
</tr>
<tr>
<td>2</td>
<td>after</td>
<td>Optional field that specifies the state of the row after the event occurred. In a delete event value, the <code>after</code> field is null, signifying that the row no longer exists.</td>
</tr>
<tr>
<td>3</td>
<td>source</td>
<td>Mandatory field that describes the source metadata for the event. In a delete event value, the <code>source</code> field structure is the same as for create and update events for the same table. Many <code>source</code> field values are also the same. In a delete event value, the <code>ts_ms</code> and LSN field values, as well as other values, might have changed. But the <code>source</code> field in a delete event value provides the same metadata:</td>
</tr>
</tbody>
</table>

- 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)
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.

### 6.4. 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 6.10, “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 6.7. Mappings for Db2 basic data types

<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>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.</td>
</tr>
<tr>
<td>BIGINT</td>
<td>INT64</td>
<td>n/a</td>
</tr>
<tr>
<td>BINARY</td>
<td>BYTES</td>
<td>n/a</td>
</tr>
<tr>
<td>BLOB</td>
<td>BYTES</td>
<td>n/a</td>
</tr>
<tr>
<td>CHAR[(N)]</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>CLOB</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>DATE</td>
<td>INT32</td>
<td><code>io.debezium.time.Date</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>String representation of a timestamp without timezone information</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>BYTES</td>
<td><code>org.apache.kafka.connect.data.Decimal</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>BYTES</td>
<td><code>org.apache.kafka.connect.data.Decimal</code></td>
</tr>
<tr>
<td>DBCLOB</td>
<td>STRING</td>
<td>n/a</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT64</td>
<td>n/a</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT32</td>
<td>n/a</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT32</td>
<td>n/a</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT16</td>
<td>n/a</td>
</tr>
<tr>
<td>TIME</td>
<td>INT32</td>
<td><code>io.debezium.time.Time</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>String representation of a time without timezone information</td>
</tr>
</tbody>
</table>
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 6.8. Mappings when time.precision.mode is adaptive**

<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>io.debezium.time.Date</td>
</tr>
<tr>
<td>TIME(0), TIME(1), TIME(2), TIME(3)</td>
<td>INT32</td>
<td>io.debezium.time.Time</td>
</tr>
</tbody>
</table>

Represents the number of milliseconds past midnight, and does not include timezone information.
<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
</table>
| 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 6.9. Mappings when `time.precision.mode` is connect
<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>INT32</td>
<td>org.apache.kafka.connect.data.Date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of days since the epoch.</td>
</tr>
<tr>
<td>TIME([P])</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds since the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Represents the number of milliseconds since the epoch, and does not include timezone information.</td>
</tr>
<tr>
<td>DATETIME2</td>
<td>INT64</td>
<td>org.apache.kafka.connect.data.Timestamp</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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 6.10. Decimal types**

<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
</table>
### 6.5. Setting Up DB2 to Run aDebezium 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 6.5.1, “Configuring Db2 tables for change data capture”
- Section 6.5.2, “Effect of Db2 capture agent configuration on server load and latency”
- Section 6.5.3, “Db2 capture agent configuration parameters”

<table>
<thead>
<tr>
<th>Db2 data type</th>
<th>Literal type (schema type)</th>
<th>Semantic type (schema name) and Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC[(P[,S])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <strong>scale</strong> schema parameter contains an integer that represents how many digits the decimal point is shifted. The <strong>connect.decimal.precision</strong> schema parameter contains an integer that represents the precision of the given decimal value.</td>
</tr>
<tr>
<td>DECIMAL[(P[,S])]</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <strong>scale</strong> schema parameter contains an integer that represents how many digits the decimal point is shifted. The <strong>connect.decimal.precision</strong> schema parameter contains an integer that represents the precision of the given decimal value.</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <strong>scale</strong> schema parameter contains an integer that represents how many digits the decimal point is shifted. The <strong>connect.decimal.precision</strong> schema parameter contains an integer that represents the precision of the given decimal value.</td>
</tr>
<tr>
<td>MONEY</td>
<td>BYTES</td>
<td>org.apache.kafka.connect.data.Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <strong>scale</strong> schema parameter contains an integer that represents how many digits the decimal point is shifted. The <strong>connect.decimal.precision</strong> schema parameter contains an integer that represents the precision of the given decimal value.</td>
</tr>
</tbody>
</table>
### 6.5.1. Configuring Db2 tables for change data capture

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

**Prerequisites**

- You are logged in to Db2 as the `db2instl` user.
- On the Db2 host, the Debezium management UDFs are available in the `$HOME/asncdctools/src` directory. UDFs are available from the [Debezium examples repository](https://github.com/Debezium/examples).

**Procedure**

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

   ```bash
   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.

   ```bash
   db2 start db DB_NAME
   ```

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

   ```bash
   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:

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

   b. Restart the database:

      ```bash
      db2 restart db DB_NAME
      ```

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

   ```bash
   db2 connect to DB_NAME
   ```
6. Copy the Debezium management UDFs and set permissions for them:

   cp $HOME/asncdctools/src/asncdc $HOME/sqllib/function
   chmod 777 $HOME/sqllib/function

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

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

8. Create the ASN control tables:

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

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

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

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

10. Start the ASN agent:

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

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

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

12. Reinitialize the ASN service:

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

Additional resource

Reference table for Debezium Db2 management UDFs

### 6.5.2. 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.

6.5.3. 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.

6.6. 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:
6.6.1. 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.

- 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-26-rhel7:1.6.0 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
FROM registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/  
USER 1001
EOF
```

You can specify any file name that you want.
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:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
annotations:
  strimzi.io/use-connector-resources: "true"
spec:
  #...
  image: debezium-container-for-db2
```

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**

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector  
  labels:
    strimzi.io/cluster: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.db2.Db2Connector  
  tasksMax: 1
  config:
    database.hostname: 192.168.99.100
    database.port: 50000
    database.user: db2inst1
    database.password: Password!
    database.dbname: mydatabase
    database.server.name: fulfillment
    database.include.list: public.inventory
```

**Table 6.11. Descriptions of connector configuration settings**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The name of the connector when we register it with a Kafka Connect cluster.</td>
</tr>
<tr>
<td>2</td>
<td>The name of this Db2 connector class.</td>
</tr>
<tr>
<td>3</td>
<td>Only one task should operate at any one time.</td>
</tr>
<tr>
<td>4</td>
<td>The connector’s configuration.</td>
</tr>
<tr>
<td>5</td>
<td>The database host, which is the address of the Db2 instance.</td>
</tr>
<tr>
<td>6</td>
<td>The port number of the Db2 instance.</td>
</tr>
<tr>
<td>7</td>
<td>The name of the Db2 user.</td>
</tr>
</tbody>
</table>
### Item Description

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>The password for the Db2 user.</td>
</tr>
<tr>
<td>9</td>
<td>The name of the database to capture changes from.</td>
</tr>
<tr>
<td>10</td>
<td>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.</td>
</tr>
<tr>
<td>11</td>
<td>A list of all tables whose changes Debezium should capture.</td>
</tr>
</tbody>
</table>

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:

   ```bash
   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:

   ```bash
   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:

   ```text
   ... 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:

   ```bash
   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.
6.6.2. 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
- Pass-through configuration properties

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

Table 6.12. Required connector configuration properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>Unique name for the connector. Attempting to register again with the same name will fail. This property is required by all Kafka Connect connectors.</td>
</tr>
<tr>
<td>connector.class</td>
<td></td>
<td>The name of the Java class for the connector. Always use a value of <code>io.debezium.connector.db2.Db2Connector</code> or for the Db2 connector.</td>
</tr>
<tr>
<td>tasks.max</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>database.hostname</td>
<td></td>
<td>IP address or hostname of the Db2 database server.</td>
</tr>
<tr>
<td>database.port</td>
<td>50000</td>
<td>Integer port number of the Db2 database server.</td>
</tr>
<tr>
<td>database.user</td>
<td></td>
<td>Name of the Db2 database user for connecting to the Db2 database server.</td>
</tr>
<tr>
<td>database.password</td>
<td></td>
<td>Password to use when connecting to the Db2 database server.</td>
</tr>
<tr>
<td>database.dbname</td>
<td></td>
<td>The name of the Db2 database from which to stream the changes</td>
</tr>
</tbody>
</table>

CHAPTER 6. DEBEZIUM CONNECTOR FOR DB2

261
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database.server.name</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>database.history.kafka.topic</td>
<td></td>
<td>The full name of the Kafka topic where the connector stores the database schema history.</td>
</tr>
<tr>
<td>database.history.kafka.bootstrap.servers</td>
<td></td>
<td>A list of host/port pairs that the connector uses to establish an initial connection to the Kafka cluster. This connection is used for retrieving 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 Debezium Kafka Connect process.</td>
</tr>
<tr>
<td>table.include.list</td>
<td></td>
<td>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 <code>schemaName.tableName</code>. By default, the connector captures changes in every non-system table. Do not also set the <code>table.exclude.list</code> property.</td>
</tr>
<tr>
<td>table.exclude.list</td>
<td></td>
<td>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 <code>schemaName.tableName</code>. Do not also set the <code>table.include.list</code> property.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>column.exclude.list</code></td>
<td><code>empty string</code></td>
<td>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 <code>schemaName.tableName.columnName</code>. Primary key columns are always included in the event's key, even if they are excluded from the value.</td>
</tr>
<tr>
<td><code>column.mask.hash_.hash Algorithm_.with.salt_.salt</code></td>
<td><code>n/a</code></td>
<td>An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in change event values. A pseudonym is a field value that consists of the hashed value obtained by applying the <code>hashAlgorithm</code> algorithm and the <code>salt</code> salt that you specify in the property name.</td>
</tr>
</tbody>
</table>

Based on the hash algorithm applied, referential integrity is kept while data is masked. Supported hash algorithms are described in the `MessageDigest section` of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash value is automatically shortened to the length of the column.

You can specify multiple instances of this property with different algorithms and salts. Fully-qualified names for columns are of the form `schemaName.tableName.columnName`. For example:

```
column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = +
inventory.orders.customerName,
inventory.shipment.customerName
```

where `CzQMA0cB5K` is a randomly selected salt.
Depending on the `hashAlgorithm` used, the `salt` selected, and the actual data set, the field value may not be completely masked.
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time.precision.mode</code></td>
<td><code>adaptive</code></td>
<td>Time, date, and timestamps can be represented with different kinds of precision:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>adaptive</strong> 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>connect</strong> 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.</td>
</tr>
<tr>
<td><code>tombstones.on.delete</code></td>
<td><code>true</code></td>
<td>Controls whether a tombstone event should be generated after a delete event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>true</strong> - delete operations are represented by a delete event and a subsequent tombstone event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>false</strong> - only a delete event is sent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>After a delete operation, emitting a tombstone event enables Kafka to delete all change event records that have the same key as the deleted row.</td>
</tr>
<tr>
<td><code>include.schema.changes</code></td>
<td><code>true</code></td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>column.truncate.to._length.chars</code></td>
<td>n/a</td>
<td>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 <code>schemaName.tableName.columnName</code>. In change event records, values in these columns are truncated if they are longer than the number of characters specified by <code>length</code> in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, <code>column.truncate.to.20.chars</code>.</td>
</tr>
<tr>
<td><code>column.mask.with._length.chars</code></td>
<td>n/a</td>
<td>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 <code>schemaName.tableName.columnName</code>. In change event values, the values in the specified table columns are replaced with <code>length</code> 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.</td>
</tr>
</tbody>
</table>
| `column.propagate.source.type` | n/a | An optional, comma-separated list of regular expressions that match the fully-qualified names of columns. Fully-qualified names for columns are of the form `databaseName.tableName.columnName`, or `databaseName.schemaName.tableName.columnName`. For each specified column, the connector adds the column's original type and original length as parameters to the corresponding field schemas in the emitted change records. The following added schema parameters propagate the original type name and also the original length for variable-width types: 

```sql
__debezium.source.column.type +
__debezium.source.column.length +
__debezium.source.column.scale
```

This property is useful for properly sizing corresponding columns in sink databases. |
<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| `datatype.propagate.source.type` | n/a     | An optional, comma-separated list of regular expressions that match the database-specific data type name for some columns. Fully-qualified data type names are of the form `databaseNametableName.typeName`, or `databaseName.schemaName$tableName.typeName`. For these data types, the connector adds parameters to the corresponding field schemas in emitted change records. The added parameters specify the original type and length of the column: 

`__debezium.source.column.type + __debezium.source.column.length + __debezium.source.column.scale`

These parameters propagate a column’s original type name and length, for variable-width types, respectively. This property is useful for properly sizing corresponding columns in sink databases. See [Db2 data types](#) for the list of Db2-specific data type names. |
**message.key.columns**  
*empty string*

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.

Separate entries with semicolons. Insert a colon between the fully-qualified table name and its regular expression. The format is:

```
schema-name.table-name:_regexp_;...
```

For example,

```
schemaA.table_a:regex_1;schemaB.table_b:regex_2;schemaC.table_c:regex_3
```

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.

---

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 6.13. Advanced connector configuration properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>snapshot.mode</strong></td>
<td><strong>initial</strong></td>
<td>Specifies the criteria for performing a snapshot when the connector starts:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>initial</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>schema_only</strong> - 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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>snapshot.isolation.mode</td>
<td>repeatable_read</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>read_uncommitted</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>read_committed</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>repeatable_read</strong> - 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>exclusive</strong> - 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 <strong>exclusive</strong> mode guarantees full consistency; the initial snapshot and streaming logs constitute a linear history.</td>
</tr>
<tr>
<td>event.processing.failure_handling.mode</td>
<td>fail</td>
<td>Specifies how the connector handles exceptions during processing of events. The possible values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>fail</strong> - The connector logs the offset of the problematic event and stops processing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>warn</strong> - The connector logs the offset of the problematic event and continues processing with the next event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>skip</strong> - The connector skips the problematic event and continues processing with the next event.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>poll.interval.ms</td>
<td>1000</td>
<td>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.</td>
</tr>
<tr>
<td>max.queue.size</td>
<td>8192</td>
<td>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 <strong>max.queue.size</strong> value should always be larger than the value of the <strong>max.batch.size</strong> connector configuration property.</td>
</tr>
<tr>
<td>max.batch.size</td>
<td>2048</td>
<td>Positive integer value that specifies the maximum size of each batch of events that the connector processes.</td>
</tr>
<tr>
<td>max.queue.size.in.bytes</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>heartbeat.interval.ms</code></td>
<td>0</td>
<td>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.                                                                                               Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are useful when there are many updates in a database that is being tracked but only a tiny number of updates are in tables that are in capture mode. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that the connector has few opportunities to send the latest offset to Kafka. Sending heartbeat messages enables the connector to send the latest offset to Kafka.</td>
</tr>
<tr>
<td><code>heartbeat.topics.prefix</code></td>
<td><code>__debezium-heartbeat</code></td>
<td>Specifies the prefix for the name of the topic to which the connector sends heartbeat messages. The format for this topic name is <code>&lt;heartbeat.topics.prefix&gt;_&lt;server.name&gt;</code>.</td>
</tr>
<tr>
<td><code>snapshot.delay.ms</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>snapshot.fetch.size</code></td>
<td>2000</td>
<td>During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.</td>
</tr>
<tr>
<td>Property</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| `snapshot.lock.timeout.ms` | 10000   | Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this interval, the snapshot fails. Other possible settings are:   
|                           |         | 0 - The connector immediately fails when it cannot obtain a lock.  
|                           |         | -1 - The connector waits infinitely.                                                                                                                                  |
| `snapshot.select.statementoverrides` |         | 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 `schemaName.tableName`.  
|                           |         | For each table that you specify, also specify another configuration property: `snapshot.select.statementoverrides.SCHEMA_NAME.TABLE_NAME`. For example: `snapshot.select.statementoverrides.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.  
|                           |         | 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.                                                                 |
| `sanitize.field.names`    | true    | Indicates whether field names are sanitized to adhere to Avro naming requirements.                                                                                                                                 |
|                           | false   | if not.                                                                                                                                                                                                  |
### Property Configuration Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>provide.transaction.metadata</code></td>
<td><code>false</code></td>
<td>Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify <code>true</code> if you want the connector to do this. See <a href="#">Transaction metadata</a> for details.</td>
</tr>
</tbody>
</table>

#### Pass-through connector configuration properties

The connector also supports *pass-through* configuration properties that it uses when it creates Kafka producers and consumers:

- All connector configuration properties that begin with the `database.history.producer.` prefix are used (without the prefix) when creating the Kafka producer that writes to the database history topic.
- All connector configuration properties that begin with the `database.history.consumer.` prefix are used (without the prefix) when creating the Kafka consumer that reads the database history when the connector starts.

For example, the following connector configuration properties secure connections to the Kafka broker:

```plaintext
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
```

Be sure to consult the [Kafka documentation](#) for all of the configuration properties for Kafka producers and consumers. Note that the Db2 connector uses the new [consumer](#).

Also, the connector passes configuration properties that start with `database.` to the JDBC URL, for example, `database.applicationName=debezium`.

### 6.7. 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.

### 6.7.1. Monitoring Debezium during snapshots of Db2 databases

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last snapshot event that the connector has read.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>string[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>int</td>
<td>The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>int</td>
<td>The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>TotalTableCount</td>
<td>int</td>
<td>The total number of tables that are being included in the snapshot.</td>
</tr>
<tr>
<td>RemainingTableCount</td>
<td>int</td>
<td>The number of tables that the snapshot has yet to copy.</td>
</tr>
<tr>
<td>SnapshotRunning</td>
<td>boolean</td>
<td>Whether the snapshot was started.</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnapshotAborted</td>
<td>boolean</td>
<td>Whether the snapshot was aborted.</td>
</tr>
<tr>
<td>SnapshotCompleted</td>
<td>boolean</td>
<td>Whether the snapshot completed.</td>
</tr>
<tr>
<td>SnapshotDurationInSeconds</td>
<td>long</td>
<td>The total number of seconds that the snapshot has taken so far, even if not complete.</td>
</tr>
<tr>
<td>RowsScanned</td>
<td>Map&lt;String, Long&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>long</td>
<td>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.</td>
</tr>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

### 6.7.2. Monitoring Debezium Db2 connector record streaming

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastEvent</td>
<td>string</td>
<td>The last streaming event that the connector has read.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastEvent</td>
<td>long</td>
<td>The number of milliseconds since the connector has read and processed the most recent event.</td>
</tr>
<tr>
<td>TotalNumberOfEventsSeen</td>
<td>long</td>
<td>The total number of events that this connector has seen since last started or reset.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NumberOfEventsFiltered</td>
<td>long</td>
<td>The number of events that have been filtered by include/exclude list filtering rules configured on the connector.</td>
</tr>
<tr>
<td>MonitoredTables</td>
<td>String[]</td>
<td>The list of tables that are monitored by the connector.</td>
</tr>
<tr>
<td>QueueTotalCapacity</td>
<td>Int</td>
<td>The length the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>QueueRemainingCapacity</td>
<td>Int</td>
<td>The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.</td>
</tr>
<tr>
<td>Connected</td>
<td>Boolean</td>
<td>Flag that denotes whether the connector is currently connected to the database server.</td>
</tr>
<tr>
<td>MilliSecondsBehindSource</td>
<td>Long</td>
<td>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.</td>
</tr>
<tr>
<td>NumberOfCommittedTransactions</td>
<td>Long</td>
<td>The number of processed transactions that were committed.</td>
</tr>
<tr>
<td>SourceEventPosition</td>
<td>Map&lt;String, String&gt;</td>
<td>The coordinates of the last received event.</td>
</tr>
<tr>
<td>LastTransactionId</td>
<td>String</td>
<td>Transaction identifier of the last processed transaction.</td>
</tr>
<tr>
<td>MaxQueueSizeInBytes</td>
<td>Long</td>
<td>The maximum buffer of the queue in bytes.</td>
</tr>
</tbody>
</table>
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentQueueSizeInBytes</td>
<td>long</td>
<td>The current data of records in the queue in bytes.</td>
</tr>
</tbody>
</table>

### 6.7.3. Monitoring Debezium Db2 connector schema history

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Status</strong></td>
<td>string</td>
<td>One of STOPPED, RECOVERING (recovering history from the storage), RUNNING describing the state of the database history.</td>
</tr>
<tr>
<td>RecoveryStartTime</td>
<td>long</td>
<td>The time in epoch seconds at what recovery has started.</td>
</tr>
<tr>
<td>ChangesRecovered</td>
<td>long</td>
<td>The number of changes that were read during recovery phase.</td>
</tr>
<tr>
<td>ChangesApplied</td>
<td>long</td>
<td>the total number of schema changes applied during recovery and runtime.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastRecoveredChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was recovered from the history store.</td>
</tr>
<tr>
<td>MilliSecondsSinceLastAppliedChange</td>
<td>long</td>
<td>The number of milliseconds that elapsed since the last change was applied.</td>
</tr>
<tr>
<td>LastRecoveredChange</td>
<td>string</td>
<td>The string representation of the last change recovered from the history store.</td>
</tr>
<tr>
<td>LastAppliedChange</td>
<td>string</td>
<td>The string representation of the last applied change.</td>
</tr>
</tbody>
</table>

### 6.8. 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 6.14. Descriptions of Debezium management UDFs**

<table>
<thead>
<tr>
<th>Task</th>
<th>Command and notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start the ASN agent</td>
<td>VALUES ASNCDC.ASNCDCSERVICES('start','asncdc');</td>
</tr>
<tr>
<td>Stop the ASN agent</td>
<td>VALUES ASNCDC.ASNCDCSERVICES('stop','asncdc');</td>
</tr>
<tr>
<td>Check the status of the ASN agent</td>
<td>VALUES ASNCDC.ASNCDCSERVICES('status','asncdc');</td>
</tr>
<tr>
<td>Put a table into capture mode</td>
<td>CALL ASNCDC.ADDTABLE('MYSHEMA', 'MYTABLE');</td>
</tr>
<tr>
<td></td>
<td>Replace <code>MYSHEMA</code> with the name of the schema that contains the table you want to put into capture mode. Likewise, replace <code>MYTABLE</code> with the name of the table to put into capture mode.</td>
</tr>
<tr>
<td>Remove a table from capture mode</td>
<td>CALL ASNCDC.REMOVETABLE('MYSHEMA', 'MYTABLE');</td>
</tr>
<tr>
<td>Reinitialize the ASN service</td>
<td>VALUES ASNCDC.ASNCDCSERVICES('reinit','asncdc');</td>
</tr>
<tr>
<td></td>
<td>Do this after you put a table into capture mode or after you remove a table from capture mode.</td>
</tr>
</tbody>
</table>

### 6.9. 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.

### 6.9.1. 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**.
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.

6.9.2. 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 7. 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.

7.1. 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

7.2. 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.

7.2.1. 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 value is used to specify the JVM parameter `-Dzookeeper.jmx.log4j.disable=$JMXLOG4J`.
7.2.2. 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`

7.2.3. 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`

7.3. MONITORING DEBEZIUM ON OPENSHIFT

If you are using Debezium on OpenShift, you can obtain JMX metrics by opening a JMX port on 9999. For more information, see [JMX Options](#) in Using AMQ Streams on OpenShift.

In addition, you can use Prometheus and Grafana to monitor the JMX metrics. For more information, see [Setting up metrics and dashboards for AMQ Streams](#), in Deploying and Upgrading AMQ Streams on OpenShift.
CHAPTER 8. 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.

8.1. 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.

8.2. 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:

```
log4j.rootLogger=INFO, stdout
```
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** appenders.

The **stdout** appenders write log messages to the console (as opposed to a file).

The **stdout** appenders use a pattern matching algorithm to format the log messages.

The pattern for the **stdout** appenders (see the Log4j documentation for details).

Unless you configure other loggers, all of the loggers that Debezium uses inherit the rootLogger configuration.

### 8.3. 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.

### 8.3.1. 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.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n
```

Red Hat Integration 2021.Q1 Debezium User Guide

284
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:

      ```properties
      log4j.logger.io.debezium.connector.mysql=INFO, stdout
      log4j.logger.io.debezium.connector.mysql.BinlogReader=DEBUG, stdout
      ...  
      ```

3.3.2. 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'
   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,696 INFO   MySQL|dbserver1|snapshot  Snapshot is using user
   'debezium' with these MySQL grants:
   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,697 INFO   MySQL|dbserver1|snapshot  GRANT SELECT, RELOAD,
   SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO
   'debezium'@'%' [io.debezium.connector.mysql.SnapshotReader]
   ...
   ```
Each line in the log includes the connector type (for example, MySQL), the name of the connector (for example, dbserver1), and the activity of the thread (for example, snapshot).

8.4. DEBEZIUM LOGGING ON OPENSSHIFT

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 9. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION

When default Debezium connector behavior is not right for your application, you can use the following Debezium features to configure the behavior you need.

**Topic router SMT**
Single message transform that re-routes data change event records to topics that you specify.

**Content-based router SMT**
Single message transform that evaluates data change event record content and re-routes event records to particular topics according to content.

**Kafka Connect automatic topic creation**
Enables Connect to create topics at runtime, and apply configuration settings to those topics based on their names.

**Filter SMT**
Single message transform that uses an expression that you specify to evaluate data change event records. The connector streams only those events that evaluate to true.

**Event flattening SMT**
Single message transform that flattens the complex structure of a data change event record into the simplified format that might be required by some Kafka consumers.

**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.

**Outbox event router SMT**
Single message transform that provides support for the outbox pattern.

**CloudEvents converter**
Enables a Debezium connector to emit change event records that conform to the CloudEvents specification.

### 9.1. 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 9.1.1, “Use case for routing Debezium records to topics that you specify”
- Section 9.1.2, “Example of routing Debezium records for multiple tables to one topic”
- Section 9.1.3, “Ensuring unique keys across Debezium records routed to the same topic”
- Section 9.1.4, “Options for configuring Debezium topic routing transformation”

### 9.1.1. 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. Logical tables 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.

### 9.1.2. 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:

```properties
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.

### 9.1.3. 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:

```plaintext
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:

```plaintext
transforms.Reroute.key.field.regex=(.*)customers\_shard(.*)
transforms.Reroute.key.field.replacement=$2
```

With this configuration, suppose that the default destination topic names are:
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`:

```ini
... transforms.Reroute.key.enforce.uniqueness=false ...
```

### 9.1.4. Options for configuring Debezium topic routing transformation

The following table describes topic routing SMT configuration options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>topic.regex</code></td>
<td></td>
<td>Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.</td>
</tr>
<tr>
<td><code>topic.replacement</code></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td><code>key.enforce.uniqueness</code></td>
<td><code>true</code></td>
<td>Indicates whether to add a field to the record’s change event key. Adding a key field ensures that each event key is unique across the tables whose change event records go to the same topic. This helps to prevent collisions of change events for records that have the same key but that originate from different source tables. Specify <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.</td>
</tr>
</tbody>
</table>
### 9.2. 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.

9.2.1. 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-1.4.2.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-1.4.2.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.

9.2.2. 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:

```bash
... transforms=route transforms.route.type=io.debezium.transforms.ContentBasedRouter
```
The preceding example specifies the use of the Groovy expression language.

Records that do not match the pattern are routed to the default topic.

9.2.3. 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 9.2. Content-based routing expression variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>A key of the message.</td>
<td>org.apache.kafka.connect.data.Struct</td>
</tr>
<tr>
<td>value</td>
<td>A value of the message.</td>
<td>org.apache.kafka.connect.data.Struct</td>
</tr>
<tr>
<td>keySchema</td>
<td>Schema of the message key.</td>
<td>org.apache.kafka.connect.data.Schema</td>
</tr>
<tr>
<td>valueSchema</td>
<td>Schema of the message value.</td>
<td>org.apache.kafka.connect.data.Schema</td>
</tr>
<tr>
<td>topic</td>
<td>Name of the target topic.</td>
<td>String</td>
</tr>
<tr>
<td>headers</td>
<td>A Java map of message headers. The key field is the header name. The headers variable exposes the following properties:</td>
<td>java.util.Map&lt;String, io.debezium.transforms.scripting.RecordHeader&gt;</td>
</tr>
</tbody>
</table>

- **value** (of type `Object`)
- **schema** (of type `org.apache.kafka.connect.data.Schema`)

An expression can invoke arbitrary methods on its variables. Expressions should resolve to a Boolean value that determines how the SMT dispositions the message. When the routing condition in an expression evaluates to `true`, the message is retained. When the routing condition evaluates to `false`, the message is removed.

Expressions should not result in any side effects. That is, they should not modify any variables that they pass.
9.2.4. 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:

```java
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:

```javascript
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:

```javascript
value.op == 'u' ? 'updates' : null
```

9.2.5. Options for configuring the content-based routing transformation

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic.regex</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

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.

Specifies how the transformation handles null (tombstone) messages. You can specify one of the following options:

- keep (Default) Pass the messages through.
- drop Remove the messages completely.
- evaluate Apply the condition logic to the messages.

9.3. 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 9.3.1, “Disabling automatic topic creation for the Kafka broker”
- Section 9.3.2, “Configuring automatic topic creation in Kafka Connect”
- Section 9.3.3, “Configuration of automatically created topics”
- Section 9.3.3.1, “Topic creation groups”
- Section 9.3.3.2, “Topic creation group configuration properties”
- Section 9.3.3.3, “Specifying the configuration for the Debezium default topic creation group”
- Section 9.3.3.4, “Specifying the configuration for Debezium custom topic creation groups”
- Section 9.3.3.5, “Registering Debezium custom topic creation groups”

9.3.1. 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 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`.

9.3.2. 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:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
customapi: 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.

### 9.3.3. 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.

#### 9.3.3.1. 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.

#### 9.3.3.2. 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.

9.3.3.3. 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:

  ```yaml
  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:

  ```yaml
  apiVersion: kafka.strimzi.io/v1beta1
  kind: KafkaConnector
  metadata:
    name: inventory-connector
    labels:
      strimzi.io/cluster: my-connect-cluster
  spec:
    ...
  
  config:
    ...
    topic.creation.default.replication.factor: 3
    topic.creation.default.partitions: 10
    topic.creation.default.cleanup.policy: compact
    topic.creation.default.compression.type: lz4
    ...
  ```

You can include any Kafka topic-level configuration property in the configuration for the default group.

Table 9.3. Connector configuration for the default topic creation group
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>topic.creation.default.replication.factor</strong> defines the replication factor for topics created by the default group. &lt;br&gt;<strong>replication.factor</strong> is mandatory for the <strong>default</strong> group but optional for custom groups. Custom groups will fall back to the <strong>default</strong> group’s value if not set. Use <code>-1</code> to use the Kafka broker’s default value.</td>
</tr>
<tr>
<td>2</td>
<td><strong>topic.creation.default.partitions</strong> defines the number of partitions for topics created by the default group. &lt;br&gt;<strong>partitions</strong> is mandatory for the <strong>default</strong> group but optional for custom groups. Custom groups will fall back to the <strong>default</strong> group’s value if not set. Use <code>-1</code> to use the Kafka broker’s default value.</td>
</tr>
<tr>
<td>3</td>
<td><strong>topic.creation.default.cleanup.policy</strong> is mapped to the <strong>cleanup.policy</strong> property of the <strong>topic level configuration parameters</strong> and defines the log retention policy.</td>
</tr>
<tr>
<td>4</td>
<td><strong>topic.creation.default.compression.type</strong> is mapped to the <strong>compression.type</strong> property of the <strong>topic level configuration parameters</strong> and defines how messages are compressed on hard disk.</td>
</tr>
</tbody>
</table>

**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.

### 9.3.3.4. 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**:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector
...
spec:
...
config:
...
  topic.creation.inventory.include: dbserver1\.inventory\.*
  topic.creation.inventory.partitions: 20
  topic.creation.inventory.cleanup.policy: compact
```
Table 9.4. Connector configuration for custom inventory and application logs topic creation groups

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>

| 2    | **topic.creation.inventory.include** defines a regular expression to match all topics that start with dbserver1.inventory. The configuration that is defined for the inventory group is applied only to topics with names that match the specified regular expression. |

| 3    | Defines the configuration for the applicationlogs group. The replication.factor and partitions properties are optional for custom groups. If no value is set, custom groups fall back to the value set for the default group. Set the value to -1 to use the value that is set for the Kafka broker. |

| 4    | **topic.creation.applicationlogs.include** defines a regular expression to match all topics that start with dbserver1.logs.applog-. The configuration that is defined for the applicationlogs group is applied only to topics with names that match the specified regular expression. Because an exclude property is also defined for this group, the topics that match the include regular expression might be further restricted by the that exclude property. |

| 5    | **topic.creation.applicationlogs.exclude** defines a regular expression to match all topics that start with dbserver1.logs.applog-old-. The configuration that is defined for the applicationlogs group is applied only to topics with names that start with 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. |

9.3.3.5. 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:

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector

spec:
  config:
    topic.creation.groups: inventory,applicationlogs
```

**Completed configuration**

The following example shows a completed configuration that includes the configuration for a default topic group, along with the configurations for an inventory and an applicationlogs custom topic creation group:

**Example: Configuration for a default topic creation group and two custom groups**

```yaml
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector

spec:
  config:
    topic.creation.default.replication.factor: 3,
    topic.creation.default.partitions: 10,
    topic.creation.default.cleanup.policy: compact
    topic.creation.default.compression.type: lz4
    topic.creation.groups: inventory,applicationlogs
    topic.creation.inventory.include: dbserver1\.inventory\.*
    topic.creation.inventory.partitions: 20
    topic.creation.inventory.cleanup.policy: compact
    topic.creation.inventory.delete.retention.ms: 7776000000
    topic.creation.applicationlogs.include: dbserver1\.logs\applog\.*
    topic.creation.applicationlogs.exclude: dbserver1\.logs\applog-old\.*
    topic.creation.applicationlogs.replication.factor: 1
    topic.creation.applicationlogs.partitions: 20
    topic.creation.applicationlogs.cleanup.policy: delete
    topic.creation.applicationlogs.retention.ms: 7776000000
    topic.creation.applicationlogs.compression.type: lz4
```

**9.4. 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.

### 9.4.1. 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-1.4.2.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-1.4.2.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.

9.4.2. 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.

```groovy
... 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.

9.4.3. 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>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>A key of the message.</td>
<td>org.apache.kafka.connect.data.Struct</td>
</tr>
<tr>
<td>value</td>
<td>A value of the message.</td>
<td>org.apache.kafka.connect.data.Struct</td>
</tr>
<tr>
<td>keySchema</td>
<td>Schema of the message key.</td>
<td>org.apache.kafka.connect.data.Schema</td>
</tr>
<tr>
<td>valueSchema</td>
<td>Schema of the message value.</td>
<td>org.apache.kafka.connect.data.Schema</td>
</tr>
<tr>
<td>topic</td>
<td>Name of the target topic.</td>
<td>String</td>
</tr>
</tbody>
</table>
### Name | Description | Type
--- | --- | ---
headers | A Java map of message headers. The key field is the header name. The `headers` variable exposes the following properties:
- **value** (of type `Object`)
- **schema** (of type `io.debezium.transforms.scripting.RecordHeader`)

An expression can invoke arbitrary methods on its variables. Expressions should resolve to a Boolean value that determines how the SMT dispositions the message. When the filter condition in an expression evaluates to **true**, the message is retained. When the filter condition evaluates to **false**, the message is removed.

Expressions should not result in any side-effects. That is, they should not modify any variables that they pass.

### 9.4.4. 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:

\[
\text{value.op} == \text{'u'} && \text{value.before.id} == 2
\]

### 9.4.5. Options for configuring filter transformation

The following table lists the configuration options that you can use with the filter SMT.

**Table 9.6. filter SMT configuration options**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>topic.regex</strong></td>
<td></td>
<td>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 <strong>topic.regex</strong>, 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 <strong>topic.regex</strong>, the SMT passes the event to the topic unmodified.</td>
</tr>
<tr>
<td><strong>language</strong></td>
<td></td>
<td>The language in which the expression is written. Must begin with <strong>jsr223</strong>., for example, <strong>jsr223.groovy</strong>, or <strong>jsr223.graal.js</strong>. Debezium supports bootstrapping through the JSR 223 API (&quot;Scripting for the Java™ Platform&quot;) only.</td>
</tr>
<tr>
<td><strong>condition</strong></td>
<td></td>
<td>The expression to be evaluated for every message. Must evaluate to a Boolean value where a result of <strong>true</strong> keeps the message, and a result of <strong>false</strong> removes it.</td>
</tr>
</tbody>
</table>
| **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. |

### 9.5. 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 9.5.1, “Description of Debezium change event structure”
- Section 9.5.2, “Behavior of Debezium event flattening transformation”
- Section 9.5.3, “Configuration of Debezium event flattening transformation”
- Section 9.5.4, “Example of adding Debezium metadata to the Kafka record”
- Section 9.5.5, “Options for configuring Debezium event flattening transformation”

9.5.1. 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:

```json
{
  "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:

```json
{
  "field1" : "newvalue1",
  "field2" : "newvalue2"
}
```

To provide the needed Kafka record format for consumers, configure the event flattening SMT.

9.5.2. 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.

9.5.3. 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:
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.

9.5.4. 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:

```json
{
  ...
  "__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`.

### 9.5.5. Options for configuring Debezium event flattening transformation

The following table describes the options that you can specify to configure the event flattening SMT.

**Table 9.7. Descriptions of event flattening SMT configuration options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drop.tombstones</code></td>
<td><code>true</code></td>
<td>Debezium generates a tombstone record for each <code>DELETE</code> 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>.</td>
</tr>
</tbody>
</table>
Debezium generates a change event record for each **DELETE** operation. The default behavior is that event flattening SMT removes these records from the stream. To keep Kafka records for **DELETE** operations in the stream, set `delete.handling.mode` to `none` or `rewrite`.

Specify **none** to keep the change event record in the stream. The record contains only "value": "null".

Specify **rewrite** to keep the change event record in the stream and edit the record to have a value field that contains the key/value pairs that were in the `before` field and also add `__deleted: true` to the value. This is another way to indicate that the record has been deleted.

When you specify **rewrite**, the updated simplified records for **DELETE** operations might be all you need to track deleted records. You can consider accepting the default behavior of dropping the tombstone records that the Debezium connector creates.

### Option: `delete.handling.mode`

<table>
<thead>
<tr>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop</td>
<td>Debezium generates a change event record for each <strong>DELETE</strong> operation. The default behavior is that event flattening SMT removes these records from the stream. To keep Kafka records for <strong>DELETE</strong> operations in the stream, set <code>delete.handling.mode</code> to <code>none</code> or <code>rewrite</code>. Specify <strong>none</strong> to keep the change event record in the stream. The record contains only &quot;value&quot;: &quot;null&quot;. Specify <strong>rewrite</strong> 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 <code>before</code> field and also add <code>__deleted: true</code> to the value. This is another way to indicate that the record has been deleted. When you specify <strong>rewrite</strong>, the updated simplified records for <strong>DELETE</strong> 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.</td>
</tr>
</tbody>
</table>

### Option: `route.by.field`

To use row data to determine the topic to route the record to, set this option to an `after` field attribute. The SMT routes the record to the topic whose name matches the value of the specified `after` field attribute. For a **DELETE** operation, set this option to a `before` field attribute.

For example, configuration of `route.by.field=destination` routes records to the topic whose name is the value of `after.destination`. The default behavior is that a Debezium connector sends each change event record to a topic whose name is formed from the name of the database and the name of the table in which the change was made.

If you are configuring the event flattening SMT on a sink connector, setting this option might be useful when the destination topic name dictates the name of the database table that will be updated with the simplified change event record. If the topic name is not correct for your use case, you can configure `route.by.field` to re-route the event.

### Option: `route.by.field`
<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add.fields.prefix</code></td>
<td><code>__</code> (double-underscore)</td>
<td>Set this optional string to prefix a field.</td>
</tr>
<tr>
<td>Option</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>add.fields</strong></td>
<td></td>
<td>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 <code>source.ts_ms</code>. Optionally, you can override the field name via <code>&lt;field name&gt;:&lt;new field name&gt;</code>, e.g. like so: new field name like <code>version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP</code>. Please note that the <strong>new field name</strong> is case-sensitive. When the SMT adds metadata fields to the simplified record's value, it prefixes each metadata field name with a double underscore. For a struct specification, the SMT also inserts an underscore between the struct name and the field name. If you specify a field that is not in the change event record, the SMT still adds the field to the record's value.</td>
</tr>
<tr>
<td><strong>add.headers.prefix</strong></td>
<td>__ (double-underscore)</td>
<td>Set this optional string to prefix a header.</td>
</tr>
</tbody>
</table>
add.headers

Set this option to a comma-separated list, with no spaces, of metadata fields to add to the header of the simplified Kafka record. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field, for example `source.ts_ms`.

Optionally, you can override the field name via `<field name>:<new field name>`, e.g. like so: new field name like `version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP`.

Please note that the **new field name** is case-sensitive.

When the SMT adds metadata fields to the simplified record’s header, it prefixes each metadata field name with a double underscore. For a struct specification, the SMT also inserts an underscore between the struct name and the field name.

If you specify a field that is not in the change event record, the SMT does not add the field to the header.

### 9.6. CONFIGURING DEBEZIUM CONNECTORS TO USE AVRO Serialization

A Debezium connector works in the Kafka Connect framework to capture each row-level change in a database by generating a change event record. For each change event record, the Debezium connector completes the following actions:

1. Applies configured transformations.
2. Serializes the record key and value into a binary form by using the configured Kafka Connect converters.
3. Writes the record to the correct Kafka topic.

You can specify converters for each individual Debezium connector instance. Kafka Connect provides a JSON converter that serializes the record keys and values into JSON documents. The default behavior is that the JSON converter includes the record’s message schema, which makes each record very verbose. The Getting Started with Debezium guide shows what the records look like when both payload and schemas are included. If you want records to be serialized with JSON, consider setting the following connector configuration properties to false:

- `key.converter.schemas.enable`
- `value.converter.schemas.enable`
Setting these properties to false excludes the verbose schema information from each record.

Alternatively, you can serialize the record keys and values by using Apache Avro. The Avro binary format is compact and efficient. Avro schemas make it possible to ensure that each record has the correct structure. Avro’s schema evolution mechanism enables schemas to evolve. This is essential for Debezium connectors, which dynamically generate each record’s schema to match the structure of the database table that was changed. Over time, change event records written to the same Kafka topic might have different versions of the same schema. Avro serialization makes it easier for the consumers of change event records to adapt to a changing record schema.

To use Apache Avro serialization, you must deploy a schema registry that manages Avro message schemas and their versions. For information about setting up this registry, see the documentation for Red Hat Integration - Service Registry.

9.6.1. 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.

9.6.2. 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 Debezium Service Registry Kafka Connect zip file and extracting it into the Debezium connector’s directory.
3. Configure a Debezium connector instance to use Avro serialization by setting configuration properties as follows:

```java
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.

### 9.6.3. 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, Installing Service Registry from the OpenShift OperatorHub](#), which provides instructions for:
   - Installing AMQ Streams
   - Setting up AMQ Streams storage
   - Installing Service Registry

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:

```bash
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-26-rhel7:1.6.0 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-25:1.5.0
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/v1beta1
da: 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:

   ```yaml
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnector
   metadata:
     name: inventory-connector
     labels:
       strimzi.io/cluster: my-connect-cluster
   spec:
     class: io.debezium.connector.mysql.MySqlConnector
     tasksMax: 1
     config:
       database.hostname: mysql
       database.port: 3306
       database.user: debezium
       database.password: dbz
       database.server.id: 184054
       database.server.name: dbserver1
       database.include.list: inventory
       database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092
       database.history.kafka.topic: schema-changes.inventory
       key.converter: io.apicurio.registry.utils.converter.AvroConverter
       key.converter.apicurio.registry.url: http://apicurio:8080/api
       key.converter.apicurio.registry.global-id:
       io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
       value.converter: io.apicurio.registry.utils.converter.AvroConverter
       value.converter.apicurio.registry.url: http://apicurio:8080/api
       value.converter.apicurio.registry.global-id:
       io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
   ``

   b. Apply the connector instance, for example:

   ```bash
   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:

   ```bash
   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:

   ```plaintext
   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-
   ```
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 17:57:34,424 INFO Completed snapshot in 00:00:03.632 (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-snapshot]  
...  
2020-02-21 17:57:34,775 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]  
...
9.6.4. 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.

9.7. 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 in only an outbox table. 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 9.7.1, “Example of a Debezium outbox message”
- Section 9.7.2, “Outbox table structure expected by Debezium outbox event router SMT”
9.7.1. 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,  
"\lineltems": [{"\id": 1, "\item": "Debezium in Action", "\status": "ENTERED",  
"\quantity": 2,  
"\totalPrice": 39.98},  
{"\id": 2, "\item": "Debezium for Dummies",  
"\status": "ENTERED",  
"\quantity": 1,  
"\totalPrice": 29.99}],  
"orderDate": "2019-01-31T12:13:01",  
"customerId": 123}  
```

A Debezium connector that is configured to apply the outbox event router SMT generates the above message by transforming a Debezium raw message like this:

```
# Kafka Message key: "406c07f3-26f0-4eea-a50c-109940064b8f"
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484

{  
"before": null,  
"after": {  
"id": "406c07f3-26f0-4eea-a50c-109940064b8f",  
"aggregateid": "1",  
"aggregatetype": "Order",  
"payload": "{\"\id\": 1,  
\"\lineltems\": [{\"\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": "0.9.3.Final",  
"connector": "postgresql",  
"name": "dbserver1-bare",  
"db": "orderdb",  
"ts_usec": 155689029448870,  
"txId": 584,  
"lsn": 24064704,  
"schema": "inventory",  
"table": "outboxevent",  
"snapshot": false,  
"last_snapshot_record": null
```
This example of a Debezium outbox message is based on the default outbox event router configuration, which assumes an outbox table structure and event routing based on aggregates. To customize behavior, the outbox event router SMT provides numerous configuration options.

### 9.7.2. 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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>uuid</td>
<td>not null</td>
</tr>
<tr>
<td>aggregatetype</td>
<td>character varying(255)</td>
<td>not null</td>
</tr>
<tr>
<td>aggregateid</td>
<td>character varying(255)</td>
<td>not null</td>
</tr>
<tr>
<td>type</td>
<td>character varying(255)</td>
<td>not null</td>
</tr>
<tr>
<td>payload</td>
<td>jsonb</td>
<td></td>
</tr>
</tbody>
</table>

**Table 9.8. Descriptions of expected outbox table columns**

<table>
<thead>
<tr>
<th>Column</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>To obtain the unique ID of the event from a different outbox table column, set the <code>table.field.event.id</code> SMT option in the connector configuration.</td>
</tr>
<tr>
<td>aggregatetype</td>
<td>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>$\{routedByValue\}</code> variable in the <code>route.topic.replacement</code> SMT option.</td>
</tr>
<tr>
<td></td>
<td>For example, in a default configuration, the <code>route.by.field</code> SMT option is set to <code>aggregatetype</code> and the <code>route.topic.replacement</code> SMT option is set to <code>outbox.event.$\{routedByValue\}</code>. Suppose that your application adds two records to the outbox table. In the first record, the value in the <code>aggregatetype</code> column is <code>customers</code>. In the second record, the value in the <code>aggregatetype</code> column is <code>orders</code>. 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. To obtain this value from a different outbox table column, set the <code>route.by.field</code> SMT option in the connector configuration.</td>
</tr>
</tbody>
</table>
aggregateid

Contains the event key, which provides an ID for the payload. The SMT uses this value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions.

To obtain the event key from a different outbox table column, set the `table.field.event.key` SMT option in the connector configuration.

type

A user-defined value that helps categorize or organize events.

payload

The representation of the event itself. The default structure is JSON. The content in this field becomes one of these:

- 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.

To obtain the event payload from a different outbox table column, set the `table.field.event.payload` SMT option in the connector configuration.

9.7.3. 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:

```properties
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
```

9.7.4. 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:

```properties
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.

### 9.7.5. 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
```

### 9.7.6. 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 9.9. Descriptions of outbox event router SMT configuration options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table.field.event.id</td>
<td>id</td>
<td>Table</td>
<td>Specifies the outbox table column that contains the unique event ID.</td>
</tr>
<tr>
<td>table.field.event.key</td>
<td>aggregateid</td>
<td>Table</td>
<td>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.</td>
</tr>
<tr>
<td>Option</td>
<td>Default</td>
<td>Group</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>table.field.event.timestamp</td>
<td></td>
<td>Table</td>
<td>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.</td>
</tr>
<tr>
<td>table.field.event.payload</td>
<td>payload</td>
<td>Table</td>
<td>Specifies the outbox table column that contains the event payload.</td>
</tr>
<tr>
<td>table.field.event.payload.id</td>
<td>aggregateid</td>
<td>Table</td>
<td>Specifies the outbox table column that contains the payload ID.</td>
</tr>
<tr>
<td>table.fields.additional.placement</td>
<td></td>
<td>Table, Envelope</td>
<td>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: <code>id:header,my-field:envelope</code> To specify an alias for the column, specify a trio with the alias as the third value, for example: <code>id:header,my-field:envelope:my-alias</code> The second value is the placement and it must always be header or envelope. Configuration examples are in emitting additional fields in Debezium outbox messages.</td>
</tr>
<tr>
<td>table.field.event.schema.version</td>
<td></td>
<td>Table, Schema</td>
<td>When set, this value is used as the schema version as described in the Kafka Connect Schema Javadoc.</td>
</tr>
<tr>
<td>Option</td>
<td>Default</td>
<td>Group</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>route.by.field</strong></td>
<td>aggregatetype</td>
<td>Router</td>
<td>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.</td>
</tr>
<tr>
<td><strong>route.topic.regex</strong></td>
<td>(?&lt;routedByValue&gt;.* )</td>
<td>Router</td>
<td>Specifies a regular expression that the outbox SMT applies in the RegexRouter to outbox table records. This regular expression is part of the setting of the route.topic.replacement SMT option. The default behavior is that the SMT replaces the default ${routedByValue} variable in the setting of the route.topic.replacement SMT option with the setting of the route.by.field outbox SMT option.</td>
</tr>
</tbody>
</table>
| **route.topic.replacement**  | outbox.event.${routedByValue} | Router  | Specifies the name of the topic to which the connector emits outbox messages. The default topic name is outbox.event. followed by the aggregatetype column value in the outbox table record. For example, if the aggregatetype value is customers, the topic name is outbox.event.customers. To change the topic name, you can:  
  - Set the route.by.field option to a different column.  
  - Set the route.topic.regex option to a different regular expression. |
| **route.tombstone.on.empty.payload** | false | Router  | Indicates whether an empty or null payload causes the connector to emit a tombstone event.                                                                                                                    |
Debezium Determines the behavior of the SMT when there is an `UPDATE` operation on the outbox table. Possible settings are:

- **warn** - The SMT logs a warning and continues to the next outbox table record.
- **error** - The SMT logs an error and continues to the next outbox table record.
- **fatal** - The SMT logs an error and the connector stops processing.

All changes in an outbox table are expected to be `INSERT` operations. That is, an outbox table functions as a queue; updates to records in an outbox table are not allowed. The SMT automatically filters out `DELETE` operations on an outbox table.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>debezium.op.invalid.behavior</code></td>
<td><code>warn</code></td>
<td>Debezium</td>
<td>Determines the behavior of the SMT when there is an <code>UPDATE</code> operation on the outbox table. Possible settings are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>warn</strong> - The SMT logs a warning and continues to the next outbox table record.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>error</strong> - The SMT logs an error and continues to the next outbox table record.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- <strong>fatal</strong> - The SMT logs an error and the connector stops processing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tracing.span.context.field</code></td>
<td><code>tracingspancontext</code></td>
<td>Tracing</td>
<td>The name of the field containing tracing span context.</td>
</tr>
<tr>
<td><code>tracing.operation.name</code></td>
<td><code>debezium-read</code></td>
<td>Tracing</td>
<td>The operation name representing the Debezium processing span.</td>
</tr>
<tr>
<td><code>tracing.with.context.field.only</code></td>
<td><code>false</code></td>
<td>Tracing</td>
<td>When true only events that have serialized context field should be traced.</td>
</tr>
</tbody>
</table>

**Distributed tracing**

The extension has support for the distributed tracing. See [tracing documentation](#) for more details.

### 9.8. 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.
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 9.8.1, “Example Debezium change event records in CloudEvents format”
- Section 9.8.2, “Example of configuring Debezium CloudEvents converter”
- Section 9.8.3, “Debezium CloudEvents converter configuration options”

For information about using Avro, see:

- Avro serialization
- Apicurio Registry

**9.8.1. 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", 
}
```
"datacontenttype" : "application/json",
"iodebeziumop" : "r",
"iodebeziumversion" : "1.4.2.Final",
"iodebeziumconnector" : "postgresql",
"iodebeziumname" : "test_server",
"iodebeziumtsms" : "1578923739738",
"iodebeziumsnapshot" : "true",
"iodebeziumdb" : "postgres",
"iodebeziumschema" : "s1",
"iodebeziumentable" : "a",
"iodebeziumentxl" : "565",
"iodebeziumentlsn" : "29274832",
"iodebeziumentxmin" : null,
"iodebeziumtxid" : "565",
"iodebeziumtxtotalorder" : "1",
"iodebeziumtxdatacollectionorder" : "1",
"data" : {
  "before" : null,
  "after" : {
    "pk" : 1,
    "name" : "Bob"
  }
}

1 Unique ID that the connector generates for the change event based on the change event’s content.
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 The CloudEvents specification version.
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 Time of the change in the source database.
6 Describes the content type of the data attribute, which is JSON in this example. The only 
alternative is Avro.
7 An operation identifier. Possible values are r for read, c for create, u for update, or d for delete.
8 All source attributes that are known from Debezium change events are mapped to CloudEvents 
extension attributes by using the iodebezium prefix for the attribute name.
9 When enabled in the connector, each transaction attribute that is known from Debezium change 
events is mapped to a CloudEvents extension attribute by using the iodebeziumtx prefix for the 
attribute name.
10 The actual data change itself. Depending on the operation and the connector, the data might 
contain before, after and/or patch fields.

The following example also shows what a CloudEvents change event record emitted by a PostgreSQL
connector looks like. In this example, the PostgreSQL connector is again configured to use JSON as the CloudEvents format envelope, but this time the connector is configured to use Avro for the data format.

```json
{
    "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",
    "iodebeziumop": "r",
    "iodebeziumversion": "1.4.2.Final",
    "iodebeziumconnector": "postgresql",
    "iodebeziumname": "test_server",
    "iodebeziumtsms": "1578924258597",
    "iodebeziumsnapshot": "true",
    "iodebeziumdb": "postgres",
    "iodebeziumschema": "s1",
    "iodebeziumtable": "a",
    "iodebeziumtxid": "578",
    "iodebeziumtxln": "33227720",
    "iodebeziumxmin": null,
    "iodebeziumtxidxid": "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.

### 9.8.2. 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](http://my-registry/schemas/ids/1) to serialize the data attribute as binary Avro data.

```json
"value.converter": "io.debezium.converters.CloudEventsConverter",
"value.converter.serializer.type": "json",
```
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`.

### 9.8.3. Debezium CloudEvents converter configuration options

When you configure a Debezium connector to use the CloudEvent converter you can specify the following options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>serializer.type</code></td>
<td>json</td>
<td>The encoding type to use for the CloudEvents envelope structure. The value can be <code>json</code> or <code>avro</code>.</td>
</tr>
<tr>
<td><code>data.serializer.type</code></td>
<td>json</td>
<td>The encoding type to use for the <code>data</code> attribute. The value can be <code>json</code> or <code>avro</code>.</td>
</tr>
<tr>
<td><code>json. ...</code></td>
<td>N/A</td>
<td>Any configuration options to be passed through to the underlying converter when using JSON. The <code>json. </code> prefix is removed.</td>
</tr>
<tr>
<td><code>avro. ...</code></td>
<td>N/A</td>
<td>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.</td>
</tr>
</tbody>
</table>

Revised on 2021-05-13 02:59:10 UTC