Red Hat Integration 2020-Q2

Debezium User Guide

For use with Debezium 1.1

Last Updated: 2020-07-25

Red Hat Integration 2020-Q2 Debezium User Guide

For use with Debezium 1.1

Legal Notice

Copyright © 2020 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

PREFACE

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 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 7, *Debezium logging*
- Chapter 6, *Monitoring Debezium*
- Chapter 8, *Configuring Debezium connectors for your application*

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, “Debezium Architecture”

1.1. DEBEZIUM FEATURES

Debezium is a set of source connectors for Apache Kafka Connect, ingesting changes from different databases using change data capture (CDC). Unlike other approaches such as polling or dual writes, log-based CDC as implemented by Debezium:

- makes sure that **all data changes are captured**
- produces change events with a **very low delay** (e.g. ms range for MySQL or Postgres) while avoiding increased CPU usage of frequent polling
- requires **no changes to your data model**(such as "Last Updated" column)
- can **capture deletes**
- can **capture old record state and further metadata**such as transaction id and causing query (depending on the database’s capabilities and configuration)

To learn more about the advantages of log-based CDC, refer to this blog post .

The actual change data capture feature of Debezium is amended 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 gets started up and not all logs still exist (typically the case when the database has been running for some time and has discarded any transaction logs not needed any longer for transaction recovery or replication); different modes exist for snapshotting, refer to the docs of the specific connector you’re using to learn more
- **Filters:** the set of captured schemas, tables and columns can be configured via whitelist/blacklist filters
- **Masking:** the values from specific columns can be masked, e.g. for sensitive data
- **Monitoring:** most connectors can be monitored using JMX
- Different ready-to-use **message transformations**: e.g. for message routing, extraction of new record state (relational connectors, MongoDB) and routing of events from a transactional outbox table

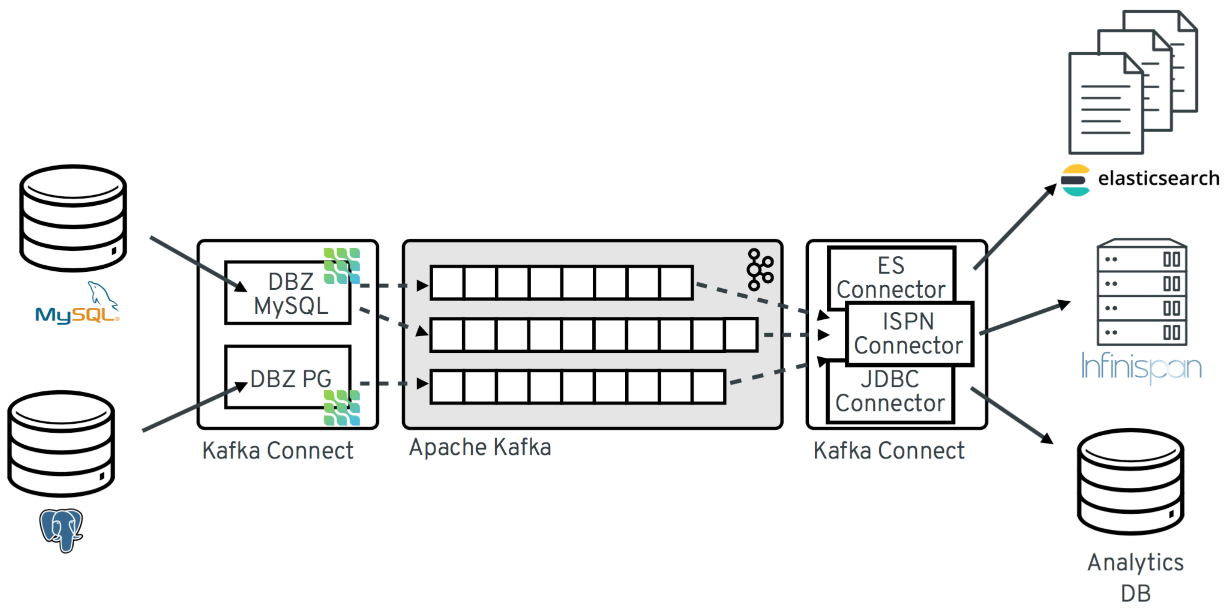
Refer to the connector documentation for a list of all supported databases and detailed information about the features and configuration options of each connector.

1.2. DEBEZIUM ARCHITECTURE

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

- Source connectors such as Debezium that send records into Kafka
- Sink connectors that propagate records from Kafka topics to other systems

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



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

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

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

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

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

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

CHAPTER 2. DEBEZIUM CONNECTOR FOR MYSQL

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

The MySQL connector reads the binlog and produces change events for row-level **INSERT**, **UPDATE**, and **DELETE** operations and records the change events in a Kafka topic. 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 and initial *consistent snapshot* of each of your databases. The MySQL connector reads the binlog from the point at which the snapshot was made.

2.1. OVERVIEW OF HOW THE MYSQL CONNECTOR WORKS

The Debezium MySQL connector tracks the structure of the tables, performs snapshots, transforms binlog events into Debezium change events and records where those events are recorded in Kafka.

2.1.1. How the MySQL connector uses database schemas

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 the row-level changes to the data and 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 and to produce the appropriate change event. In a separate database history Kafka topic, the connector also 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 generate schema change events on a different topic that is intended for consumer applications. This is described in how the MySQL connector handles schema change topics.

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 then helper tables created during the migration process need to be included among whitelisted tables.

If downstream systems do not need the messages generated by the temporary table then a simple message transform can be written and applied to filter them out.

For information about topic naming conventions, see MySQL connector and Kafka topics.

2.1.2. How the MySQL connector performs database snapshots

When your Debezium MySQL connector is first started, it performs an initial *consistent snapshot* of your database. The following flow describes how this snapshot is completed.

NOTE

This is the default snapshot mode which is set as **initial** in the **snapshot.mode** property. For other snapshots modes, please check out the MySQL connector configuration properties.

The connector…

Step	Action
1	Grabs a **global read lock** that blocks *writes* by other database clients. NOTE The snapshot itself does not prevent other clients from applying DDL which might interfere with the connector's attempt to read the binlog position and table schemas. The global read lock is kept while the binlog position is read before released in a later step.
2	Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the *consistent snapshot*.
3	Reads the current binlog position.
4	Reads the schema of the databases and tables allowed by the connector's configuration.
5	Releases the **global read lock**. This now allows other database clients to write to the database.
6	Writes the DDL changes to the schema change topic, including all necessary **DROP…** and **CREATE…** DDL statements. NOTE This happens if applicable.
7	Scans the database tables and generates **CREATE** events on the relevant table-specific Kafka topics for each row.
8	Commits the transaction.
9	Records the completed snapshot in the connector offsets.

2.1.2.1. What happens if the connector fails?

If the connector fails, stops, or is rebalanced while making the *initial snapshot*, the connector creates a new snapshot once restarted. Once that *intial snapshot* is completed, the Debezium MySQL connector restarts from the same position in the binlog so it does not miss any updates.

NOTE

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 MySQL connector common issues.

2.1.2.2. What if Global Read Locks are not allowed?

Some environments do not allow a **global read lock**. 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.

IMPORTANT

The user must have **LOCK_TABLES** privileges.

The connector…

Step	Action
1	Starts a transaction with repeatable read semantics to ensure that all subsequent reads within the transaction are done against the *consistent snapshot*.
2	Reads and filters the names of the databases and tables.
3	Reads the current binlog position.
4	Reads the schema of the databases and tables allowed by the connector's configuration.
5	Writes the DDL changes to the schema change topic, including all necessary **DROP…** and **CREATE…** DDL statements. NOTE This happens if applicable.
6	Scans the database tables and generates **CREATE** events on the relevant table-specific Kafka topics for each row.
7	Commits the transaction.
8	Releases the table-level locks.
9	Records the completed snapshot in the connector offsets.

2.1.3. How the MySQL connector handles schema change topics

You can configure the Debezium **MySQL connector** to produce schema change events that include all DDL statements applied to databases in the MySQL server. The connector writes all of these events to a Kafka topic named **<serverName>** where **serverName** is the name of the connector as specified in the **database.server.name** configuration property.

IMPORTANT

If you choose to use *schema change events*, use the schema change topic and **do not** consume the database history topic.

NOTE

It is vital that there is a global order of the events in the database schema history. Therefore, the database history topic must not be partitioned. This means that a partition count of 1 must be specified when creating this topic. When relying on auto topic creation, make sure that Kafka's **num.partitions** configuration option (the default number of partitions) is set to **1**.

2.1.3.1. Schema change topic structure

Each message that is written to the schema change topic contains a message key which includes the name of the connected database used when applying DDL statements:

```
{
  "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 message value contains a structure that includes the DDL statements, the database to which the statements were applied, and the position in the binlog where the statements appeared:

```
{
  "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_sec"
        },
        {
          "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": "0.10.0.Beta4",
      "name": "mysql-server-1",
      "server_id": 0,
      "ts_sec": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "snapshot": true,
      "thread": null,
      "db": null,
      "table": null,
      "query": null
    }
  }
}
```

2.1.3.1.1. Important tips regarding schema change topics

The **ddl** field may contain multiple DDL statements. Every statement applies to the database in the **databaseName** field and appears in the same order as they were applied in 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 topic.

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

What if a client submits DDL statements 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.

Additional resources

- If you do not use the *schema change topics* detailed here, check out the database history topic.

2.1.4. MySQL connector events

All data change events produced by the Debezium MySQL connector contain a key and a value. The change event key and the change event value each contain a *schema* and a *payload* where the schema describes the structure of the payload and the payload contains the data.

WARNING

The MySQL connector ensures that all Kafka Connect schema names adhere to the Avro schema name format . This is important as any character that is not a latin letter or underscore is replaced by an underscore which can lead to unexpected conflicts in schema names when the logical server names, database names, and table names container other characters that are replaced with these underscores.

2.1.4.1. Change event key

For any given table, the change event's key has a structure that contains a field for each column in the **PRIMARY KEY** (or unique constraint) at the time the event was created. Let us look at an example table and then how the schema and payload would appear for the table.

example 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;
```

example change event key

```
{
 "schema": { 1
   "type": "struct",
 "name": "mysql-server-1.inventory.customers.Key", 2
 "optional": false, 3
 "fields": [ 4
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
 "payload": { 5
   "id": 1001
  }
}
```

1. The **schema** describes what is in the **payload**.

2. The **mysql-server-1.inventory.customers.Key** is the name of the schema which defines the structure where **mysql-server-1** is the connector name, **inventory** is the database, and **customers** is the table.

3. Denotes that the **payload** is not optional.

4. Specifies the type of fields expected in the **payload**.

5. The payload itself, which in this case only contains a single **id** field.

This key describes the row in the **inventory.customers** table which is out from the connector entitled **mysql-server-1** whose **id** primary key column has a value of **1001**.

2.1.4.2. Change event value

The change event value contains a schema and a payload section. There are three types of change event values which have an envelope structure. The fields in this structure are explained below and marked on each of the change event value examples.

- Section 2.1.4.2.1, “Create change event value”
- Section 2.1.4.2.2, “Update change event value”
- Section 2.1.4.2.3, “Delete change event value”

Item	Field name	Description
1	**name**	**mysql-server-1.inventory.customers.Key** is the name of the schema which defines the structure where **mysql-server-1** is the connector name, **inventory** is the database and **customers** is the table
2	**op**	A **mandatory** string that describes the type of operation. **values** • **c** = create • **u** = update • **d** = delete • **r** = read (*initial snapshot* only)
3	**before**	An optional field that specifies the state of the row before the event occurred.
4	**after**	An optional field that specifies the state of the row after the event occurred.
5	**source**	A **mandatory** field that describes the source metadata for the event including: • the Debezium version • the connector name • the binlog name where the event was recorded • the binlog position • the row within the event • if the event was part of a snapshot • the name of the affected database and table • the id of the MySQL thread creating the event (non-snapshot only) • the MySQL server ID (if available) • timestamp NOTE If the binlog_rows_query_log_events option is enabled and the connector has the **include.query** option enabled, a **query** field is displayed which contains the original SQL statement that generated the event.

Item	Field name	Description
6	**ts_ms**	An optional field that displays the time at which the connector processed the event. NOTE The time is based on the system clock in the JVM running the Kafka Connect task.

Let us look at an example table and then how the schema and payload would appear for the table.

example 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;
```

2.1.4.2.1. Create change event value

This example shows a *create* event for the **customers** table:

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
      "optional": true,
      "name": "mysql-server-1.inventory.customers.Value",
      "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.product.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"
    },
    "payload": { 2
      "op": "c",
      "ts_ms": 1465491411815,
      "before": null,
      "after": {
        "id": 1004,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
      },
      "source": {
        "version": "1.1.2.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')"
      }
    }
  }
```

1. The **schema** portion of this event's *value* shows the schema for the envelope, the schema for the source structure (which is specific to the MySQL connector and reused across all events), and the table-specific schemas for the **before** and **after** fields.

2. The **payload** portion of this event's *value* shows the information in the event, namely that it is describing that the row was created (because **op=c**), and that the **after** field value contains the values of the new inserted row's **id**, **first_name**, **last_name**, and **email** columns.

2.1.4.2.2. Update change event value

The value of an *update* change event on the **customers** table has the exact same schema as a *create* event. The payload is structured the same, but holds different values. Here is an example (formatted for readability):

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { 2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 3
      "version": "1.1.2.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u", 4
    "ts_ms": 1465581029523 5
  }
}
```

Comparing this to the value in the *insert* event, you can see a couple of differences in the **payload** section:

1. The **before** field now has the state of the row with the values before the database commit.

2. The **after** field now has the updated state of the row, and the **first_name** value is now **Anne Marie**. You can compare the **before** and **after** structures to determine what actually changed in this row because of the commit.

3. The **source** field structure has the same fields as before, but the values are different (this event is from a different position in the binlog). The **source** structure shows information about MySQL's record of this change (providing traceability). It also has information you can use to compare to other events in this and other topics to know whether this event occurred before, after, or as part of the same MySQL commit as other events.

4 The **op** field value is now **u**, signifying that this row changed because of an update.

5 The **ts_ms** field shows the timestamp when Debezium processed this event.

NOTE

When the columns for a row's primary or unique key are updated, the value of the row's key is changed and Debezium outputs three events: a *DELETE* event and tombstone event with the old key for the row, followed by an *INSERT* event with the new key for the row.

2.1.4.2.3. Delete change event value

The value of a *delete* change event on the **customers** table has the exact same schema as *create* and *update* events. The payload is structured the same, but holds different values. Here is an example (formatted for readability):

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, 2
    "source": { 3
      "version": "1.1.2.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d", 4
    "ts_ms": 1465581902461 5
  }
}
```

Comparing the **payload** portion to the payloads in the *create* and *update* events, you can see some differences:

1 The **before** field now has the state of the row that was deleted with the database commit.

2 The **after** field is **null**, signifying that the row no longer exists.

3 The **source** field structure has many of the same values as before, except the **ts_sec** and **pos** fields have changed (and the file might have changed in other scenarios).

4 The **op** field value is now **d**, signifying that this row was deleted.

5 The **ts_ms** shows the timestamp when Debezium processed this event.

This event provides a consumer with the information that 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.

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

When a row is deleted, the *delete* event value listed above still works with log compaction, because Kafka can still remove all earlier messages with that same key. If the message value is **null**, Kafka knows that it can remove all messages with that same key. To make this possible, Debezium's MySQL connector always follows a *delete* event with a special tombstone event that has the same key but a **null** value.

2.1.5. How the MySQL connector maps 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 the value is represented 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 following table shows how the connector maps the 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)

MySQL type	Literal type	Semantic type
BOOLEAN, BOOL	**BOOLEAN**	*n/a*
BIT(1)	**BOOLEAN**	*n/a*

MySQL type	Literal type	Semantic type
BIT(>1)	**BYTES**	**io.debezium.data.Bits** NOTE The **length** schema parameter contains an integer that represents the number of bits. The **byte[]** contains the bits in *little-endian* form and is sized to contain the specified number of bits. example (where n is bits) `numBytes = n/8 + (n%8== 0 ? 0 : 1)`
TINYINT	**INT16**	*n/a*
SMALLINT[(M)]	**INT16**	*n/a*
MEDIUMINT[(M)]	**INT32**	*n/a*
INT, INTEGER[(M)]	**INT32**	*n/a*
BIGINT[(M)]	**INT64**	*n/a*
REAL[(M,D)]	**FLOAT32**	*n/a*
FLOAT[(M,D)]	**FLOAT64**	*n/a*
DOUBLE[(M,D)]	**FLOAT64**	*n/a*
CHAR(M)]	**STRING**	*n/a*
VARCHAR(M)]	**STRING**	*n/a*
BINARY(M)]	**BYTES**	*n/a*
VARBINARY(M)]	**BYTES**	*n/a*
TINYBLOB	**BYTES**	*n/a*
TINYTEXT	**STRING**	*n/a*
BLOB	**BYTES**	*n/a*
TEXT	**STRING**	*n/a*

MySQL type	Literal type	Semantic type
MEDIUMBLOB	**BYTES**	*n/a*
MEDIUMTEXT	**STRING**	*n/a*
LONGBLOB	**BYTES**	*n/a*
LONGTEXT	**STRING**	*n/a*
JSON	**STRING**	**io.debezium.data.Json** NOTE Contains the string representation of a **JSON** document, array, or scalar.
ENUM	**STRING**	**io.debezium.data.Enum** NOTE The **allowed** schema parameter contains the comma-separated list of allowed values.
SET	**STRING**	**io.debezium.data.EnumSet** NOTE The **allowed** schema parameter contains the comma-separated list of allowed values.
YEAR[(2|4)]	**INT32**	**io.debezium.time.Year**
TIMESTAMP[(M)]	**STRING**	**io.debezium.time.ZonedTimestamp** NOTE In ISO 8601 format with microsecond precision. MySQL allows **M** to be in the range of**0-6**.

2.1.5.1. Temporal values

Excluding the **TIMESTAMP** data type, MySQL temporal types depend on the value of the **time.precision.mode** 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 milli-seconds or micro-seconds based on the column's precision by using UTC. The **TIMESTAMP** type represents a timestamp without time zone information and is converted by MySQL from the server (or session's) current time zone into UTC when writing and vice versa when reading back the value. For example:

- **DATETIME** with a value of **2018-06-20 06:37:03** becomes **1529476623000**.
- **TIMESTAMP** with a value of **2018-06-20 06:37:03** becomes **2018-06-20T13:37:03Z**.

Such columns are converted into an equivalent **io.debezium.time.ZonedTimestamp** in UTC based on the server (or session's) current time zone. The time zone will be queried from the server by default. If this fails, it must be specified explicitly by the **database.serverTimezone** connector configuration property. For example, if the database's time zone (either globally or configured for the connector by means of the **database.serverTimezone property**) 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".

Note that the time zone of the JVM running Kafka Connect and Debezium does not affect these conversions.

More details about properties related to termporal 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.

MySQL type	Literal type	Semantic type
DATE	**INT32**	**io.debezium.time.Date** NOTE Represents the number of days since epoch.
TIME[(M)]	**INT64**	**io.debezium.time.MicroTime** NOTE Represents the time value in microseconds and does not include time zone information. MySQL allows **M** to be in the range of**0-6**.

MySQL type	Literal type	Semantic type
DATETIME, DATETIME(0), DATETIME(1), DATETIME(2), DATETIME(3)	**INT64**	**io.debezium.time.Timestamp** NOTE Represents the number of milliseconds past epoch and does not include time zone information.
DATETIME(4), DATETIME(5), DATETIME(6)	**INT64**	**io.debezium.time.MicroTimestamp** NOTE Represents the number of microseconds past epoch and does not include time zone information.

time.precision.mode=connect

The MySQL connector uses the predefined 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**. Only values in 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.

MySQL type	Literal type	Semantic type
DATE	**INT32**	**org.apache.kafka.connect.data.Date** NOTE Represents the number of days since epoch.
TIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Time** NOTE Represents the time value in microseconds since midnight and does not include time zone information.

MySQL type	Literal type	Semantic type
DATETIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Timestamp** NOTE Represents the number of milliseconds since epoch, and does not include time zone information.

== Decimal values

Decimals are handled via the **decimal.handling.mode** property.

TIP

See MySQL connector configuration properties for more details.

decimal.handling.mode=precise

MySQL type	Literal type	Semantic type
NUMERIC[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** NOTE The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
DECIMAL[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** NOTE The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.

decimal.handling.mode=double

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

decimal.handling.mode=string

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

2.1.5.2. Boolean values

MySQL handles the **BOOLEAN** value internally in a specific way. The **BOOLEAN** column is internally mapped to **TINYINT(1)** datatype. When the table is created during streaming then it uses proper **BOOLEAN** mapping as Debezium receives the original DDL. During snapshot Debezium executes **SHOW CREATE TABLE** to obtain table definition which returns **TINYINT(1)** for both **BOOLEAN** and **TINYINT(1)** columns.

Debezium then has no way how to obtain the original type mapping and will map to **TINYINT(1)**.

An example configuration is

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

2.1.5.3. Spatial data types

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

MySQL type	Literal type	Semantic type

MySQL type	Literal type	Semantic type
GEOMETRY, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	**STRUCT**	**io.debezium.data.geometry.Geometry** NOTE Contains a structure with two fields: ● **srid (INT32**: a spatial reference system id that defines the type of geometry object stored in the structure ● **wkb (BYTES)**: a binary representation of the geometry object encoded in the Well-Known-Binary (wkb) format. See the Open Geospatial Consortium for more details.

2.1.6. The MySQL connector and Kafka topics

The Debezium MySQL connector writes events for all **INSERT**, **UPDATE**, and **DELETE** operations from a single table to a single Kafka topic. The Kafka topic naming convention is as follows:

format

```
serverName.databaseName.tableName
```

Example 2.1. example

Let us say that **fulfillment** is the server name and **inventory** is the database which contains **three** tables of **orders**, **customers**, and **products**. The Debezium MySQL connector produces events on three Kafka topics, one for each table in the database:

```
fulfillment.inventory.orders

fulfillment.inventory.customers

fulfillment.inventory.products
```

2.1.7. MySQL supported topologies

The Debezium MySQL connector supports the following MySQL topologies:

Standalone

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

Master and slave

The Debezium MySQL connector can follow one of the masters or one of the slaves (*if that slave has its binlog enabled*), but the connector only sees changes in the cluster that are visible to that server. Generally, this is not a problem except for the multi-master topologies.

The connector records its position in the server's binlog, which is different on each server in the cluster. Therefore, the connector will need to follow just one MySQL server instance. If that server fails, it 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 far easier to tolerate and almost immediately recover from problems and failures. Most HA MySQL clusters use GTIDs so that slaves are able to keep track of all changes on any of the master.

Multi-master

A multi-master MySQL topology uses one or more MySQL slaves that each replicate from multiple masters. This is a powerful way to aggregate the replication of multiple MySQL clusters, and requires using GTIDs.
The Debezium MySQL connector can use these multi-master MySQL slaves as sources, and can fail over to different multi-master MySQL slaves as long as thew new slave is caught up to the old slave (*e.g., the new slave has all of the transactions that were last seen on the first slave*). This works even if the connector is only using 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-master MySQL slave 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.

IMPORTANT

Because these hosted options do not allow a **global read lock**, table-level locks are used to create the *consistent snapshot*.

2.2. SETTING UP MYSQL SERVER

2.2.1. Creating a MySQL user for Debezium

You have to define a MySQL user with appropriate permissions on all databases that the Debezium MySQL connector monitors.

Prerequisites

- You must have a MySQL server.
- You must know basic 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';
```

TIP

See permissions explained for notes on each permission.

IMPORTANT

If using a hosted option such as Amazon RDS or Amazon Aurora that do 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 Overview of how the MySQL connector works for more details.

3. Finalize the user's permissions:

```
mysql> FLUSH PRIVILEGES;
```

2.2.1.1. Permissions explained

Permission/item	Description
SELECT	enables the connector to select rows from tables in databases NOTE This is only used when performing a snapshot.
RELOAD	enables the connector the use of the **FLUSH** statement to clear or reload internal caches, flush tables, or acquire locks. NOTE This is only used when performing a snapshot.
SHOW DATABASES	enables the connector to see database names by issuing the **SHOW DATABASE** statement. NOTE This is only used when performing a snapshot.
REPLICATION SLAVE	enables the connector to connect to and read the MySQL server binlog.

Permission/item	Description
REPLICATION CLIENT	enables the connector the use of following statements: • **SHOW MASTER STATUS** • **SHOW SLAVE STATUS** • **SHOW BINARY LOGS** IMPORTANT This is always required for the connector.
ON	Identifies the **database** to which the permission apply.
TO 'user'	Specifies the **user** to which the permissions are granted.
IDENTIFIED BY 'password'	Specifies the **password** for the user.

2.2.2. 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

- You must have a MySQL server.
- You should have appropriate MySQL user privileges.

Procedure

1. Check if the **log-bin** option is already on or not.

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

2. If **OFF**, configure your MySQL server configuration file with the following:

TIP

See Binlog config properties for notes on each property.

```
server-id         = 223344 (1)
log_bin           = mysql-bin (2)
binlog_format     = ROW (3)
```

```
binlog_row_image  = FULL 4
expire_logs_days  = 10 5
```

3. Confirm your changes by checking the binlog status once more.

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

2.2.2.1. Binlog configuration properties

Number	Property	Description
1	**server-id**	The value for the **server-id** must be unique for each server and replication client within the MySQL cluster. When the MySQL connector is setup, we assign the connector a unique server ID.
2	**log_bin**	The value of **log_bin** is the base name of the sequence of binlog files.
3	**binlog_format**	The **binlog-format** must be set to **ROW** or **row**.
4	**binlog_row_image**	The **binlog_row_image** must be set to **FULL** or **full**.
5	**expire_logs_days**	This is the number of days for automatic binlog file removal. The default is **0** which means no automatic removal. **TIP** Set the value to match the needs of your environment.

2.2.3. 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 the Debezium MySQL connector, using GTIDs simplifies replication and allows you to more easily confirm if master and slave servers are consistent.

NOTE

GTIDs are only available from MySQL 5.6.5 and later. See the MySQL documentation for more details.

Prerequisites

- You must have a MySQL server.
- You must know basic SQL commands.
- You must have 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%';
```

response

```
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| enforce_gtid_consistency | ON    |
| gtid_mode                | ON    |
+--------------------------+-------+
```

2.2.3.1. Options explained

Permission/item	Description
gtid_mode	Boolean which specifies whether GTID mode of the MySQL server is enabled or not. • **ON** = enabled • **OFF** = disabled
enforce_gtid_consistency	Boolean which instructs the server whether or not to enforce GTID consistency by allowing the execution of statements that can be logged in a transactionally safe manner; required when using GTIDs. • **ON** = enabled • **OFF** = disabled

2.2.4. Setting up 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

- You must have a MySQL server.

- You must know basic SQL commands.
- You must have 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>
```

2.2.4.1. Options explained

Permission/item	Description
interactive_timeout	The number of seconds the server waits for activity on an interactive connection before closing it. TIP See MySQL's documentation for more details.
wait_timeout	The number of seconds the server waits for activity on a noninteractive connection before closing it. TIP See MySQL's documentation for more details.

2.2.5. Enabling query log events for Debezium

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.

NOTE

This option is only available from MySQL 5.6 and later.

Prerequisites

- You must have a MySQL server.
- You must know basic SQL commands.
- You must have access to the MySQL configuration file.

Procedure

1. Enable **binlog_rows_query_log_events**:

```
mysql> binlog_rows_query_log_events=ON
```

2.2.5.1. Options explained

Permission/item	Description
binlog_rows_query_log_events`	Boolean which enables/disables support for including the original **SQL** statement in the binlog entry. ● **ON** = enabled ● **OFF** = disabled

2.3. DEPLOYING THE MYSQL CONNECTOR

2.3.1. Installing the MySQL connector

Installing the Debezium MySQL connector is a simple process whereby you only need to download the JAR, extract it to your Kafka Connect environment, and ensure the plug-in's parent directory is specified in your Kafka Connect environment.

Prerequisites

- You have Zookeeper, Kafka, and Kafka Connect installed.
- You have MySQL Server installed and setup.

Procedure

1. Download the Debezium MySQL connector.
2. Extract the files into your Kafka Connect environment.
3. Add the plug-in's parent directory to your Kafka Connect **plugin.path**:

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

> **NOTE**
>
> The above example assumes you have extracted the Debezium MySQL connector to the **/kafka/connect/debezium-connector-mysql** path.

4. Restart your Kafka Connect process. This ensures the new JARs are picked up.

2.3.2. Configuring the MySQL connector

Typically, you configure the Debezium MySQL connector in a **.yaml** file using the configuration properties available for the connector.

Prerequisites

- You should have completed the installation process for the connector.

Procedure

1. Set the **"name"** of the connector in the **.yaml** file.
2. Set the configuration properties that you require for your Debezium MySQL connector.

TIP

For a complete list of configuration properties, see MySQL connector configuration properties.

MySQL connector example configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector  1
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 1  2
  config:  3
    database.hostname: mysql  4
    database.port: 3306
    database.user: debezium
    database.password: dbz
    database.server.id: 184054  5
    database.server.name: dbserver1  6
    database.whitelist: inventory  7
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092  8
    database.history.kafka.topic: schema-changes.inventory  9
```

1 1 The name of the connector.

2 2 Only one task should operate at any one time. Because the MySQL connector reads the MySQL server's **binlog**, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.

3 3 The connector's configuration.

4 4 The database host, which is the name of the container running the MySQL server (**mysql**).

5 5 6 A unique server ID and name. The server name is the logical identifier for the MySQL server or cluster of servers. This name will be used as the prefix for all Kafka topics.

7 Only changes in the **inventory** database will be detected.

8 9 The connector will store the history of the database schemas in Kafka using this broker (the same broker to which you are sending events) and topic name. Upon restart, the connector will recover the schemas of the database that existed at the point in time in the **binlog** when the connector should begin reading.

2.3.3. MySQL connector configuration properties

The configuration properties listed here are **required** to run the Debezium MySQL connector. There are also advanced MySQL connector properties whose default value rarely needs to be changed and therefore, they do not need to be specified in the connector configuration.

TIP

The Debezium MySQL connector supports *pass-through* configuration when creating the Kafka producer and consumer. See information about pass-through properties at the end of this section, and also see the Kafka documentation for more details about *pass-through* properties.

Property	Default	Description
name		Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium .connector.mysql.MySqlConnector** for the MySQL connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The MySQL connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname		IP address or hostname of the MySQL database server.
database.port	**3306**	Integer port number of the MySQL database server.
database.user		Name of the MySQL database to use when connecting to the MySQL database server.
database.password		Password to use when connecting to the MySQL database server.

Property	Default	Description
database.server.name		Logical name that identifies and provides a namespace for the particular MySQL database server/cluster being monitored. The logical name should be unique across all other connectors, since it is used as a prefix for all Kafka topic names emanating from this connector. Only alphanumeric characters and underscores should be used.
database.server.id	*random*	A numeric ID of this database client, which must be unique across all currently-running database processes in the MySQL cluster. This connector joins the MySQL database cluster as another server (with this unique ID) so it can read the binlog. By default, a random number is generated between 5400 and 6400, though we recommend setting an explicit value.
database.history.kafka.topic		The full name of the Kafka topic where the connector will store the database schema history.
database.history .kafka.bootstrap.servers		A list of host/port pairs that the connector will use for establishing an initial connection to the Kafka cluster. This connection will be used for retrieving database schema history previously stored by the connector, and for writing each DDL statement read from the source database. This should point to the same Kafka cluster used by the Kafka Connect process.
database.whitelist	*empty string*	An optional comma-separated list of regular expressions that match database names to be monitored; any database name not included in the whitelist will be excluded from monitoring. By default all databases will be monitored. May not be used with **database.blacklist**.
database.blacklist	*empty string*	An optional comma-separated list of regular expressions that match database names to be excluded from monitoring; any database name not included in the blacklist will be monitored. May not be used with **database.whitelist**.

Property	Default	Description
table.whitelist	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be monitored; any table not included in the whitelist will be excluded from monitoring. Each identifier is of the form *databaseName*.*tableName*. By default the connector will monitor every non-system table in each monitored database. May not be used with **table.blacklist**.
table.blacklist	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be excluded from monitoring; any table not included in the blacklist will be monitored. Each identifier is of the form *databaseName*.*tableName*. May not be used with **table.whitelist**.
column.blacklist	*empty string*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that should be excluded from change event message values. Fully-qualified names for columns are of the form *databaseName*.*tableName*.*columnName*, or *databaseName*.*schemaName*.*tableName*.*columnName*.
column.truncate.to.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event message values if the field values are longer than the specified number of characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer. Fully-qualified names for columns are of the form *databaseName*.*tableName*.*columnName*.
column.mask.with.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*****) characters. 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 *databaseName*.*tableName*.*columnName*.

Property	Default	Description
column.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__Debezium.source.column.type**, **__Debezium.source.column.length** and **_Debezium.source.column.scale** 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 names for columns are of the form *databaseName.tableName.columnName*, or *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 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 *databaseName.tableName.typeName*, or *databaseName.schemaName.tableName.typeName*. See how the MySQL connector maps data types for the list of MySQL-specific data type names.
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; 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.

Property	Default	Description
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; or **double** represents them using **double** values, which may result in a loss of precision but will be far easier to use. **string** option encodes values as formatted string which is easy to consume but a semantic information about the real type is lost.
bigint.unsigned.handling.mode	**long**	Specifies how BIGINT UNSIGNED columns should be represented in change events, including: **precise** uses **java.math.BigDecimal** to represent values, which are encoded in the change events using a binary representation and Kafka Connect's **org.apache.kafka.connect.data.Decimal** type; **long** (the default) represents values using Java's **long**, which may not offer the precision but will be far easier to use in consumers. **long** is usually the preferable setting. Only when working with values larger than 2^63, the **precise** setting should be used as those values cannot be conveyed using **long**.
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change will be recorded 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. The default is **true**.
include.query	**false**	Boolean value that specifies whether the connector should include the original SQL query that generated the change event. Note: This option requires MySQL be configured with the binlog_rows_query_log_events option set to ON. Query will not be present for events generated from the snapshot process. WARNING: Enabling this option may expose tables or fields explicitly blacklisted or masked by including the original SQL statement in the change event. For this reason this option is defaulted to 'false'.

Property	Default	Description
event.processing .failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during deserialization of binlog events. **fail** will propagate the exception (indicating the problematic event and its binlog offset), causing the connector to stop. **warn** will cause the problematic event to be skipped and the problematic event and its binlog offset to be logged. **skip** will cause problematic event will be skipped.
inconsistent.schema.handling.mode	**fail**	Specifies how the connector should react to binlog events that relate to tables that are not present in internal schema representation (i.e. internal representation is not consistent with database) **fail** will throw an exception (indicating the problematic event and its binlog offset), causing the connector to stop. **warn** will cause the problematic event to be skipped and the problematic event and its binlog offset to be logged. **skip** will cause the problematic event to be skipped.
max.queue.size	**8192**	Positive integer value that specifies the maximum size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the binlog reader when, for example, writes to Kafka are slower or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified in the **max.batch.size** property.
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.
connect.timeout.ms	**30000**	A positive integer value that specifies the maximum time in milliseconds this connector should wait after trying to connect to the MySQL database server before timing out. Defaults to 30 seconds.

Property	Default	Description
gtid.source.includes		A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources matching one of these include patterns will be used. May not be used with **gtid.source.excludes**.
gtid.source.excludes		A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources matching none of these exclude patterns will be used. May not be used with **gtid.source.includes**.
tombstones.on.delete	**true**	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.
message.key.columns	*empty string*	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 table>:<a comma-separated list of columns>** representing the custom key. Fully-qualified tables could be defined as *databaseName.tableName*.

2.3.3.1. Advanced MySQL connector properties

The following table describes advanced MySQL connector properties.

Property	Default	Description
connect.keep.alive	**true**	A boolean value that specifies whether a separate thread should be used to ensure the connection to the MySQL server/cluster is kept alive.
table.ignore.builtin	**true**	Boolean value that specifies whether built-in system tables should be ignored. This applies regardless of the table whitelist or blacklists. By default system tables are excluded from monitoring, and no events are generated when changes are made to any of the system tables.

Property	Default	Description
database.history.kafka.recovery.poll.interval.ms	**100**	An integer value that specifies the maximum number of milliseconds the connector should wait during startup/recovery while polling for persisted data. The default is 100ms.
database.history.kafka.recovery.attempts	**4**	The maximum number of times that the connector should attempt to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is **recovery.attempts** x **recovery.poll.interval.ms**.
database.history.skip.unparseable.ddl	**false**	Boolean value that specifies if connector should ignore malformed or unknown database statements or stop processing and let operator to 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 binlog is processed.
database.history.store.only.monitored.tables.ddl	**false**	Boolean value that specifies if connector should should record all DDL statements or (when **true**) only those that are relevant to tables that are monitored by Debezium (via filter configuration). The safe default is **false**. This feature should be used only with care as the missing data might be necessary when the filters are changed.
database.ssl.mode	**disabled**	Specifies whether to use an encrypted connection. The default is **disabled**, and specifies to use an unencrypted connection. The **preferred** option establishes an encrypted connection if the server supports secure connections but falls back to an unencrypted connection otherwise. The **required** option establishes an encrypted connection but will fail if one cannot be made for any reason. The **verify_ca** option behaves like **required** but additionally it verifies the server TLS certificate against the configured Certificate Authority (CA) certificates and will fail if it doesn't match any valid CA certificates. The **verify_identity** option behaves like **verify_ca** but additionally verifies that the server certificate matches the host of the remote connection.

Property	Default	Description
binlog.buffer.size	0	The size of a look-ahead buffer used by the binlog reader. Under specific conditions it is possible that 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 decide whether the changes from the transaction will be streamed or not. The size of the buffer defines the maximum number of changes in the transaction that Debezium can buffer while searching for transaction boundaries. If the size of transaction is larger than the buffer then Debezium needs to rewind and re-read the events that has not fit into the buffer while streaming. Value **0** disables buffering. Disabled by default. *Note:* This feature should be considered an incubating one. We need a feedback from customers but it is expected that it is not completely polished.
snapshot.mode	**initial**	Specifies the criteria for running a snapshot upon startup of the connector. The default is **initial**, and specifies the connector can run a snapshot only when no offsets have been recorded for the logical server name. The **when_needed** option specifies that the connector run a snapshot upon startup whenever it deems it necessary (when no offsets are available, or when a previously recorded offset specifies a binlog location or GTID that is not available in the server). The **never** option specifies that the connect should never use snapshots and that upon first startup with a logical server name the connector should read from the beginning of the binlog; this should be used with care, as it is only valid when the binlog is guaranteed to contain the entire history of the database. If you don’t need the topics to contain a consistent snapshot of the data but only need them to have the changes since the connector was started, you can use the **schema_only** option, where the connector only snapshots the schemas (not the data). **schema_only_recovery** is a recovery option for an existing connector to recover a corrupted or lost database history topic, or to periodically "clean up" a database history topic (which requires infinite retention) that may be growing unexpectedly.

Property	Default	Description
snapshot.locking.mode	**minimal**	Controls if and how long the connector holds onto the global MySQL read lock (preventing any updates to the database) while it is performing a snapshot. There are three possible values **minimal**, **extended**, and **none**. **minimal** The connector holds the global read lock for just the initial portion of the snapshot while the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table, and this can be done in a consistent fashion using the REPEATABLE READ transaction even when the global read lock is no longer held and while other MySQL clients are updating the database. **extended** In some cases where clients are submitting operations that MySQL excludes from REPEATABLE READ semantics, it may be desirable to block all writes for the entire duration of the snapshot. For these such cases, use this option. **none** Will prevent the connector from acquiring any table locks during the snapshot process. This value can be used with all snapshot modes but it is safe to use if and *only* if no schema changes are happening while the snapshot is taken. Note that 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.select.statement.overrides		Controls which rows from tables will be included in snapshot. This property contains a comma-separated list of fully-qualified tables *(DB_NAME.TABLE_NAME)*. Select statements for the individual tables are specified in further configuration properties, one for each table, identified by the id **snapshot.select.statement.overrides.[DB_NAME].[TABLE_NAME]**. The value of those properties is the SELECT statement to use when retrieving data from the specific table during snapshotting. *A possible use case for large append-only tables is setting a specific point where to start (resume) snapshotting, in case a previous snapshotting was interrupted.* **Note**: This setting has impact on snapshots only. Events captured from binlog are not affected by it at all.

Property	Default	Description
min.row.count.to.stream.results	**1000**	During a snapshot operation, the connector will query each included table to produce a read event for all rows in that table. This parameter determines whether the MySQL connection will pull all results for a table into memory (which is fast but requires large amounts of memory), or whether the results will instead be streamed (can be slower, but will work for very large tables). The value specifies the minimum number of rows a table must contain before the connector will stream results, and defaults to 1,000. Set this parameter to '0' to skip all table size checks and always stream all results during a snapshot.
heartbeat.interval.ms	**0**	Controls how frequently the heartbeat messages are sent. This property contains an interval in milli-seconds that defines how frequently the connector sends heartbeat messages into a heartbeat topic. Set this parameter to **0** to not send heartbeat messages at all. Disabled by default.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.
database.initial.statements		A semicolon separated list of SQL statements to be executed when a JDBC connection (not the transaction log reading connection) to the database is established. Use doubled semicolon (';;') to use a semicolon as a character and not as a delimiter. *Note: The connector may establish JDBC connections at its own discretion, so this should typically be used for configuration of session parameters only, but not for executing DML statements.*
snapshot.delay.ms		An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size		Specifies the maximum number of rows that should be read in one go from each table while taking a snapshot. The connector will read the table contents in multiple batches of this size.

Property	Default	Description
snapshot.lock.timeout.ms	**10000**	Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If table locks cannot be acquired in this time interval, the snapshot will fail. See How the MySQL connector performs database snapshots.
enable.time.adjuster		MySQL allows user to insert year value as either 2-digit or 4-digit. In case of two digits the value is automatically mapped to 1970 - 2069 range. This is usually done by database. Set to **true** (the default) when Debezium should do the conversion. Set to **false** when conversion is fully delegated to the database.
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names will be sanitized to adhere to Avro naming requirements.

2.3.3.2. 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
```

2.3.3.3. 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.

2.3.4. MySQL connector monitoring metrics

The Debezium MySQL connector has three 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
- binlog metrics; for monitoring the connector when reading CDC table data
- schema history metrics; for monitoring the status of the connector's schema history

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

2.3.4.1. Snapshot metrics

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

Attribute	Type	Description
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
HoldingGlobalLock	**boolean**	Whether the connector currently holds a global or table write lock.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.

Attribute	Type	Description
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between snapshot reader and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between snapshot reader and the main Kafka Connect loop.

2.3.4.2. Binlog metrics

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

NOTE

The transaction-related attributes are only available if binlog event buffering is enabled. See binlog.buffer.size in the advanced connector configuration properties for more details.

Attribute	Type	Description
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the MySQL server.
BinlogFilename	**string**	The name of the binlog filename that the connector has most recently read.

Attribute	Type	Description
BinlogPosition	**long**	The most recent position (in bytes) within the binlog that the connector has read.
IsGtidModeEnabled	**boolean**	Flag that denotes whether the connector is currently tracking GTIDs from MySQL server.
GtidSet	**string**	The string representation of the most recent GTID set seen by the connector when reading the binlog.
LastEvent	**string**	The last binlog event that the connector has read.
SecondsSinceLastEvent (obsolete)	**long**	The number of seconds since the connector has read and processed the most recent event.
SecondsBehindMaster (obsolete)	**long**	The number of seconds between the last event's MySQL timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the MySQL server and the MySQL connector are running.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last event's MySQL timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the MySQL server and the MySQL connector are running.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfSkippedEvents	**long**	The number of events that have been skipped by the MySQL connector. Typically events are skipped due to a malformed or unparseable event from MySQL's binlog.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
NumberOfDisconnects	**long**	The number of disconnects by the MySQL connector.
SourceEventPosition	**map<string, string>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.
LastEvent	**string**	The last binlog event that the connector has read.

Attribute	Type	Description
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
MonitoredTables	**string[]**	The list of tables that are monitored by Debezium.
QueueTotalCapcity	**int**	The length of the queue used to pass events between binlog reader and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between binlog reader and the main Kafka Connect loop.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
NumberOfRolledBackTransactions	**long**	The number of processed transactions that were rolled back and not streamed.
NumberOfNotWellFormedTransactions	**long**	The number of transactions that have not conformed to expected protocol **BEGIN** + **COMMIT**/**ROLLBACK**. Should be**0** under normal conditions.
NumberOfLargeTransactions	**long**	The number of transactions that have not fitted into the look-ahead buffer. Should be significantly smaller than **NumberOfCommittedTransactions** and **NumberOfRolledBackTransactions** for optimal performance.

2.3.4.3. Schema history metrics

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

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

Attribute	Type	Description
ChangesApplied	**long**	The total number of schema changes applie during recovery and runtime.
MilliSecondsSinceLastRecoveredChange	**long**	The number of milliseconds that elapsed since the last change was recovered from the history store.
MilliSecondsSinceLastAppliedChange	**long**	The number of milliseconds that elapsed since the last change was applied.
LastRecoveredChange	**string**	The string representation of the last change recovered from the history store.
LastAppliedChange	**string**	The string representation of the last applied change.

2.4. MYSQL CONNECTOR COMMON ISSUES

2.4.1. Configuration and startup errors

The Debezium MySQL connector fails, reports an error, and stops running when the following startup errors occur:

- The connector’s configuration is invalid.
- The connector cannot connect to the MySQL server using the specified connectivity parameters.
- The connector is attempting to restart at a position in the binlog where MySQL no longer has the history available.

If you receive any of these errors, you receive more details in the error message. The error message also contains workarounds where possible.

2.4.2. MySQL is unavailable

If your MySQL server becomes unavailable, the Debezium MySQL connector fails with an error and the connector stops. You simply need to restart the connector when the server is available.

2.4.2.1. Using GTIDs

If you have GTIDs enabled and a highly available MySQL cluster, restart the connector immediately as the connector will simply 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.

2.4.2.2. Not Using GTIDs

If you do not have GTIDs enabled, the connector only records the binlog position of the MySQL server to which it was connected. In order to restart from the correct binlog position, you must reconnect to that specific server.

2.4.3. Kafka Connect stops

There are three scenarios that cause some issues when Kafka Connect stops:

- Section 2.4.3.1, "Kafka Connect stops gracefully"
- Section 2.4.3.2, "Kafka Connect process crashes"
- Section 2.4.3.3, "Kafka becomes unavailable"

2.4.3.1. Kafka Connect stops gracefully

When Kafka Connect stops gracefully, there is only a short delay while the Debezium MySQL connector tasks are stopped and restarted on new Kafka Connect processes.

2.4.3.2. 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 may generate some of the same events processed prior to the crash, creating duplicate events.

TIP

Each change event message includes source-specific information about:

- the event origin
- the MySQL server's event time
- the binlog filename and position
- GTIDs (if used)

2.4.3.3. Kafka becomes unavailable

The Kafka Connect framework records Debezium change events in Kafka 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 last left off.

2.4.4. 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.

TIP

See How the MySQL connector performs database snapshots for more information on initial snapshots.

CHAPTER 3. DEBEZIUM CONNECTOR FOR POSTGRESQL

Debezium's PostgreSQL Connector can monitor and record row-level changes in the schemas of a PostgreSQL database.

The first time it connects to a PostgreSQL server/cluster, it reads a consistent snapshot of all of the schemas. When that snapshot is complete, the connector continuously streams the changes that were committed to PostgreSQL 9.6 or later and generates corresponding insert, update and delete events. All of the events for each table are recorded in a separate Kafka topic, where they can be easily consumed by applications and services.

3.1. OVERVIEW

PostgreSQL's *logical decoding* feature was first introduced in version 9.4 and is a mechanism which allows the extraction of the changes which were committed to the transaction log and the processing of these changes in a user-friendly manner via the help of an *output plug-in*. This output plug-in must be installed prior to running the PostgreSQL server and enabled together with a replication slot in order for clients to be able to consume the changes.

PostgreSQL connector contains two different parts which work together in order to be able to read and process server changes:

- A logical decoding output plug-in, which has to be installed and configured in the PostgreSQL server.
- Java code (the actual Kafka Connect connector) which reads the changes produced by the plug-in, using PostgreSQL's *streaming replication protocol*, via the PostgreSQL *JDBC driver*

The connector then produces a *change event* for every row-level insert, update, and delete operation that was received, recording all the change events for each table in a separate Kafka topic. Your client applications read the Kafka topics that correspond to the database tables they're interested in following, and react to every row-level event it sees in those topics.

PostgreSQL normally purges 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, we start with a consistent view of all of the data, yet continue reading without having lost any of the changes made while the snapshot was taking place.

The connector is also tolerant of failures. As the connector reads changes and produces events, it records the position in the write-ahead log with each event. If the connector stops for any reason (including communication failures, network problems, or crashes), upon restart it simply continues reading the WAL where it last left off. This includes snapshots: if the snapshot was not completed when the connector is stopped, upon restart it will begin a new snapshot.

3.1.1. Logical decoding output plug-in

The **pgoutput** logical decoder is the only supported logical decoder in the Tecnhology Preview release of Debezium.

pgoutput, the standard logical decoding plug-in in PostgreSQL 10+, is maintained by the Postgres community, and is also used by Postgres for logical replication. The **pgoutput** plug-in is always present, meaning that no additional libraries must be installed, and the connector will interpret the raw replication event stream into change events directly.

IMPORTANT

The connector's functionality relies on PostgreSQL's logical decoding feature. Please be aware of the following limitations which are also reflected by the connector:

1. Logical Decoding does not support DDL changes: this means that the connector is unable to report DDL change events back to consumers.

2. Logical Decoding replication slots are only supported on **primary** servers: this means that when there is a cluster of PostgreSQL servers, the connector can only run on the active **primary** server. It cannot run on **hot** or **warm** standby replicas. If the **primary** server fails or is demoted, the connector will stop. Once the **primary** has recovered the connector can simply be restarted. If a different PostgreSQL server has been promoted to **primary**, the connector configuration must be adjusted before the connector is restarted. Make sure you read more about how the connector behaves when things go wrong.

IMPORTANT

Debezium currently supports only databases with UTF-8 character encoding. With a single byte character encoding it is not possible to correctly process strings containing extended ASCII code characters.

3.2. SETTING UP POSTGRESQL

This release of Debezium only supports the native pgoutput logical replication stream. To set up PostgreSQL using pgoutput, you will need to enable a replication slot, and configure a user with sufficient privileges to perform the replication.

3.2.1. Configuring the replication slot

PostgreSQL's logical decoding uses replication slots.

First, you configure the replication slot:

postgresql.conf

```
wal_level=logical
max_wal_senders=1
max_replication_slots=1
```

- **wal_level** tells the server to use logical decoding with the write-ahead log
- **max_wal_senders** tells the server to use a maximum of 1 separate processes for processing WAL changes
- **max_replication_slots** tells the server to allow a maximum of 1 replication slots to be created for streaming WAL changes

Replication slots are guaranteed to retain all WAL required for Debezium even during Debezium outages. It is important for this reason to closely monitor replication slots to avoid too much disk consumption and other conditions that can happen such as catalog bloat if a replication slot stays unused for too long. For more information, refer to the the Postgres documentation.

NOTE

We recommend reading and understanding the WAL configuration documentation regarding the mechanics and configuration of the PostgreSQL write-ahead log.

3.2.2. Setting up Permissions

Next, configure a database user who can perform replications.

Replication can only be performed by a database user that has appropriate permissions and only for a configured number of hosts.

In order to give a user replication permissions, define a PostgreSQL role that has *at least* the **REPLICATION** and **LOGIN** permissions. For example:

```
CREATE ROLE name REPLICATION LOGIN;
```

NOTE

Superusers have by default both of the above roles.

Finally, configure the PostgreSQL server to allow replication to take place between the server machine and the host on which the PostgreSQL connector is running:

pg_hba.conf

```
local   replication     <youruser>                    trust (1)
host    replication     <youruser>  127.0.0.1/32      trust (2)
host    replication     <youruser>  ::1/128           trust (3)
```

1. Tells the server to allow replication for **<youruser>** locally (i.e. on the server machine)
2. Tells the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV4**
3. Tells the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV6**

NOTE

See *the PostgreSQL documentation* for more information on network masks.

3.2.3. WAL Disk Space Consumption

In certain cases, it is possible that PostgreSQL disk space consumed by WAL files either experiences spikes or increases out of usual proportions. There are three potential reasons that explain the situation:

- Debezium regularly confirms LSN of processed events to the database. This is visible as **confirmed_flush_lsn** in the **pg_replication_slots** slots table. The database is responsible for reclaiming the disk space and the WAL size can be calculated from **restart_lsn** of the same table. So if the **confirmed_flush_lsn** is regularly increasing and **restart_lsn** lags then the database does need to reclaim the space. Disk space is usually reclaimed in batch blocks so this is expected behavior and no action on a user's side is necessary.

- There are many updates in a monitored database but only a minuscule amount relates to the monitored table(s) and/or schema(s). This situation can be easily solved by enabling periodic heartbeat events using **heartbeat.interval.ms** configuration option.

- The PostgreSQL instance contains multiple databases where one of them is a high-traffic database. Debezium monitors another database that is low-traffic in comparison to the other one. Debezium then cannot confirm the LSN as replication slots work per-database and Debezium is not invoked. As WAL is shared by all databases it tends to grow until an event is emitted by the database monitored by Debezium.

To overcome the third cause it is necessary to

- enable periodic heartbeat record generation using the **heartbeat.interval.ms** configuration option

- regularly emit change events from the database tracked by Debezium

A separate process would then periodically update the table (either inserting a new event or updating the same row all over). PostgreSQL then will invoke Debezium which will confirm the latest LSN and allow the database to reclaim the WAL space. This task can be automated by means of the **heartbeat.action.query** connector option (see below).

TIP

For users on AWS RDS with Postgres, a similar situation to the third cause may occur on an idle environment, since AWS RDS makes writes to its own system tables not visible to the useres on a frequent basis (5 minutes). Again regularly emitting events will solve the problem.

3.2.4. How the PostgreSQL connector works

3.2.4.1. Snapshots

Most PostgreSQL servers are configured to not retain the complete history of the database in the WAL segments, so the PostgreSQL connector would be unable to see the entire history of the database by simply reading the WAL. So, by default the connector will upon first startup perform an initial *consistent snapshot* of the database. Each snapshot consists of the following steps (when using the builtin snapshot modes, **custom** snapshot modes may override this):

1. Start a transaction with a SERIALIZABLE, READ ONLY, DEFERRABLE isolation level to ensure that all subsequent reads within this transaction are done against a single consistent version of the data. Any changes to the data due to subsequent **INSERT**, **UPDATE**, and **DELETE** operations by other clients will not be visible to this transaction.

2. Obtain a **ACCESS SHARE MODE** lock on each of the monitored tables to ensure that no structural changes can occur to any of the tables while the snapshot is taking place. Note that these locks do not prevent table **INSERTS**, **UPDATES** and **DELETES** from taking place during the operation. *This step is omitted when using the exported snapshot mode to allow for a lock-free snapshots.*

3. Read the current position in the server's transaction log.

4. Scan all of the database tables and schemas, and 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 will begin a new snapshot. Once the connector does complete its initial snapshot, the PostgreSQL connector then continues streaming from the position read during step 3, ensuring that it does not miss any updates. If the connector stops again for any reason, upon restart it will simply continue streaming changes from where it previously left off.

A second snapshot mode allows the connector to perform snapshots **always**. This behavior tells the connector to *always* perform a snapshot when it starts up, and after the snapshot completes to continue streaming changes from step 3 in the above sequence. This mode can be used in cases when it is known that some WAL segments have been deleted and are no longer available, or in case of a cluster failure after a new primary has been promoted so that the connector does not miss any potential changes that could have taken place after the new primary had been promoted but before the connector was restarted on the new primary.

The third snapshot mode instructs the connector to **never** performs snapshots. When a new connector is configured this way, if will either continue streaming changes from a previous stored offset or it will start from the point in time when the PostgreSQL logical replication slot was first created on the server. Note that this mode is useful only when you know all data of interest is still reflected in the WAL.

The fourth snapshot mode, **initial only**, will perform a database snapshot and then stop before streaming any other changes. If the connector had started but did not complete a snapshot before stopping, the connector will restart the snapshot process and stop once the snapshot completes.

The fifth snapshot mode, **exported**, will perform 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.

3.2.4.2. Streaming Changes

The PostgreSQL connector will typically spend 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* where the client can receive changes from the server as they are committed in the server's transaction log at certain positions (also known as **Log Sequence Numbers** or in short 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 PostgreSQL connector acts as a PostgreSQL client, and when it receives these changes it transforms the events into Debezium *create*, *update*, or *delete* events that include the LSN position of the event. The PostgreSQL connector forwards these change events to the Kafka Connect framework (running in the same process), which then asynchronously writes them in the same order to the appropriate Kafka topic. Kafka Connect uses the term *offset* for the source-specific position information that Debezium includes with each event, and Kafka Connect periodically records the most recent offset in another Kafka topic.

When Kafka Connect gracefully shuts down, it stops the connectors, flushes all events to Kafka, and records the last offset received from each connector. Upon restart, Kafka Connect reads the last recorded offset for each connector, and starts the connector from that point. The PostgreSQL connector uses the LSN recorded in each change event as the offset, so that upon restart the connector requests the PostgreSQL server send it the events starting just after that position.

NOTE

The PostgreSQL connector retrieves the schema information as part of the events sent by the logical decoder plug-in. The only exception is the information about which columns compose the primary key, as this information is obtained from the JDBC metadata (side channel). If the primary key definition of a table changes (by adding, removing or renaming PK columns), then there exists a slight risk of an unfortunate timing when the primary key information from JDBC will not be synchronized with the change data in the logical decoding event and a small amount of messages will be created with an inconsistent key structure. If this happens then a restart of the connector and a reprocessing of the messages will fix the issue. To prevent the issue completely it is recommended to synchronize updates to the primary key structure with Debezium roughly using following sequence of operations:

- Put the database or an application into a read-only mode
- Let Debezium process all remaining events
- Stop Debezium
- Update the primary key definition
- Put the database or the application into read/write state and start Debezium again

3.2.4.3. PostgreSQL 10+ Logical Decoding Support (pgoutput)

As of PostgreSQL 10+, a new logical replication stream mode was introduced, called *pgoutput*. This logical replication stream mode is natively supported by PostgreSQL, which means that this connector can consume that replication stream without the need for additional plug-ins being installed. This is particularly valuable for environments where installation of plug-ins is not supported or allowed.

See Setting up PostgreSQL for more details.

3.2.4.4. Topics Names

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** configuration property, *schemaName* is the name of the database schema where the operation occurred, and *tableName* is the name of the database table on which the operation occurred.

For example, consider a PostgreSQL installation with a **postgres** database and an **inventory** schema that contains four tables: **products**, **products_on_hand**, **customers**, and **orders**. If the connector monitoring this database were given a logical server 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**

If on the other hand the tables were not part of a specific schema but rather created in the default **public** PostgreSQL schema, then the name of the Kafka topics would be:

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

3.2.4.5. Meta Information

Each **record** produced by the PostgreSQL connector has, in addition to the *database event*, some meta-information about 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 be placed:

```
"sourcePartition": {
    "server": "fulfillment"
 },
 "sourceOffset": {
    "lsn": "24023128",
    "txId": "555",
    "ts_ms": "1482918357011"
 },
 "kafkaPartition": null
```

The PostgreSQL connector uses only 1 Kafka Connect *partition* and it places the generated events into 1 Kafka partition. Therefore, the name of the **sourcePartition** will always default to the name of the **database.server.name** configuration property, while the **kafkaPartition** has the value **null** which means that the connector does not use a specific Kafka partition.

The **sourceOffset** portion of the message 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 which caused the event
- **ts_ms** represents the number of microseconds since Unix Epoch as the server time at which the transaction was committed

3.2.4.6. Events

All data change events produced by the PostgreSQL connector have a key and a value, although the structure of the key and value depend on the table from which the change events originated (see Topic names).

NOTE

Starting with Kafka 0.10, Kafka can optionally record with the message key and value 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* are valid Avro schema names. This means that the logical server name must start with Latin letters or an underscore (e.g., [a-z,A-Z,_]), and the remaining characters in the logical server name and all characters in the schema and table names must be Latin letters, digits, or an underscore (e.g., [a-z,A-Z,0-9,_]). If not, then all invalid characters will automatically be replaced with an underscore character.

This can lead to unexpected conflicts when the logical server name, schema names, and table names contain other characters, and the only distinguishing characters between table full names are invalid and thus replaced with underscores.

Debezium and Kafka Connect are designed around *continuous streams of event messages*, and the structure of these events may change over time. This could be difficult for consumers to deal with, so to make it easy Kafka Connect makes each event self-contained. Every message key and value has two parts: a *schema* and *payload*. The schema describes the structure of the payload, while the payload contains the actual data.

3.2.4.6.1. Change Event’s Key

For a given table, the change event’s key will have a structure that contains a field for each column in the primary key (or unique key constraint with **REPLICA IDENTITY** set to **FULL** or **USING INDEX** on the table) of the table at the time the event was created.

Consider a **customers** table defined in the **public** database schema:

```
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)
);
```

If the **database.server.name** configuration property has the value **PostgreSQL_server**, every change event for the **customers** table while it has this definition will feature the same key structure, which in JSON looks like this:

```
{
  "schema": {
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key",
    "optional": false,
    "fields": [
          {
              "name": "id",
              "index": "0",
              "schema": {
                  "type": "INT32",
                  "optional": "false"
```

```
            }
          }
        ]
      },
      "payload": {
        "id": "1"
      },
    }
```

The **schema** portion of the key contains a Kafka Connect schema describing what is in the key portion. In this case, it means that the **payload** value is not optional, is a structure defined by a schema named **PostgreSQL_server.public.customers.Key**, and has one required field named **id** of type **int32**. If you look at the value of the key's **payload** field, you see that it is indeed a structure (which in JSON is just an object) with a single **id** field, whose value is **1**.

Therefore, we interpret this key as describing the row in the **public.customers** table (output from the connector named **PostgreSQL_server**) whose **id** primary key column had a value of **1**.

NOTE

Although the **column.blacklist** configuration property allows you to capture only a subset of table columns, all columns in a primary or unique key are always included in the event's key.

WARNING

If the table does not have a primary or unique key, then the change event's key will be null. This makes sense since the rows in a table without a primary or unique key constraint cannot be uniquely identified.

3.2.4.6.2. Change Event's Value

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

- **op** is a mandatory field that contains a string value describing the type of operation. Values for the PostgreSQL connector are **c** for create (or insert), **u** for update, **d** for delete, and **r** for read (in the case of a snapshot).
- **before** is an optional field that if present contains the state of the row *before* the event occurred. The structure will be described by the **PostgreSQL_server.public.customers.Value** Kafka Connect schema, which the **PostgreSQL_server** connector uses for all rows in the **public.customers** table.

WARNING

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

- **after** is an optional field that if present contains the state of the row *after* the event occurred. The structure is described by the same **PostgreSQL_server.public.customers.Value** Kafka Connect schema used in **before**.
- **source** is a mandatory field that contains a structure describing the source metadata for the event, which in the case of PostgreSQL contains several fields: the Debezium version, the connector name, the name of the affected database, schema and table, whether the event is part of an ongoing snapshot or not and the same fields from the record's *meta information* section
- **ts_ms** is optional and if present contains the time (using the system clock in the JVM running the Kafka Connect task) at which the connector processed the event.

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

3.2.4.6.3. Replica Identity

REPLICA IDENTITY is a PostgreSQL specific table-level setting which determines the amount of information that is available to **logical decoding** in case of **UPDATE** and **DELETE** events. More specifically, this controls what (if any) information is available regarding the previous values of the table columns involved, whenever one of the aforementioned events occur.

There are 4 possible values for **REPLICA IDENTITY**:

- DEFAULT - **UPDATE** and **DELETE** events will only contain the previous values for the primary key columns of a table, in case of **UPDATE** only the primary columns with changed values are present
- NOTHING - **UPDATE** and **DELETE** events will not contain any information about the previous value on any of the table columns
- FULL - **UPDATE** and **DELETE** events will contain the previous values of all the table's columns
- INDEX **index name** - **UPDATE** and **DELETE** events will contain the previous values of the columns contained in the index definition named **index name**, in case of **UPDATE** only the indexed columns with changed values are present

3.2.4.6.4. Create Events

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

```
{
  "schema": {
    "type": "struct",
    "fields": [
{
    "type": "struct",
    "fields": [
        {
            "type": "int32",
            "optional": false,
            "field": "id"
        },
        {
            "type": "string",
            "optional": false,
            "field": "first_name"
        },
        {
            "type": "string",
            "optional": false,
            "field": "last_name"
        },
        {
            "type": "string",
            "optional": false,
            "field": "email"
        }
    ],
    "optional": true,
    "name": "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"
},
    {
      "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"
},
"payload": {
    "before": null,
    "after": {
        "id": 1,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
    },
    "source": {
        "version": "1.1.2.Final",
        "connector": "postgresql",
        "name": "PostgreSQL_server",
        "ts_ms": 1559033904863,
        "snapshot": true,
        "db": "postgres",
        "schema": "public",
        "table": "customers",
        "txId": 555,
        "lsn": 24023128,
        "xmin": null
    },
    "op": "c",
    "ts_ms": 1559033904863
}
}
```

If we look at the **schema** portion of this event's *value*, we can see the schema for the *envelope*, the schema for the **source** structure (which is specific to the PostgreSQL connector and reused across all events), and the table-specific schemas for the **before** and **after** fields.

NOTE

The names of the schemas for the **before** and **after** fields are of the form *logicalName*.*schemaName*.*tableName*.Value, and thus are entirely independent from all other schemas for all other tables.

This means that when using the Avro Converter, the resulting Avro schemas for *each table* in each *logical source* have their own evolution and history.

If we look at the **payload** portion of this event's *value*, we can see the information in the event, namely that it is describing that the row was created (since **op=c**), and that the **after** field value contains the values of the new inserted row's' **id**, **first_name**, **last_name**, and **email** columns.

NOTE

It may appear that the JSON representations of the events are much larger than the rows they describe. This is true, because the JSON representation must include the *schema* and the *payload* portions of the message.

It is possible and even recommended to use the Avro Converter to dramatically decrease the size of the actual messages written to the Kafka topics.

3.2.4.6.5. Update Events

The value of an *update* change event on this table will actually have the exact same *schema*, and its payload will be structured the same but will hold different values. Here's an example:

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1
    },
    "after": {
      "id": 1,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.1.2.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": null,
      "db": "postgres",
      "schema": "public",
      "table": "customers",
      "txId": 556,
      "lsn": 24023128,
      "xmin": null
    },
    "op": "u",
        "ts_ms": 1465584025523
    }
}
```

When we compare this to the value in the *insert* event, we see a couple of differences in the **payload** section:

- The **op** field value is now **u**, signifying that this row changed because of an update
- The **before** field now has the state of the row with the values before the database commit, but only for the primary key column **id**. This is because the *REPLICA IDENTITY* which is by default **DEFAULT**.

NOTE

Should we want to see the previous values of all the columns for the row, we would have to change the **customers** table first by running **ALTER TABLE customers REPLICA IDENTITY FULL**

- The **after** field now has the updated state of the row, and here was can see that the **first_name** value is now **Anne Marie**.
- The **source** field structure has the same fields as before, but the values are different since this event is from a different position in the WAL.
- The **ts_ms** shows the timestamp that Debezium processed this event.

There are several things we can learn by just looking at this **payload** section. We can compare the **before** and **after** structures to determine what actually changed in this row because of the commit. The **source** structure tells us information about PostgreSQL's record of this change (providing traceability), but more importantly this has information we can compare to other events in this and other topics to know whether this event occurred before, after, or as part of the same PostgreSQL commit as other events.

NOTE

When the columns for a row's primary/unique key are updated, the value of the row's key has changed so Debezium will output *three* events: a **DELETE** event and tombstone event with the old key for the row, followed by an **INSERT** event with the new key for the row.

3.2.4.6.6. Delete Events

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

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": null,
        "source": {
                "version": "1.1.2.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
        }
    }
```

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

- The **op** field value is now **d**, signifying that this row was deleted
- The **before** field now has the state of the row that was deleted with the database commit. Again this only contains the primary key column due to the *REPLICA IDENTITY* setting
- The **after** field is null, signifying that the row no longer exists
- The **source** field structure has many of the same values as before, except the **ts_ms**, **lsn** and **txId** fields have changed
- The **ts_ms** shows the timestamp that Debezium processed this event.

This event gives a consumer all kinds of information that it can use to process the removal of this row.

WARNING

Please pay attention to the tables without PK, any delete messages from such table with REPLICA IDENTITY DEFAULT will have no **before** part (because they have no PK which is the only field for the default identity level) and therefore will be skipped as totally empty. To be able to process messages from tables without PK set REPLICA IDENTITY to FULL level.

The PostgreSQL connector's events are designed to work with Kafka log compaction, which allows for the removal of some older messages as long as at least the most recent message for every key is kept. This allows Kafka to reclaim storage space while ensuring the topic contains a complete dataset and can be used for reloading key-based state.

When a row is deleted, the *delete* event value listed above still works with log compaction, since Kafka can still remove all earlier messages with that same key. But only if the message value is **null** will Kafka know that it can remove *all messages* with that same key. To make this possible, the PostgreSQL connector always follows the *delete* event with a special *tombstone* event that has the same key but **null** value.

3.2.5. Transaction Metadata

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

3.2.5.1. 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 emmitted 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:

```
{
  "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
    }
  ]
}
```

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

3.2.5.2. 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:

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

3.2.5.3. Data Types

As described above, the PostgreSQL connector represents the changes to rows with events that are structured like the table in which the row exist. The event contains a field for each column value, and how that value is represented in the event depends on the PostgreSQL data type of the column. This section describes this mapping.

The following table describes how the connector maps each of the PostgreSQL data types to a *literal type* and *semantic type* within the events' fields.

Here, the *literal type* describes how the value is literally represented using Kafka Connect schema types, namely **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**.

The *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.

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
BOOLEAN	**BOOLEAN**	n/a	
BIT(1)	**BOOLEAN**	n/a	

BIT(> 1), **BIT VARYING[(M)]**	**BYTES**	**io.debezium.data.Bits**	The **length** schema parameter contains an integer representing the number of bits. The resulting **byte[]** will contain the bits in little-endian form and will be sized to contain at least the specified number of bits (e.g., **numBytes = n/8 + (n%8== 0 ? 0 : 1)** where **n** is the number of bits).
SMALLINT, **SMALLSERIAL**	**INT16**	n/a	
INTEGER, **SERIAL**	**INT32**	n/a	
BIGINT, **BIGSERIAL**	**INT64**	n/a	
REAL	**FLOAT32**	n/a	
DOUBLE PRECISION	**FLOAT64**	n/a	
CHAR[(M)]	**STRING**	n/a	
VARCHAR[(M)]	**STRING**	n/a	
CHARACTER[(M)]	**STRING**	n/a	
CHARACTER VARYING[(M)]	**STRING**	n/a	
TIMESTAMPTZ, **TIMESTAMP WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTimestamp**	A string representation of a timestamp with timezone information, where the timezone is GMT
TIMETZ, **TIME WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTime**	A string representation of a time value with timezone information, where the timezone is GMT
INTERVAL [P]	**INT64**	**io.debezium.time.MicroDuration** (default)	The approximate number of microseconds for a time interval using the **365.25** / **12.0** formula for days per month average

INTERVAL [P]	**String**	**io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**)	The string representation of the interval value that follows pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, e.g. **P1Y2M3DT4H5M6.78S**
BYTEA	**BYTES**	n/a	
JSON, **JSONB**	**STRING**	**io.debezium.data.Json**	Contains the string representation of a JSON document, array, or scalar.
XML	**STRING**	**io.debezium.data.Xml**	Contains the string representation of an XML document
UUID	**STRING**	**io.debezium.data.Uuid**	Contains the string representation of a PostgreSQL UUID value
POINT	**STRUCT**	**io.debezium.data.geometry.Point**	Contains a structure with 2 **FLOAT64** fields - **(x,y)** - each representing the coordinates of a geometric point
LTREE	**STRING**	**io.debezium.data.Ltree**	Contains the string representation of a PostgreSQL LTREE value
CITEXT	**STRING**	n/a	
INET	**STRING**	n/a	
INT4RANGE	**STRING**	n/a	Range of integer
INT8RANGE	**STRING**	n/a	Range of bigint
NUMRANGE	**STRING**	n/a	Range of numeric
TSRANGE	**STRING**	n/a	Contains the string representation of timestamp range without time zone.
TSTZRANGE	**STRING**	n/a	Contains the string representation of a timestamp range with (local system) time zone.
DATERANGE	**STRING**	n/a	Contains the string representation of a date range. It always has an exclusive upper-bound.

ENUM	**STRING**	**io.debezium.data.Enum**	Contains the string representation of the PostgreSQL ENUM value. The set of allowed values are maintained in the schema parameter named **allowed**.

Other data type mappings are described in the following sections.

3.2.5.3.1. Temporal Values

Other than PostgreSQL's **TIMESTAMPTZ** and **TIMETZ** data types (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:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
DATE	**INT32**	**io.debezium.time.Date**	Represents the number of days since epoch.
TIME(1), **TIME(2)**, **TIME(3)**	**INT32**	**io.debezium.time.Time**	Represents the number of milliseconds past midnight, and does not include timezone information.
TIME(4), **TIME(5)**, **TIME(6)**	**INT64**	**io.debezium.time.MicroTime**	Represents the number of microseconds past midnight, and does not include timezone information.
TIMESTAMP(1), **TIMESTAMP(2)**, **TIMESTAMP(3)**	**INT64**	**io.debezium.time.Timestamp**	Represents the number of milliseconds past 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 epoch, and does not include timezone information.

When the **time.precision.mode** configuration property is set to **adaptive_time_microseconds**, 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, except that all TIME fields will be captured as microseconds:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes

DATE	**INT32**	**io.debezium.time.Date**	Represents the number of days since epoch.
TIME([P])	**INT64**	**io.debezium.time.MicroTime**	Represents the time value in microseconds and does not include timezone information. PostgreSQL allows precision **P** to be in the range 0-6 to store up to microsecond precision.
TIMESTAMP(1) , **TIMESTAMP(2)**, **TIMESTAMP(3)**	**INT64**	**io.debezium.time.Timestamp**	Represents the number of milliseconds past 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 epoch, and does not include timezone information.

When the **time.precision.mode** configuration property is set to **connect**, then the connector will use the predefined Kafka Connect logical types. This may be useful when consumers only know about the built-in Kafka Connect logical types and are unable to handle variable-precision time values. On the other hand, since PostgreSQL supports 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:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
DATE	**INT32**	**org.apache.kafka.connect.data.Date**	Represents the number of days since 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** > 3.
TIMESTAMP([P])	**INT64**	**org.apache.kafka.connect.data.Timestamp**	Represents the number of milliseconds since 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** > 3.

3.2.5.3.2. **TIMESTAMP** values

The **TIMESTAMP** type represents a timestamp without time zone information. Such columns are converted into an equivalent Kafka Connect value based on UTC. So for instance the **TIMESTAMP** value "2018-06-20 15:13:16.945104" will be represented by a **io.debezium.time.MicroTimestamp** with the value "1529507596945104" (assuming **time.precision.mode** is not set to **connect**).

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

3.2.5.3.3. Decimal Values

When **decimal.handling.mode** configuration property is set to **precise**, then the connector will use the predefined Kafka Connect **org.apache.kafka.connect.data.Decimal** logical type for all **DECIMAL** and **NUMERIC** columns. This is the default mode.

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
NUMERIC[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scaled** schema parameter contains an integer representing how many digits the decimal point was shifted.
DECIMAL[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scaled** schema parameter contains an integer representing how many digits the decimal point was shifted.

There is an exception to this rule. When the **NUMERIC** or **DECIMAL** types are used without any scale constraints then it means that the values coming from the database have a different (variable) scale for each value. In this case a type **io.debezium.data.VariableScaleDecimal** is used and it contains both value and scale of the transferred value.

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
NUMERIC	**STRUCT**	**io.debezium.data.VariableScaleDecimal**	Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.
DECIMAL	**STRUCT**	**io.debezium.data.VariableScaleDecimal**	Contains a structure with two fields: **scale** of type **INT32** that contains the scale of the transferred value and **value** of type **BYTES** containing the original value in an unscaled form.

However, when **decimal.handling.mode** configuration property is set to **double**, then the connector will represent all **DECIMAL** and **NUMERIC** values as Java double values and encodes them as follows:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
NUMERIC[(M[,D])]	**FLOAT64**		
DECIMAL[(M[,D])]	**FLOAT64**		

The last option for **decimal.handling.mode** configuration property is **string**. In this case the connector will represent all **DECIMAL** and **NUMERIC** values as their formatted string representation and encodes them as follows:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
NUMERIC[(M[,D])]	**STRING**		
DECIMAL[(M[,D])]	**STRING**		

PostgreSQL supports **NaN** (not a number) special value to be stored in the **DECIMAL**/**NUMERIC** values. Only **string** and **double** modes are able to handle such values encoding them as either **Double.NaN** or string constant **NAN**.

3.2.5.3.4. HStore Values

When **hstore.handling.mode** configuration property is set to **json** (the default), the connector will represent all **HSTORE** values as string-ified JSON values and encode them as follows:

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
HSTORE	**STRING**	**io.debezium.data.Json**	Example: output representation using the JSON converter is **{\"key\" : \"val\"}**

When **hstore.handling.mode** configuration property is set to **map**, then the connector will use the **MAP** schema type for all **HSTORE** columns.

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
HSTORE	**MAP**		Example: output representation using the JSON converter is **{"key" : "val"}**

3.2.5.4. PostgreSQL Domain Types

PostgreSQL also supports the notion of user-defined types that are based upon other underlying types. When such column types are used, Debezium exposes the column's representation based on the full type hierarchy.

IMPORTANT

Special consideration should be taken when monitoring columns that use domain types.

When a column is defined using a domain type that extends one of the default database types and the domain type defines a custom length/scale, the generated schema will inherit that defined length/scale.

When a column is defined using a domain type that extends another domain type that defines a custom length/scale, the generated schema will **not** inherit the defined length/scale because the PostgreSQL driver's column metadata implementation.

3.2.5.4.1. Network Address Types

PostgreSQL also have data types that can store IPv4, IPv6, and MAC addresses. It is better to use these instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions.

PostgreSQL Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
INET	**STRING**		IPv4 and IPv6 networks
CIDR	**STRING**		IPv4 and IPv6 hosts and networks
MACADDR	**STRING**		MAC addresses
MACADDR 8	**STRING**		MAC addresses in EUI-64 format

3.2.5.4.2. PostGIS Types

The PostgreSQL connector also has full support for all of the PostGIS data types

PostGIS Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
GEOMETRY (planar)	**STRUCT**	**io.debezium.data.geometry.Geometry**	Contains a structure with 2 fields • **srid (INT32)** - Spatial Reference System Identifier defining what type of geometry object is stored in the structure • **wkb (BYTES)** - a binary representation of the geometry object encoded in the Well-Known-Binary format. Please see Open Geospatial Consortium Simple Features Access specification for the format details.
GEOGRAPHY (spherical)	**STRUCT**	**io.debezium.data.geometry.Geography**	Contains a structure with 2 fields • **srid (INT32)** - Spatial Reference System Identifier defining what type of geography object is stored in the structure • **wkb (BYTES)** - a binary representation of the geometry object encoded in the Well-Known-Binary format. Please see Open Geospatial Consortium Simple Features Access specification for the format details.

3.2.5.4.3. Toasted values

PostgreSQL has a hard limit on the page size. This means that values larger than ca. 8 KB need to be stored using TOAST storage. This impacts replication messages coming from database, as the values that were stored using the TOAST mechanism and 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 database, as this would lead into race conditions potentially. Debezium thus follows these rules to handle the toasted values:

- tables with **REPLICA IDENTITY FULL**: TOAST column values are part of the **before** and **after** blocks of change events as any other column
- tables with **REPLICA IDENTITY DEFAULT**: when receiving an **UPDATE** event from the database, any unchanged TOAST column value which is not part of the replica identity will not be part of that event; similarly, when receiving a **DELETE** event, any such TOAST column will not be part of the **before** block. As Debezium cannot safely provide the column value in this case, it returns a placeholder value defined in configuration option **toasted.value.placeholder**.

IMPORTANT

There is a specific problem related to Amazon RDS instances. **wal2json** plug-in has evolved over the time and there were releases that provided out-of-band toasted values. Amazon supports different versions of the plug-in for different PostgreSQL versions. Please consult Amazon's documentation to obtain version to version mapping. For consistent toasted values handling we recommend to

- use **pgoutput** plug-in for PostgreSQL 10+ instances
- set **include-unchanged-toast=0** for older versions of the **wal2json** plug-in by using the **slot.stream.params** configuration option

3.3. DEPLOYING THE POSTGRESQL CONNECTOR

Installing the PostgreSQL connector is a simple process whereby you only need to download the JAR, extract it to your Kafka Connect environment, and ensure the plug-in's parent directory is specified in your Kafka Connect environment.

Prerequisites

- You have Zookeeper, Kafka, and Kafka Connect installed.
- You have PostgreSQL installed and setup.

Procedure

1. Download the Debezium 1.1.3 PostgreSQL connector.
2. Extract the files into your Kafka Connect environment.
3. Add the plug-in's parent directory to your Kafka Connect **plugin.path**:

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

NOTE

The above example assumes you have extracted the Debezium PostgreSQL connector to the **/kafka/connect/Debezium-connector-postgresql** path.

4. Restart your Kafka Connect process. This ensures the new JARs are picked up.

Additional resources

For more information on the deployment process, and deploying connectors with AMQ Streams, refer to the Debezium installation guides.

- Installing Debezium on OpenShift
- Installing Debezium on RHEL

3.3.1. Example Configuration

To use the connector to produce change events for a particular PostgreSQL server or cluster:

1. Install the logical decoding plug-in
2. Configure the PostgreSQL server to support logical replication
3. Create a configuration file for the PostgreSQL connector.

When the connector starts, it will grab a consistent snapshot of the databases in your PostgreSQL server and start streaming changes, producing events for every inserted, updated, and deleted row. You can also choose to produce events for a subset of the schemas and tables. Optionally ignore, mask, or truncate columns that are sensitive, too large, or not needed.

Following is an example of the configuration for a PostgreSQL connector that monitors a PostgreSQL server at port 5432 on 192.168.99.100, which we logically name **fullfillment**. Typically, you configure the Debezium PostgreSQL connector in a **.yaml** file using the configuration properties available for the connector.

```
apiVersion: kafka.strimzi.io/v1beta1
  kind: KafkaConnector
  metadata:
    name: inventory-connector  (1)
    labels: strimzi.io/cluster: my-connect-cluster
  spec:
    class: io.debezium.connector.postgresql.PostgresConnector
    tasksMax: 1  (2)
    config:  (3)
      database.hostname: postgresqldb  (4)
      database.port: 5432
      database.user: debezium
      database.password: dbz
      database.dbname: postgres
      database.server.name: fullfillment  (5)
      database.whitelist: public.inventory  (6)
```

(1) The name of the connector.

(2) Only one task should operate at any one time. Because the PostgreSQL connector reads the PostgreSQL server's **binlog**, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.

(3) The connector's configuration.

(4) The database host, which is the name of the container running the PostgreSQL server (**postgresqldb**).

(5) A unique server name. The server name is the logical identifier for the PostgreSQL server or cluster of servers. This name will be used as the prefix for all Kafka topics.

(6) Only changes in the **public.inventory** database will be detected.

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 PostgreSQL database and record events to Kafka topics.

3.3.2. Monitoring

The Debezium PostgreSQL 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 change events via logical decoding

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

3.3.2.1. Snapshot Metrics

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

Attribute Name	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.

TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.

3.3.2.2. Streaming Metrics

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

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

NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**map<string, string>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

3.3.3. Connector Properties

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

Property	Default	Description

name		Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.postgresql.PostgresConnector** for the PostgreSQL connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The PostgreSQL connector always uses a single task and therefore does not use this value, so the default is always acceptable.
plugin.name	**decoderbufs**	The name of the Postgres logical decoding plug-in installed on the server. The only supported value is **pgoutput**. When the processed transactions are very large it is possible that the **JSON** batch event with all changes in the transaction will not fit into the hard-coded memory buffer of size 1 GB. In such cases it is possible to switch to so-called **streaming** mode when every change in transactions is sent as a separate message from PostgreSQL into Debezium.
slot.name	**debezium**	The name of the Postgres logical decoding slot created for streaming changes from a plug-in and database instance. Values must conform to Postgres replication slot naming ruleswhich state: *"Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character."*
slot.drop.on.stop	**false**	Whether or not to drop the logical replication slot when the connector finishes orderly. Should only be set to **true** in testing or development environments. Dropping the slot allows WAL segments to be discarded by the database, so it may happen that after a restart the connector cannot resume from the WAL position where it left off before.

publication.name	**dbz_publication**	The name of the PostgreSQL publication created created for streaming changes when using **pgoutput**. This publication is created at start-up if it does not already exist to include *all tables*. Debezium will then use its own white-/blacklist filtering capabilities to limit change events to the specific tables of interest if configured. Note the connector user must have superuser permissions in order to create this publication, so it is usually preferable to create the publication upfront. If the publication already exists (either for all tables or configured with a subset of tables), Debezium will instead use the publication as defined.
database.hostname		IP address or hostname of the PostgreSQL database server.
database.port	**5432**	Integer port number of the PostgreSQL database server.
database.user		Name of the PostgreSQL database to use when connecting to the PostgreSQL database server.
database.password		Password to use when connecting to the PostgreSQL database server.
database.dbname		The name of the PostgreSQL database from which to stream the changes
database.server.name		Logical name that identifies and provides a namespace for the particular PostgreSQL database server/cluster being monitored. The logical name should be unique across all other connectors, since it is used as a prefix for all Kafka topic names coming from this connector. Only alphanumeric characters and underscores should be used.
schema.whitelist		An optional comma-separated list of regular expressions that match schema names to be monitored; any schema name not included in the whitelist will be excluded from monitoring. By default all non-system schemas will be monitored. May not be used with **schema.blacklist**.

schema.blacklist		An optional comma-separated list of regular expressions that match schema names to be excluded from monitoring; any schema name not included in the blacklist will be monitored, with the exception of system schemas. May not be used with **schema.whitelist**.
table.whitelist		An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be monitored; any table not included in the whitelist will be excluded from monitoring. Each identifier is of the form *schemaName*.*tableName*. By default the connector will monitor every non-system table in each monitored schema. May not be used with **table.blacklist**.
table.blacklist		An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be excluded from monitoring; any table not included in the blacklist will be monitored. Each identifier is of the form *schemaName*.*tableName*. May not be used with **table.whitelist**.
column.blacklist		An optional comma-separated list of regular expressions that match the fully-qualified names of columns that should be excluded from change event message values. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*.

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; **adaptive_time_microseconds** captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type, with the exception of TIME type fields, which are always captured as microseconds; or **connect** always represents time and timestamp values using Kafka Connect's built-in representations for Time, Date, and Timestamp, which uses millisecond precision regardless of the database columns' precision. See temporal values.
decimal.handling.mode	**precise**	Specifies how the connector should handle values for **DECIMAL** and **NUMERIC** columns: **precise** (the default) represents them precisely using **java.math.BigDecimal** values represented in change events in a binary form; or **double** represents them using **double** values, which may result in a loss of precision but will be far easier to use. **string** option encodes values as formatted string which is easy to consume but a semantic information about the real type is lost. See Section 3.2.5.3.3, "Decimal Values".
hstore.handling.mode	**map**	Specifies how the connector should handle values for **hstore** columns: **map** (the default) represents using **MAP**; or **json** represents them using **json string**.**json** option encodes values as formatted string such as **{"key" : "val"}**. See Section 3.2.5.3.4, "HStore Values".
interval.handling.mode	**numeric**	Specifies how the connector should handle values for **interval** columns: **numeric** (the default) represents interval using approximate number of microseconds; **string** represents them exactly, using the string pattern representation **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, e.g. **P1Y2M3DT4H5M6.78S**. See Section 3.2.5.3, "Data Types".

database.sslmode	**disable**	Whether to use an encrypted connection to the PostgreSQL server. Options include: **disable** (the default) to use an unencrypted connection ; **require** to use a secure (encrypted) connection, and fail if one cannot be established; **verify-ca** like **require** but additionally verify the server TLS certificate against the configured Certificate Authority (CA) certificates, or fail if no valid matching CA certificates are found; **verify-full** like **verify-ca** but additionally verify that the server certificate matches the host to which the connection is attempted. See the PostgreSQL documentation for more information.
database.sslcert		The path to the file containing the SSL Certificate for the client. See the PostgreSQL documentation for more information.
database.sslkey		The path to the file containing the SSL private key of the client. See the PostgreSQL documentation for more information.
database.sslpassword		The password to access the client private key from the file specified by **database.sslkey**. See the PostgreSQL documentation for more information.
database.sslrootcert		The path to the file containing the root certificate(s) against which the server is validated. See the PostgreSQL documentation for more information.
database.tcpKeepAlive		Enable TCP keep-alive probe to verify that database connection is still alive. (enabled by default). See the PostgreSQL documentation for more information.
tombstones.on.delete	**true**	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.

column.truncate.to.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event message values if the field values are longer than the specified number of characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*.
column.mask.with.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*****) characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*.
column.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** 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 names for columns are of the form *databaseName*.*tableName*.*columnName*, or *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 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 *databaseName*.*tableName*.*typeName*, or *databaseName*.*schemaName*.*tableName*.*typeName*. See the list of PostgreSQL-specific data type names.
message.key.columns	*empty string*	A semi-colon list of regular expressions that match fully-qualified tables and columns to map a primary key. Each item (regular expression) must match the fully-qualified **<fully-qualified table>:<a comma-separated list of columns>** representing the custom key. Fully-qualified tables could be defined as *schemaName*.*tableName*.

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

Property	Default	Description

snapshot.mode	**initial**	Specifies the criteria for running a snapshot upon startup of the connector. The default is **initial**, and specifies the connector can run a snapshot only when no offsets have been recorded for the logical server name. The **always** option specifies that the connector run a snapshot each time on startup. The **never** option specifies that the connect should never use snapshots and that upon first startup with a logical server name the connector should read from either from where it last left off (last LSN position) or start from the beginning from the point of the view of the logical replication slot. The **initial_only** option specifies that the connector should only take an initial snapshot and then stop, without processing any subsequent changes. The **exported** option specifies that the database snapshot will be based on the point in time when the replication slot was created and is an excellent way to perform the snapshot in a lock-free way.
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 table locks cannot be acquired in this time interval, the snapshot will fail. See snapshots
snapshot.select.statement.overrides		Controls which rows from tables will be included in snapshot. This property contains a comma-separated list of fully-qualified tables *(DB_NAME.TABLE_NAME)*. Select statements for the individual tables are specified in further configuration properties, one for each table, identified by the id **snapshot.select.statement.overrides.[DB_NAME].[TABLE_NAME]**. The value of those properties is the SELECT statement to use when retrieving data from the specific table during snapshotting. *A possible use case for large append-only tables is setting a specific point where to start (resume) snapshotting, in case a previous snapshotting was interrupted.* NOTE: This setting has impact on snapshots only. Events generated by logical decoder are not affected by it at all.

event.processing .failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events. **fail** will propagate the exception (indicating the offset of the problematic event), causing the connector to stop. **warn** will cause the problematic event to be skipped and the offset of the problematic event to be logged. **skip** will cause the problematic event to be skipped.
max.queue.size	**20240**	Positive integer value that specifies the maximum size of the blocking queue into which change events received via streaming replication are placed before they are written to Kafka. This queue can provide backpressure when, for example, writes to Kafka are slower or if Kafka is not available.
max.batch.size	**10240**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector.
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.

include.unknown.datatypes	**false**	When Debezium meets a field whose data type is unknown, then by default the field is omitted from the change event and a warning is logged. In some cases it may be preferable though to include the field and send it downstream to clients in the opaque binary representation so the clients will decode it themselves. Set to **false** to filter unknown data out of events and **true** to keep them in binary format. NOTE The clients risk backward compatibility issues. Not only may the database specific binary representation change between releases, but also when the datatype is supported by Debezium eventually, it will be sent downstream in a logical type, requiring adjustments by consumers. In general, when encountering unsupported data types, please file a feature request so that support can be added.
database.initial.statements		A semicolon separated list of SQL statements to be executed when a JDBC connection (not the transaction log reading connection) to the database is established. Use doubled semicolon (';;') to use a semicolon as a character and not as a delimiter. NOTE The connector may establish JDBC connections at its own discretion, so this should typically be used for configuration of session parameters only, but not for executing DML statements.

heartbeat.interval.ms	**0**	Controls how frequently heartbeat messages are sent. This property contains an interval in milli-seconds 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 tables are changed for a longer period of time. In such situation the connector would proceed to read the log from the database but never emit any change messages into Kafka, which in turn means that no offset updates will be committed to Kafka. This will cause the WAL files to be retained by the database longer than needed (as the connector actually has processed them already but never got a chance to flush the latest retrieved LSN to the database) and also may result in more change events to be re-sent after a connector restart. Set this parameter to **0** to not send heartbeat messages at all. Disabled by default.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.
heartbeat.action.query		If specified, this query will be executed upon every heartbeat against the source database. This can be used to overcome 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 any WAL records and thus acknowledging WAL positions with the database. Inserting records into some heartbeat table (which must have been created upfront) will allow the connector to receive changes from the low-traffic database and acknowledge their LSNs, preventing an unbounded WAL growth on the database host. Example: **INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')**

schema.refresh.mode	**columns_diff**	Specify the conditions that trigger a refresh of the in-memory schema for a table. **columns_diff** (the default) is the safest mode, ensuring 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 between it and the schema derived from the incoming message, unless unchanged TOASTable data fully accounts for the discrepancy. This setting can improve connector performance significantly if there are frequently-updated tables that have TOASTed data that are rarely part of these updates. However, it is possible for the in-memory schema to become outdated if TOASTable columns are dropped from the table.
snapshot.delay.ms		An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size	**10240**	Specifies the maximum number of rows that should be read in one go from each table while taking a snapshot. The connector will read the table contents in multiple batches of this size. Defaults to 10240.
slot.stream.params		Optional list of parameters to be passed to the configured logical decoding plug-in. For example, **add-tables=public.table,public.table2;include-lsn=true**.
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names will be sanitized to adhere to Avro naming requirements. See Avro naming for more details.

slot.max.retries	6	How many times to retry connecting to a replication slot when an attempt fails.
slot.retry.delay.ms	10000 (10 seconds)	The number of milli-seconds to wait between retry attempts when the connector fails to connect to a replication slot.
toasted.value.placeholder	**__debezium_unavailable_value**	Specify the constant that will be provided by Debezium to indicate that the original value is a toasted value not provided by the database. If starts with **hex:** prefix it is expected that the rest of the string repesents hexadecimally encoded octets. See Toasted Values for additional details.
provide.transaction.metadata	**false**	When set to **true** Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.

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.)

3.4. POSTGRESQL 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.

3.4.1. 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, when the connector cannot successfully connect to PostgreSQL using the specified connectivity parameters, or when the connector is restarting from a previously-recorded position in the PostgreSQL WAL (via the LSN value) and PostgreSQL no longer has that history available.

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 PostgreSQL problem has been addressed.

3.4.2. PostgreSQL Becomes Unavailable

Once the connector is running, if the PostgreSQL server it has been connected to becomes unavailable for any reason, the connector will fail with an error and the connector will stop. Simply restart the connector when the server is available.

The PostgreSQL connector stores externally the last processed offset (in the form of a PostgreSQL **log sequence number** value). Once a connector is restarted and connects to a server instance, it will ask the server to continue streaming from that particular offset. This offset will always remain available so long as the Debezium replication slot remains intact. Never drop a replication slot on the primary or you will lose data. See the next section for failure cases when a slot has been removed.

3.4.3. Cluster Failures

As of **12**, PostgreSQL allows logical replication slots *only on primary servers*, which means that a PostgreSQL connector can only be pointed to the active primary of a database cluster. Also replication slots themselves are not propagated to replicas. If the primary node goes down, only after a new primary has been promoted (with the logical decoding plug-in installed) and a replication slot has been created there, the connector can be restarted and pointed to the new server.

There are some really important caveats to failovers, and you should pause Debezium until you can verify that you have a replication slot intact which has not lost data. After a failover, you will miss change events unless your administration of failovers includes a process to recreate the Debezium replication slot before the application is allowed to write to the **new** primary. You also may need to verify in a failover situation that Debezium was able to read all changes in the slot **before the old primary failed**

One reliable method of recovering and verifying any lost changes (yet administratively difficult) is to recover a backup of your failed primary to the point immediately before it failed, which would allow you to inspect the replication slot for any unconsumed changes. In any case, it is crucial that you recreate the replication slot on the new primary prior to allowing writes to it.

3.4.4. 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 will be a short delay in processing while the connector tasks are stopped gracefully and restarted on the new processes.

3.4.5. Kafka Connect Process Crashes

If the Kafka Connector process stops unexpectedly, then any connector tasks it was running will obviously 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 PostgreSQL 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 will depend on the offset flush period and the volume of data changes just before the crash.

NOTE

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

Debezium also includes with each change event message the source-specific information about the origin of the event, including the 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 position) to know whether they have already seen a particular event.

3.4.6. 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 you've 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 re-established, at which point the connectors will resume exactly where they left off.

3.4.7. Connector Is Stopped for a Duration

If the connector is gracefully stopped, the database can continue to be used and any new changes will be recorded in the PostgreSQL WAL. When the connector is restarted, it will resume streaming changes where it last left off, recording change events for all of the changes that were made while the connector was stopped.

A properly configured Kafka cluster is able to handle 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 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 an snapshot of all of the databases and collections on the primary, and then reads the primary's oplog to apply all changes that might have been made since it began the snapshot. This new server becomes a secondary (and able to handle queries) when it catches up to the tail of the primary's oplog.

The MongoDB connector uses this same replication mechanism, though it does not actually become a member of the replica set. Just like MongoDB secondaries, however, the connector always reads the oplog of the replica set's primary. And, when the connector sees a replica set for the first time, it looks at the oplog to get the last recorded transaction and then performs a snapshot of the primary's databases and collections. When all the data is copied, the connector then starts streaming changes from the position it read earlier from the oplog. Operations in the MongoDB oplog are idempotent, so no matter how many times the operations are applied, they result in the same end state.

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

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

Additional resources

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

4.2. 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 **initial.sync.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 trys 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. Events

All data change events produced by the MongoDB connector have a key and a value.

Debezium and Kafka Connect are designed around *continuous streams of event messages*, and the structure of these events could potentially change over time if the source of those events changed in structure or if the connector is improved or changed. This could be difficult for consumers to deal with, so to make it very easy Kafka Connect makes each event self-contained. Every message key and value has two parts: a *schema* and *payload*. The schema describes the structure of the payload, while the payload contains the actual data.

4.4.6.1. Change event's key

For a given collection, the change event's key contains a single **id** field. Its value is the document's identifier represented as string which is derived from the MongoDB extended JSON serialization in strict mode. Consider a connector with a logical name of **fulfillment**, a replica set containing an **inventory** database with a **customers** collection containing documents such as:

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

Every change event for the **customers** collection will feature the same key structure, which in JSON looks like this:

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

The **schema** portion of the key contains a Kafka Connect schema describing what is in the payload portion. In this case, it means that the **payload** value is not optional, is a structure defined by a schema named **fulfillment.inventory.customers.Key**, and has one required field named **id** of type **string**. If you look at the value of the key's **payload** field, you can see that it is indeed a structure (which in JSON is just an object) with a single **id** field, whose value is a string containing the integer **1004**.

This example used a document with an integer identifier, but any valid MongoDB document identifier (including documents) will work. The value of the **id** field in the payload will simply be a string representing a MongoDB extended JSON serialization (strict mode) of the original document's **_id** field. Find below a few examples showing how **_id** fields of different types will get encoded as the event key's payload:

Type	MongoDB **_id** Value	Key's payload

Type	MongoDB **_id** Value	Key's payload
Integer	1234	**{ "id" : "1234" }**
Float	12.34	**{ "id" : "12.34" }**
String	"1234"	**{ "id" : "\"1234\"" }**
Document	{ "hi" : "kafka", "nums" : [10.0, 100.0, 1000.0] }	**{ "id" : "{\"hi\" : \"kafka\", \"nums\" : [10.0, 100.0, 1000.0]}" }**
ObjectId	ObjectId("596e275826f08b2730779e1f")	**{ "id" : "{\"$oid\" : \"596e275826f08b2730779e1f\"}" }**
Binary	BinData("a2Fma2E=",0)	**{ "id" : "{\"$binary\" : \"a2Fma2E=\", \"$type\" : \"00\"}" }**

4.4.6.2. Change event's value

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

- **op** is a mandatory field that contains a string value describing the type of operation. Values for the MongoDB connector are **c** for create (or insert), **u** for update, **d** for delete, and **r** for read (in the case of a snapshot).
- **after** is an optional field that if present contains the state of the document *after* the event occurred. MongoDB's oplog entries only contain the full state of a document for *create* events, so these are the only events that contain an *after* field.
- **source** is a mandatory field that contains a structure describing the source metadata for the event, which in the case of MongoDB contains several fields: the Debezium version, the logical name, the replica set's name, the namespace of the collection, the MongoDB timestamp (and ordinal of the event within the timestamp) at which the event occurred, the identifier of the MongoDB operation (e.g., the **h** field in the oplog event), and the initial sync flag if the event resulted during a snapshot.
- **ts_ms** is optional and if present contains the time (using the system clock in the JVM running the Kafka Connect task) at which the connector processed the event.

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

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

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "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": "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": "1.1.2.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 31,
      "h": 1546547425148721999
    },
    "op": "r",
      "ts_ms": 1558965515240
    }
  }
```

If we look at the **schema** portion of this event's *value*, we can see the schema for the *envelope* is specific to the collection, and the schema for the **source** structure (which is specific to the MongoDB connector and reused across all events). Also note that the **after** value is always a string, and that by convention it will contain a JSON representation of the document.

If we look at the **payload** portion of this event's *value*, we can see the information in the event, namely that it is describing that the document was read as part of an snapshot (since **op=r** and **snapshot=true**), and that the **after** field value contains the JSON string representation of the document.

NOTE

It may appear that the JSON representations of the events are much larger than the rows they describe. This is true, because the JSON representation must include the *schema* and the *payload* portions of the message.

The value of an *update* change event on this collection will actually have the exact same *schema*, and its payload is structured the same but will hold different values. Specifically, an update event will not have an **after** value and will instead have a **patch** string containing the JSON representation of the idempotent update operation and a **filter** string containing the JSON representation of the selection criteria for the update. The **filter** string can include multiple shard key fields for sharded collections. Here's an example:

```
{
    "schema": { ... },
    "payload": {
      "op": "u",
      "ts_ms": 1465491461815,
      "patch": "{\"$set\":{\"first_name\":\"Anne Marie\"}}",
      "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}",
      "source": {
        "version": "1.1.2.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": true,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

When we compare this to the value in the *insert* event, we see a couple of differences in the **payload** section:

- The **op** field value is now **u**, signifying that this document changed because of an update

- The **patch** field appears and has the stringified JSON representation of the actual MongoDB idempotent change to the document, which in this example involves setting the **first_name** field to a new value
- The **filter** field appears and has the stringified JSON representation of the MongoDB selection criteria used for the update
- The **after** field no longer appears
- The **source** field structure has the same fields as before, but the values are different since this event is from a different position in the oplog
- The **ts_ms** shows the timestamp that Debezium processed this event

WARNING

The content of the patch field is provided by MongoDB itself and its exact format depends on the specific database version. You should therefore be prepared for potential changes to the format when upgrading the MongoDB instance to a new version.

All examples in this document were obtained from MongoDB 3.4 and might differ if you use a different one.

NOTE

Update events in MongoDB's oplog do not have the *before* or *after* states of the changed document, so there's no way for the connector to provide this information. However, because *create* or *read* events *do* contain the starting state, downstream consumers of the stream can actually fully-reconstruct the state by keeping the latest state for each document and applying each event to that state. Debezium connector's are not able to keep such state, so it is not able to do this.

So far, you have seen samples of *create*/*read* and *update* events. The following sample shows the value of a *delete* event for the same collection. The value of a *delete* event on this collection has the exact same *schema*, and its payload is structured the same but it holds different values. In particular, a delete event does not have an **after** value nor a **patch** value:

```
{
    "schema": { ... },
    "payload": {
      "op": "d",
      "ts_ms": 1465495462115,
      "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}",
      "source": {
        "version": "1.1.2.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": true,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

When we compare this to the value in the other events, we see a couple of differences in the **payload** section:

- The **op** field value is now **d**, signifying that this document was deleted
- The **patch** field does not appear
- The **after** field does not appear
- The **filter** field appears and has the stringified JSON representation of the MongoDB selection criteria used for the delete
- The **source** field structure has the same fields as before, but the values are different since this event is from a different position in the oplog
- The **ts_ms** shows the timestamp that Debezium processed this event

The MongoDB connector provides one other kind of event. Each *delete* event is followed by a *tombstone* event that has the same key as the *delete* event but a **null** value. This provides Kafka with the information needed to run its log compaction mechanism to remove *all* messages with that key.

NOTE

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

All MongoDB connector events for a uniquely identified document have exactly the same key, signaling to Kafka that only the latest event be kept. 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 tranaction 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 emmitted 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:

```
{
  "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": "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:

```
{
  "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"
    }
  }
```

4.5. DEPLOYING THE MONGODB CONNECTOR

Installing the MongoDB connector is a simple process whereby you only need to download the JAR, extract it to your Kafka Connect environment, and ensure the plug-in's parent directory is specified in your Kafka Connect environment.

Prerequisites

- You have Zookeeper, Kafka, and Kafka Connect installed.
- You have MongoDB installed and setup.

Procedure

1. Download the Debezium MongoDB connector.
2. Extract the files into your Kafka Connect environment.
3. Add the plug-in's parent directory to your Kafka Connect **plugin.path**:

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

NOTE

The above example assumes you have extracted the Debezium MongoDB connector to the /**kafka**/**connect**/**Debezium-connector-mongodb** path.

4. Restart your Kafka Connect process. This ensures the new JARs are picked up.

Additional resources

For more information on the deployment process, and deploying connectors with AMQ Streams, refer to 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 row. 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 **fullfillment**. Typically, you configure the Debezium MongoDB connector in a **.yaml** file using the configuration properties available for the connector.

```
apiVersion: kafka.strimzi.io/v1beta1
 kind: KafkaConnector
 metadata:
  name: inventory-connector (1)
  labels: strimzi.io/cluster: my-connect-cluster
 spec:
  class: io.debezium.connector.mongodb.MongoDbConnector (2)
  config:
   mongodb.hosts: rs0/192.168.99.100:27017 (3)
   mongodb.name: fulfillment (4)
   collection.whitelist: inventory[.]* (5)
```

1. The name of our connector when we register it with a Kafka Connect service.
2. The name of the MongoDB connector class.
3. The host addresses to use to connect to the MongoDB replica set.
4. The *logical name* of the MongoDB replica set, which forms a namespace for generated events and is used in all the names of the Kafka topics to which the connector writes, the Kafka Connect schema names, and the namespaces of the corresponding Avro schema when the Avro Connector is used.
5. 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. 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.

4.5.2.1. Snapshot Metrics

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

Attribute Name	Type	Description

LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of collections that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of collections that are being included in the snapshot.
RemainingTableCount	**int**	The number of collections that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.

RowsScanned	**Map<String, Long>**	Map containing the number of documents exported for each collection in the snapshot. Collections are incrementally added to the Map during processing. Updates every 10,000 documents scanned and upon completing a collection.

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

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

4.5.2.2. Streaming Metrics

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

Attribute Name	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of collections that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the streamer and the main Kafka Connect loop.

QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to mongodb.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event’s timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**map<string, string>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

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

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

4.5.3. Connector properties

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

Property	Default	Description

Property	Default	Description
name		Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.mongodb.MongoDbConnector** for the MongoDB connector.
mongodb.hosts		The comma-separated list of hostname and port pairs (in the form 'host' or 'host:port') of the MongoDB servers in the replica set. The list can contain a single hostname and port pair. If **mongodb.members.auto.discover** is set to **false**, then the host and port pair should be prefixed with the replica set name (e.g., **rs0/localhost:27017**).
mongodb.name		A unique name that identifies the connector and/or MongoDB replica set or sharded cluster that this connector monitors. Each server should be monitored by at most one Debezium connector, since this server name prefixes all persisted Kafka topics emanating from the MongoDB replica set or cluster. Only alphanumeric characters and underscores should be used.
mongodb.user		Name of the database user to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.
mongodb.password		Password to be used when connecting to MongoDB. This is required only when MongoDB is configured to use authentication.
mongodb.authsource	**admin**	Database (authentication source) containing MongoDB credentials. This is required only when MongoDB is configured to use authentication with another authentication database than **admin**.
mongodb.ssl.enabled	**false**	Connector will use SSL to connect to MongoDB instances.

Property	Default	Description
mongodb.ssl.invalid.hostname.allowed	**false**	When SSL is enabled this setting controls whether strict hostname checking is disabled during connection phase. If **true** the connection will not prevent man-in-the-middle attacks.
database.whitelist	*empty string*	An optional comma-separated list of regular expressions that match database names to be monitored; any database name not included in the whitelist is excluded from monitoring. By default all databases is monitored. May not be used with **database.blacklist**.
database.blacklist	*empty string*	An optional comma-separated list of regular expressions that match database names to be excluded from monitoring; any database name not included in the blacklist is monitored. May not be used with **database.whitelist**.
collection.whitelist	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified namespaces for MongoDB collections to be monitored; any collection not included in the whitelist is excluded from monitoring. Each identifier is of the form *databaseName*.*collectionName*. By default the connector will monitor all collections except those in the **local** and **admin** databases. May not be used with **collection.blacklist**.
collection.blacklist	*empty string*	An optional comma-separated list of regular expressions that match fully-qualified namespaces for MongoDB collections to be excluded from monitoring; any collection not included in the blacklist is monitored. Each identifier is of the form *databaseName*.*collectionName*. May not be used with **collection.whitelist**.
snapshot.mode	**initial**	Specifies the criteria for running a snapshot upon startup of the connector. The default is **initial**, and specifies the connector reads a snapshot when either no offset is found or if the oplog no longer contains the previous offset. The **never** option specifies that the connector should never use snapshots, instead the connector should proceed to tail the log.

Property	Default	Description
field.blacklist	*empty string*	An optional comma-separated list of the fully-qualified names of fields that should be excluded from change event message values. Fully-qualified names for fields are of the form *databaseName*.*collectionName*.*fieldName*.*nestedFieldName*, where *databaseName* and *collectionName* may contain the wildcard (*) which matches any characters.
field.renames	*empty string*	An optional comma-separated list of the fully-qualified replacements of fields that should be used to rename fields in change event message values. Fully-qualified replacements for fields are of the form *databaseName*.*collectionName*.*fieldName*.*nestedFieldName*:*newNestedFieldName*, where *databaseName* and *collectionName* may contain the wildcard (*) which matches any characters, the colon character (:) is used to determine rename mapping of field. The next field replacement is applied to the result of the previous field replacement in the list, so keep this in mind when renaming multiple fields that are in the same path.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The MongoDB connector will attempt to use a separate task for each replica set, so the default is acceptable when using the connector with a single MongoDB replica set. When using the connector with a MongoDB sharded cluster, we recommend specifying a value that is equal to or more than the number of shards in the cluster, so that the work for each replica set can be distributed by Kafka Connect.
initial.sync.max.threads	**1**	Positive integer value that specifies the maximum number of threads used to perform an intial sync of the collections in a replica set. Defaults to 1.

Property	Default	Description
tombstones.on.delete	**true**	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.
snapshot.delay.ms		An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size	**0**	Specifies the maximum number of documents that should be read in one go from each collection while taking a snapshot. The connector will read the collection contents in multiple batches of this size. Defaults to 0, which indicates that the server chooses an appropriate fetch size.

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

Property	Default	Description
max.queue.size	**8192**	Positive integer value that specifies the maximum size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the oplog reader when, for example, writes to Kafka are slower or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified in the **max.batch.size** property.
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.

poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.
connect.backoff.initial.delay.ms	**1000**	Positive integer value that specifies the initial delay when trying to reconnect to a primary after the first failed connection attempt or when no primary is available. Defaults to 1 second (1000 ms).
connect.backoff.max.delay.ms	**1000**	Positive integer value that specifies the maximum delay when trying to reconnect to a primary after repeated failed connection attempts or when no primary is available. Defaults to 120 seconds (120,000 ms).
connect.max.attempts	**16**	Positive integer value that specifies the maximum number of failed connection attempts to a replica set primary before an exception occurs and task is aborted. Defaults to 16, which with the defaults for **connect.backoff.initial.delay.ms** and **connect.backoff.max.delay.ms** results in just over 20 minutes of attempts before failing.
mongodb.members.auto.discover	**true**	Boolean value that specifies whether the addresses in 'mongodb.hosts' are seeds that should be used to discover all members of the cluster or replica set (**true**), or whether the address(es) in **mongodb.hosts** should be used as is (**false**). The default is **true** and should be used in all cases except where MongoDB is fronted by a proxy.

heartbeat.interval.ms	**0**	Controls how frequently heartbeat messages are sent. This property contains an interval in milli-seconds 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.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names are sanitized to adhere to Avro naming requirements.
skipped.operations		comma-separated list of oplog operations that will be skipped during streaming. The operations include: **i** for inserts, **u** for updates, and **d** for deletes. By default, no operations are skipped.
provide.transaction.metadata	**false**	When set to **true** Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.

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.

Reconnection attempt number	Delay before attempt, in seconds	Total delay before attempt, in minutes and seconds
1	1	00:01
2	2	00:03
3	4	00:07

4	8	00:15
5	16	00:31
6	32	01:03
7	64	02:07
8	120	04:07
9	120	06:07
10	120	08:07
11	120	10:07
12	120	12:07
13	120	14:07
14	120	16:07
15	120	18:07
16	120	20:07

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.

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

Debezium's SQL Server Connector can monitor and record the row-level changes in the schemas of a SQL Server database.

The first time it connects to a SQL Server database/cluster, it reads a consistent snapshot of all of the schemas. When that snapshot is complete, the connector continuously streams the changes that were committed to SQL Server and generates corresponding insert, update and delete events. All of the events for each table are recorded in a separate Kafka topic, where they can be easily consumed by applications and services.

5.1. OVERVIEW

The functionality of the connector is based upon change data capture feature provided by SQL Server Standard (since SQL Server 2016 SP1) or Enterprise edition. Using this mechanism a SQL Server capture process monitors all databases and tables the user is interested in and stores the changes into specifically created *CDC* tables that have stored procedure facade.

The database operator must enable *CDC* for the table(s) that should be captured by the connector. The connector then produces a *change event* for every row-level insert, update, and delete operation that was published via the *CDC API*, recording all the change events for each table in a separate Kafka topic. The client applications read the Kafka topics that correspond to the database tables they're interested in following, and react to every row-level event it sees in those topics.

The database operator normally enables *CDC* in the mid-life of a database an/or table. This means that the connector does not have the complete history of all changes that have been made to the database. Therefore, when the SQL Server connector first connects to a particular SQL Server 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, we start with a consistent view of all of the data, yet continue reading without having lost any of the changes made while the snapshot was taking place.

The connector is also tolerant of failures. As the connector reads changes and produces events, it records the position in the database log (*LSN / Log Sequence Number*), that is associated with *CDC* record, with each event. If the connector stops for any reason (including communication failures, network problems, or crashes), upon restart it simply continues reading the *CDC* tables where it last left off. This includes snapshots: if the snapshot was not completed when the connector is stopped, upon restart it begins a new snapshot.

5.2. SETTING UP SQL SERVER

Before using the SQL Server connector to monitor the changes committed on SQL Server, first enable *CDC* on a monitored database. Please bear in mind that *CDC* cannot be enabled for the **master** database.

```
-- ====
-- Enable Database for CDC template
-- ====
USE MyDB
GO
EXEC sys.sp_cdc_enable_db
GO
```

Then enable *CDC* for each table that you plan to monitor.

_

```
-- ====
-- Enable a Table Specifying Filegroup Option Template
-- ====
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
```

Verify that the user have access to the *CDC* table.

```
-- ====
-- Verify the user of the connector have access, this query should not have empty result
-- ====

EXEC sys.sp_cdc_help_change_data_capture
GO
```

If the result is empty then please make sure that the user has privileges to access both the capture instance and *CDC* tables.

5.2.1. SQL Server on Azure

The SQL Server plug-in has not been tested with SQL Server on Azure. We welcome any feedback from a user to try the plug-in with database in managed environments.

5.3. HOW THE SQL SERVER CONNECTOR WORKS

5.3.1. Snapshots

SQL Server CDC is not designed to store the complete history of database changes. It is thus necessary that Debezium establishes the baseline of current database content and streams it to the Kafka. This is achieved via a process called snapshotting.

By default (snapshotting mode **initial**) the connector will upon the first startup perform an initial *consistent snapshot* of the database (meaning the structure and data within any tables to be captured as per the connector's filter configuration).

Each snapshot consists of the following steps:

1. Determine the tables to be captured
2. Obtain a lock on each of the monitored tables to ensure that no structural changes can occur to any of the tables. The level of the lock is determined by **snapshot.isolation.mode** configuration option.
3. Read the maximum LSN ("log sequence number") position in the server's transaction log.
4. Capture the structure of all relevant tables.

5. Optionally release the locks obtained in step 2, i.e. the locks are held usually only for a short period of time.
6. Scan all of the relevant database tables and schemas as valid at the LSN position read in step 3, and generate a **READ** event for each row and write that event to the appropriate table-specific Kafka topic.
7. Record the successful completion of the snapshot in the connector offsets.

5.3.2. Reading the change data tables

Upon first start-up, the connector takes a structural snapshot of the structure of the captured tables and persists this information in its internal database history topic. Then the connector identifies a change table for each of the source tables and executes the main loop

1. For each change table read all changes that were created between last stored maximum LSN and current maximum LSN
2. Order the read changes incrementally according to commit LSN and change LSN. This ensures that the changes are replayed by Debezium in the same order as were made to the database.
3. Pass commit and change LSNs as offsets to Kafka Connect.
4. Store the maximum LSN and repeat the loop.

After a restart, the connector will resume from the offset (commit and change LSNs) where it left off before.

The connector is able to detect whether CDC is enabled or disabled for whitelisted source tables and adjust its behavior.

5.3.3. Topic names

The SQL Server connector writes events for all insert, update, and delete operations on a single table to a single Kafka topic. The name of the Kafka topics always takes the form *serverName*.*schemaName*.*tableName*, where *serverName* is the logical name of the connector as specified with the **database.server.name** configuration property, *schemaName* is the name of the schema where the operation occurred, and *tableName* is the name of the database table on which the operation occurred.

For example, consider a SQL Server installation with an **inventory** database that contains four tables: **products**, **products_on_hand**, **customers**, and **orders** in schema **dbo**. If the connector monitoring this database were given a logical server name of **fulfillment**, then the connector would produce events on these four Kafka topics:

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

5.3.4. Events

All data change events produced by the SQL Server connector have a key and a value, although the structure of the key and value depend on the table from which the change events originated (see Topic names).

> **WARNING**
>
> The SQL Server connector ensures that all Kafka Connect *schema names* are valid Avro schema names. This means that the logical server name must start with Latin letters or an underscore (e.g., [a-z,A-Z,_]), and the remaining characters in the logical server name and all characters in the schema and table names must be Latin letters, digits, or an underscore (e.g., [a-z,A-Z,0-9,_]). If not, then all invalid characters will automatically be replaced with an underscore character.
>
> This can lead to unexpected conflicts when the logical server name, schema names, and table names contain other characters, and the only distinguishing characters between table full names are invalid and thus replaced with underscores.

Debezium and Kafka Connect are designed around *continuous streams of event messages*, and the structure of these events may change over time. This could be difficult for consumers to deal with, so to make it easy Kafka Connect makes each event self-contained. Every message key and value has two parts: a *schema* and *payload*. The schema describes the structure of the payload, while the payload contains the actual data.

5.3.4.1. Change Event Keys

For a given table, the change event's key will have a structure that contains a field for each column in the primary key (or unique key constraint) of the table at the time the event was created.

Consider a **customers** table defined in the **inventory** database's schema **dbo**:

```
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
);
```

If the **database.server.name** configuration property has the value **server1**, every change event for the **customers** table while it has this definition will feature the same key structure, which in JSON looks like this:

```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "int32",
                "optional": false,
                "field": "id"
            }
        ],
        "optional": false,
        "name": "server1.dbo.customers.Key"
    },
    "payload": {
        "id": 1004
    }
}
```

The **schema** portion of the key contains a Kafka Connect schema describing what is in the key portion. In this case, it means that the **payload** value is not optional, is a structure defined by a schema named **server1.dbo.customers.Key**, and has one required field named **id** of type **int32**. If you look at the value of the key's **payload** field, you can see that it is indeed a structure (which in JSON is just an object) with a single **id** field, whose value is **1004**.

Therefore, you can interpret this key as describing the row in the **dbo.customers** table (output from the connector named **server1**) whose **id** primary key column had a value of **1004**.

5.3.4.2. Change Event Values

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

- **op** is a mandatory field that contains a string value describing the type of operation. Values for the SQL Server connector are **c** for create (or insert), **u** for update, **d** for delete, and **r** for read (in the case of a snapshot).
- **before** is an optional field that if present contains the state of the row *before* the event occurred. The structure is described by the **server1.dbo.customers.Value** Kafka Connect schema, which the **server1** connector uses for all rows in the **dbo.customers** table.
- **after** is an optional field that if present contains the state of the row *after* the event occurred. The structure is described by the same **server1.dbo.customers.Value** Kafka Connect schema used in **before**.
- **source** is a mandatory field that contains a structure describing the source metadata for the event, which in the case of SQL Server contains these fields: the Debezium version, the connector name, whether the event is part of an ongoing snapshot or not, the commit LSN (not while snapshotting), the LSN of the change, database, schema and table where the change happened, and a timestamp representing the point in time when the record was changed in the source database (during snapshotting, this is the point in time of snapshotting).
  Also a field **event_serial_no** is present during streaming. This is used to differentiate among events that have the same commit and change LSN. There are mostly two situations when you can see it present with value different from **1**:
    - update events will have the value set to **2**, this is because the update generates two events in the CDC change table of SQL Server (source documentation). The first one contains the old values and the second one contains new values. So the first one is dropped and the values from it are used with the second one to create the Debezium change event.
    - when a primary key is updated, then SQL Server emits two records - **delete** to remove the record with the old primary key value and **insert** to create the record with the new primary key. Both operations share the same commit and change LSN and their event numbers are **1** and **2**.

- **ts_ms** is optional and if present contains the time (using the system clock in the JVM running the Kafka Connect task) at which the connector processed the event.

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

5.3.4.2.1. Create events

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

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "server1.dbo.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
```

```
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "server1.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": "1.1.2.Final",
    "connector": "sqlserver",
    "name": "server1",
    "ts_ms": 1559729468470,
    "snapshot": false,
    "db": "testDB",
    "schema": "dbo",
    "table": "customers",
    "change_lsn": "00000027:00000758:0003",
    "commit_lsn": "00000027:00000758:0005",
    "event_serial_no": "1"
    },
    "op": "c",
    "ts_ms": 1559729471739
  }
}
```

If we look at the **schema** portion of this event's *value*, we can see the schema for the *envelope*, the schema for the **source** structure (which is specific to the SQL Server connector and reused across all events), and the table-specific schemas for the **before** and **after** fields.

NOTE

The names of the schemas for the **before** and **after** fields are of the form *logicalName*.*schemaName*.*tableName*.Value, and thus are entirely independent from all other schemas for all other tables. This means that when using the Avro Converter, the resulting Avro schemas for *each table* in each *logical source* have their own evolution and history.

If we look at the **payload** portion of this event's *value*, we can see the information in the event, namely that it is describing that the row was created (since **op=c**), and that the **after** field value contains the values of the new inserted row's' **id**, **first_name**, **last_name**, and **email** columns.

NOTE

It may appear that the JSON representations of the events are much larger than the rows they describe. This is true, because the JSON representation must include the *schema* and the *payload* portions of the message. It is possible and even recommended to use the to dramatically decrease the size of the actual messages written to the Kafka topics.

5.3.4.2.2. Update events

The value of an *update* change event on this table will actually have the exact same *schema*, and its payload is structured the same but will hold different values. Here's an example:

```
{
  "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": "1.1.2.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
  }
}
```

When we compare this to the value in the *insert* event, we see a couple of differences in the **payload** section:

- The **op** field value is now **u**, signifying that this row changed because of an update
- The **before** field now has the state of the row with the values before the database commit
- The **after** field now has the updated state of the row, and here was can see that the **email** value is now **noreply@example.org**.
- The **source** field structure has the same fields as before, but the values are different since this event is from a different position in the transaction log.
- The **event_serial_no** field has value **2**. That is due to the update event composed of two events behind the scenes and we are exposing only the second one. If you are interested in details please check the source documentation and refer to the field **$operation**.
- The **ts_ms** shows the timestamp that Debezium processed this event.

There are several things we can learn by just looking at this **payload** section. We can compare the **before** and **after** structures to determine what actually changed in this row because of the commit. The **source** structure tells us information about SQL Server's record of this change (providing traceability), but more importantly this has information we can compare to other events in this and other topics to know whether this event occurred before, after, or as part of the same SQL Server commit as other events.

NOTE

When the columns for a row's primary/unique key are updated, the value of the row's key has changed so Debezium will output *three* events: a **DELETE** event and a tombstone event with the old key for the row, followed by an **INSERT** event with the new key for the row.

5.3.4.2.3. Delete events

So far, you have seen samples of *create* and *update* events. The following sample shows the value of a *delete* event for the same table. Once again, the **schema** portion of the value is exactly the same as with the *create* and *update* events:

```
{
  "schema": { ... },
  },
    "payload": {
      "before": {
        "id": 1005,
        "first_name": "john",
        "last_name": "doe",
        "email": "noreply@example.org"
      },
      "after": null,
      "source": {
        "version": "1.1.2.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
    }
  }
```

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

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

This event gives a consumer all kinds of information that it can use to process the removal of this row.

The SQL Server connector's events are designed to work with Kafka log compaction, which allows for the removal of some older messages as long as at least the most recent message for every key is kept. This allows Kafka to reclaim storage space while ensuring the topic contains a complete dataset and can be used for reloading key-based state.

When a row is deleted, the *delete* event value listed above still works with log compaction, since Kafka can still remove all earlier messages with that same key. But only if the message value is **null** will Kafka know that it can remove *all messages* with that same key. To make this possible, the SQL Server connector always follows the *delete* event with a special *tombstone* event that has the same key but **null** value.

5.3.5. Transaction Metadata

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

5.3.5.1. 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 emmitted 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:

```
{
  "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**.

5.3.5.2. 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:

```
{
  "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"
  }
}
```

5.3.6. Database schema evolution

Debezium is able to capture schema changes over time. Due to the way CDC is implemented in SQL Server, it is necessary to work in co-operation with a database operator in order to ensure the connector continues to produce data change events when the schema is updated.

As was already mentioned before, Debezium uses SQL Server's change data capture functionality. This means that SQL Server creates a capture table that contains all changes executed on the source table. Unfortunately, the capture table is static and needs to be updated when the source table structure changes. This update is not done by the connector itself but must be executed by an operator with elevated privileges.

There are generally two procedures how to execute the schema change:

- cold - this is executed when Debezium is stopped
- hot - executed while Debezium is running

Both approaches have their own advantages and disadvantages.

WARNING

In both cases, it is critically important to execute the procedure completely before a new schema update on the same source table is made. It is thus recommended to execute all DDLs in a single batch so the procedure is done only once.

NOTE

Not all schema changes are supported when CDC is enabled for a source table. One such exception identified is renaming a column or changing its type, SQL Server will not allow executing the operation.

NOTE

Although not required by SQL Server's CDC mechanism itself, a new capture instance must be created when altering a column from **NULL** to **NOT NULL** or vice versa. This is required so that the SQL Server connector can pick up that changed information. Otherwise, emitted change events will have the **optional** value for the corresponding field (**true** or **false**) set to match the original value.

5.3.6.1. Cold schema update

This is the safest procedure but might not be feasible for applications with high-availability requirements. The operator should follow this sequence of steps

1. Suspend the application that generates the database records
2. Wait for Debezium to stream all unstreamed changes
3. Stop the connector
4. Apply all changes to the source table schema
5. Create a new capture table for the update source table using **sys.sp_cdc_enable_table** procedure with a unique value for parameter **@capture_instance**
6. Resume the application
7. Start the connector
8. When Debezium starts streaming from the new capture table it is possible to drop the old one using **sys.sp_cdc_disable_table** stored procedure with parameter **@capture_instance** set to the old capture instance name

5.3.6.2. Hot schema update

The hot schema update does not require any downtime in application and data processing. The procedure itself is also much simpler than in case of cold schema update

1. Apply all changes to the source table schema
2. Create a new capture table for the update source table using **sys.sp_cdc_enable_table** procedure with a unique value for parameter **@capture_instance**
3. When Debezium starts streaming from the new capture table it is possible to drop the old one using **sys.sp_cdc_disable_table** stored procedure with parameter **@capture_instance** set to the old capture instance name

The hot schema update has one drawback. There is a period of time between the database schema update and creating the new capture instance. All changes that will arrive during this period are captured by the old instance with the old structure. For instance this means that in case of a newly added column any change event produced during this time will not yet contain a field for that new column. If your application does not tolerate such a transition period we recommend to follow the cold schema update.

5.3.6.3. Example

In this example, a column **phone_number** is added to the **customers** table.

```
# Start the database shell
docker-compose -f docker-compose-sqlserver.yaml exec sqlserver bash -c '/opt/mssql-
tools/bin/sqlcmd -U sa -P $SA_PASSWORD -d testDB'
```

```
-- Modify the source table schema
ALTER TABLE customers ADD phone_number VARCHAR(32);

-- Create the new capture instance
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

-- Insert new data
INSERT INTO customers(first_name,last_name,email,phone_number) VALUES
('John','Doe','john.doe@example.com', '+1-555-123456');
GO
```

Kafka Connect log will contain messages like these:

```
connect_1    | 2019-01-17 10:11:14,924 INFO   ||  Multiple capture instances present for the same
table: Capture instance "dbo_customers" [sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_CT, startLsn=00000024:00000d98:0036,
changeTableObjectId=1525580473, stopLsn=00000025:00000ef8:0048] and Capture instance
"dbo_customers_v2" [sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
connect_1    | 2019-01-17 10:11:14,924 INFO   ||  Schema will be changed for ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
...
connect_1    | 2019-01-17 10:11:33,719 INFO   ||  Migrating schema to ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
```

Eventually, there is a new field in the schema and value of the messages written to the Kafka topic.

```
...
    {
      "type": "string",
      "optional": true,
      "field": "phone_number"
    }
...
   "after": {
     "id": 1005,
     "first_name": "John",
     "last_name": "Doe",
        "email": "john.doe@example.com",
        "phone_number": "+1-555-123456"
      },
```

```
-- Drop the old capture instance
EXEC sys.sp_cdc_disable_table @source_schema = 'dbo', @source_name = 'dbo_customers',
@capture_instance = 'dbo_customers';
GO
```

5.3.7. Data types

As described above, the SQL Server connector represents the changes to rows with events that are structured like the table in which the row exist. The event contains a field for each column value, and how that value is represented in the event depends on the SQL data type of the column. This section describes this mapping.

The following table describes how the connector maps each of the SQL Server data types to a *literal type* and *semantic type* within the events' fields. Here, the *literal type* describes how the value is literally represented using Kafka Connect schema types, namely **INT8**, **INT16**, **INT32**, **INT64**, **FLOAT32**, **FLOAT64**, **BOOLEAN**, **STRING**, **BYTES**, **ARRAY**, **MAP**, and **STRUCT**. The *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.

SQL Server Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
BIT	**BOOLEAN**	n/a	
TINYINT	**INT16**	n/a	
SMALLINT	**INT16**	n/a	
INT	**INT32**	n/a	
BIGINT	**INT64**	n/a	
REAL	**FLOAT32**	n/a	
FLOAT[(N)]	**FLOAT64**	n/a	
CHAR[(N)]	**STRING**	n/a	
VARCHAR[(N)]	**STRING**	n/a	
TEXT	**STRING**	n/a	
NCHAR[(N)]	**STRING**	n/a	
NVARCHAR[(N)]	**STRING**	n/a	

NTEXT	**STRING**	n/a	
XML	**STRING**	**io.debezium.data.Xml**	Contains the string representation of a XML document
DATETIMEOFFSET[(P)]	**STRING**	**io.debezium.time.ZonedTimestamp**	A string representation of a timestamp with timezone information, where the timezone is GMT

Other data type mappings are described in the following sections.

If present, a column's default value is propagated to the corresponding field's Kafka Connect schema. Change messages will contain the field's default value (unless an explicit column value had been given), so there should rarely be the need to obtain the default value from the schema.

5.3.7.1. Temporal values

Other than SQL Server's **DATETIMEOFFSET** data type (which contain time zone information), the other temporal types depend on the value of the **time.precision.mode** configuration property. When the **time.precision.mode** configuration property is set to **adaptive** (the default), then the connector will determine the literal type and semantic type for the temporal types based on the column's data type definition so that events *exactly* represent the values in the database:

SQL Server Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
DATE	**INT32**	**io.debezium.time.Date**	Represents the number of days since epoch.
TIME(0), **TIME(1)**, **TIME(2)**, **TIME(3)**	**INT32**	**io.debezium.time.Time**	Represents the number of milliseconds past midnight, and does not include timezone information.
TIME(4), **TIME(5)**, **TIME(6)**	**INT64**	**io.debezium.time.MicroTime**	Represents the number of microseconds past midnight, and does not include timezone information.
TIME(7)	**INT64**	**io.debezium.time.NanoTime**	Represents the number of nanoseconds past midnight, and does not include timezone information.

DATETIME	**INT64**	**io.debezium.time.Timestamp**	Represents the number of milliseconds past epoch, and does not include timezone information.
SMALLDATETIME	**INT64**	**io.debezium.time.Timestamp**	Represents the number of milliseconds past epoch, and does not include timezone information.
DATETIME2(0), **DATETIME2(1)**, **DATETIME2(2)**, **DATETIME2(3)**	**INT64**	**io.debezium.time.Timestamp**	Represents the number of milliseconds past epoch, and does not include timezone information.
DATETIME2(4), **DATETIME2(5)**, **DATETIME2(6)**	**INT64**	**io.debezium.time.MicroTimestamp**	Represents the number of microseconds past epoch, and does not include timezone information.
DATETIME2(7)	**INT64**	**io.debezium.time.NanoTimestamp**	Represents the number of nanoseconds past epoch, and does not include timezone information.

When the **time.precision.mode** configuration property is set to **connect**, then the connector will use the predefined Kafka Connect logical types. This may be useful when consumers only know about the built-in Kafka Connect logical types and are unable to handle variable-precision time values. On the other hand, since SQL Server supports tenth of microsecond precision, the events generated by a connector with the **connect** time precision mode will **result in a loss of precision** when the database column has a *fractional second precision* value greater than 3:

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

SMALLDATETIME	**INT64**	**org.apache.kafka.connect.data.Timestamp**	Represents the number of milliseconds past epoch, and does not include timezone information.
DATETIME2	**INT64**	**org.apache.kafka.connect.data.Timestamp**	Represents the number of milliseconds since epoch, and does not include timezone information. SQL Server allows **P** to be in the range 0-7 to store up to tenth of microsecond precision, though this mode results in a loss of precision when **P** > 3.

5.3.7.1.1. 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.

5.3.7.2. Decimal values

SQL Server Data Type	Literal type (schema type)	Semantic type (schema name)	Notes
NUMERIC[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted. The **connect.decimal.precision** schema parameter contains an integer representing the precision of the given decimal value.
DECIMAL[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted. The **connect.decimal.precision** schema parameter contains an integer representing the precision of the given decimal value.

SMALLMONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted. The **connect.decimal.precision** schema parameter contains an integer representing the precision of the given decimal value.
MONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal**	The **scale** schema parameter contains an integer representing how many digits the decimal point was shifted. The **connect.decimal.precision** schema parameter contains an integer representing the precision of the given decimal value.

5.4. DEPLOYING THE SQL SERVER CONNECTOR

Installing the SQL Server connector is a simple process whereby you only need to download the JAR, extract it to your Kafka Connect environment, and ensure the plug-in's parent directory is specified in your Kafka Connect environment.

Prerequisites

- You have Zookeeper, Kafka, and Kafka Connect installed.
- You have SQL Server installed and setup.

Procedure

1. Download the Debezium SQL Server connector.
2. Extract the files into your Kafka Connect environment.
3. Add the plug-in's parent directory to your Kafka Connect **plugin.path**:

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

NOTE

The above example assumes you have extracted the Debezium SQL Server connector to the /**kafka/connect/debezium-connector-sqlserver** path.

4. Restart your Kafka Connect process. This ensures the new JARs are picked up.

Additional resources

For more information on the deployment process, and deploying connectors with AMQ Streams, refer to the Debezium installation guides.

- Installing Debezium on OpenShift

- Installing Debezium on RHEL

5.4.1. Example configuration

To use the connector to produce change events for a particular SQL Server database or cluster:

1. Enable the CDC on SQL Server to publish the *CDC* events in the database.
2. Create a configuration file for the SQL Server connector.

When the connector starts, it will grab a consistent snapshot of the schemas in your SQL Server database and start streaming changes, producing events for every inserted, updated, and deleted row. You can also choose to produce events for a subset of the schemas and tables. Optionally ignore, mask, or truncate columns that are sensitive, too large, or not needed.

Following is an example of the configuration for a connector instance that monitors a SQL Server server at port 1433 on 192.168.99.100, which we logically name **fullfillment**. Typically, you configure the Debezium SQL Server connector in a **.yaml** file using the configuration properties available for the connector.

```
apiVersion: kafka.strimzi.io/v1beta1
  kind: KafkaConnector
  metadata:
    name: inventory-connector  (1)
    labels: strimzi.io/cluster: my-connect-cluster
  spec:
    class: io.debezium.connector.sqlserver.SqlServerConnector (2)
    config:
      database.hostname: 192.168.99.100  (3)
      database.port: 1433 (4)
      database.user: debezium  (5)
      database.password: dbz  (6)
      database.dbname: testDB  (7)
      database.server.name: fullfullment (8)
      database.whitelist: dbo.customers   (9)
      database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 (10)
      database.history.kafka.topic: dbhistory.fullfillment  (11)
```

1. The name of our connector when we register it with a Kafka Connect service.
2. The name of this SQL Server connector class.
3. The address of the SQL Server instance.
4. The port number of the SQL Server instance.
5. The name of the SQL Server user
6. The password for the SQL Server user
7. The name of the database to capture changes from.
8. The logical name of the SQL Server instance/cluster, which forms a namespace and is used in all the names of the Kafka topics to which the connector writes, the Kafka Connect schema names, and the namespaces of the corresponding Avro schema when the Avro Connector is used.

9 A list of all tables whose changes Debezium should capture.

10 The list of Kafka brokers that this connector will use to write and recover DDL statements to the database history topic.

11 The name of the database history topic where the connector will write and recover DDL statements. This topic is for internal use only and should not be used by consumers.

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 SQL Server database, read the transaction log, and record events to Kafka topics.

5.4.2. Monitoring

The Debezium SQL Server connector has three 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 reading CDC table data
- schema history metrics; for monitoring the status of the connector's schema history

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

5.4.2.1. Snapshot Metrics

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

Attribute Name	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.

MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.

5.4.2.2. Streaming Metrics

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

Attribute Name	Type	Description

LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapcity	**int**	The length of the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapcity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**map<string, string>**	The coordinates of the last received event.

LastTransactionId	**string**	Transaction identifier of the last processed transaction.

5.4.2.3. Schema History Metrics

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

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

5.4.3. Connector properties

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

Property	Default	Description

name		Unique name for the connector. Attempting to register again with the same name will fail. (This property is required by all Kafka Connect connectors.)
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.sqlserver.SqlServerConnector** for the SQL Server connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The SQL Server connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname		IP address or hostname of the SQL Server database server.
database.port	**1433**	Integer port number of the SQL Server database server.
database.user		Username to use when connecting to the SQL Server database server.
database.password		Password to use when connecting to the SQL Server database server.
database.dbname		The name of the SQL Server database from which to stream the changes
database.server.name		Logical name that identifies and provides a namespace for the particular SQL Server database server being monitored. The logical name should be unique across all other connectors, since it is used as a prefix for all Kafka topic names emanating from this connector. Only alphanumeric characters and underscores should be used.
database.history.kafka.topic		The full name of the Kafka topic where the connector will store the database schema history.

database.history .kafka.bootstrap.servers		A list of host/port pairs that the connector will use 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. This should point to the same Kafka cluster used by the Kafka Connect process.
table.whitelist		An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be monitored; any table not included in the whitelist is excluded from monitoring. Each identifier is of the form *schemaName.tableName*. By default the connector will monitor every non-system table in each monitored schema. May not be used with **table.blacklist**.
table.blacklist		An optional comma-separated list of regular expressions that match fully-qualified table identifiers for tables to be excluded from monitoring; any table not included in the blacklist is monitored. Each identifier is of the form *schemaName.tableName*. May not be used with **table.whitelist**.
column.blacklist	*empty string*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns that should be excluded from change event message values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Note that primary key columns are always included in the event's key, also if blacklisted from the value.
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.

tombstones.on.delete	**true**	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.
column.truncate.to.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event message values if the field values are longer than the specified number of characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.
column.mask.with.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*****) characters. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.
column.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** is used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*.

datatype.propagate.source.type	*n/a*	An optional comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change messages. The schema parameters **__debezium.source.column.type**, **__debezium.source.column.length** and **__debezium.source.column.scale** will be used to propagate the original type name and length (for variable-width types), respectively. Useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of the form *schemaName.tableName.typeName*. See SQL Server data types for the list of SQL Server-specific data type names.
message.key.columns	*empty string*	A semi-colon list of regular expressions that match fully-qualified tables and columns to map a primary key. Each item (regular expression) must match the fully-qualified **<fully-qualified table>:<a comma-separated list of columns>** representing the custom key. Fully-qualified tables could be defined as *schemaName.tableName*.

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

Property	Default	Description
snapshot.mode	*initial*	A mode for taking an initial snapshot of the structure and optionally data of captured tables. Once the snapshot is complete, the connector will continue reading change events from the database's redo logs. Supported values are: **initial**: Takes a snapshot of structure and data of captured tables; useful if topics should be populated with a complete representation of the data from the captured tables. **schema_only**: Takes a snapshot of the structure of captured tables only; useful if only changes happening from now onwards should be propagated to topics.

snapshot.isolation.mode	*repeatable_read*	Mode to control which transaction isolation level is used and how long the connector locks the monitored tables. There are five possible values: **read_uncommitted**, **read_committed**, **repeatable_read**, **snapshot**, and **exclusive** (in fact, **exclusive** mode uses repeatable read isolation level, however, it takes the exclusive lock on all tables to be read). It is worth documenting that **snapshot**, **read_committed** and **read_uncommitted** modes do not prevent other transactions from updating table rows during initial snapshot, while **exclusive** and **repeatable_read** do. Another aspect is data consistency. Only **exclusive** and **snapshot** modes guarantee full consistency, that is, initial snapshot and streaming logs constitute a linear history. In case of **repeatable_read** and **read_committed** modes, it might happen that, for instance, a record added appears twice - once in initial snapshot and once in streaming phase. Nonetheless, that consistency level should do for data mirroring. For **read_uncommitted** there are no data consistency guarantees at all (some data might be lost or corrupted).
event.processing .failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events. **fail** will propagate the exception (indicating the offset of the problematic event), causing the connector to stop. **warn** will cause the problematic event to be skipped and the offset of the problematic event to be logged. **skip** will cause the problematic event to be skipped.
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait during each iteration for new change events to appear. Defaults to 1000 milliseconds, or 1 second.

max.queue.size	**8192**	Positive integer value that specifies the maximum size of the blocking queue into which change events read from the database log are placed before they are written to Kafka. This queue can provide backpressure to the CDC table reader when, for example, writes to Kafka are slower or if Kafka is not available. Events that appear in the queue are not included in the offsets periodically recorded by this connector. Defaults to 8192, and should always be larger than the maximum batch size specified in the **max.batch.size** property.
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.
heartbeat.interval.ms	**0**	Controls how frequently heartbeat messages are sent. This property contains an interval in milli-seconds 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 tables are changed for a longer period of time. In such situation the connector would proceed to read the log from the database but never emit any change messages into Kafka, which in turn means that no offset updates are committed to Kafka. This may result in more change events to be re-sent after a connector restart. Set this parameter to **0** to not send heartbeat messages at all. Disabled by default.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the naming of the topic to which heartbeat messages are sent. The topic is named according to the pattern **<heartbeat.topics.prefix>.<server.name>**.
snapshot.delay.ms		An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.

snapshot.fetch.size	**2000**	Specifies the maximum number of rows that should be read in one go from each table while taking a snapshot. The connector will read the table contents in multiple batches of this size. Defaults to 2000.
snapshot.lock.timeout.ms	**10000**	An integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If table locks cannot be acquired in this time interval, the snapshot will fail (also see snapshots). When set to **0** the connector will fail immediately when it cannot obtain the lock. Value **-1** indicates infinite waiting.
snapshot.select.statement.overrides		Controls which rows from tables are included in snapshot. This property contains a comma-separated list of fully-qualified tables *(SCHEMA_NAME.TABLE_NAME)*. Select statements for the individual tables are specified in further configuration properties, one for each table, identified by the id **snapshot.select.statement.overrides.[SCHEMA_NAME].[TABLE_NAME]**. The value of those properties is the SELECT statement to use when retrieving data from the specific table during snapshotting. *A possible use case for large append-only tables is setting a specific point where to start (resume) snapshotting, in case a previous snapshotting was interrupted.* **Note**: This setting has impact on snapshots only. Events captured during log reading are not affected by it.
sanitize.field.names	**true** when connector configuration explicitly specifies the **key.converter** or **value.converter** parameters to use Avro, otherwise defaults to **false**.	Whether field names are sanitized to adhere to Avro naming requirements.

database.server.timezone		Timezone of the server. This is used to define the timezone of the transaction timestamp (ts_ms) retrieved from the server (which is actually not zoned). Default value is unset. Should only be specified when running on SQL Server 2014 or older and using different timezones for the database server and the JVM running the Debezium connector. When unset, default behavior is to use the timezone of the VM running the Debezium connector. In this case, when running on on SQL Server 2014 or older and using different timezones on server and the connector, incorrect ts_ms values may be produced. Possible values include "Z", "UTC", offset values like "+02:00", short zone ids like "CET", and long zone ids like "Europe/Paris".
provide.transaction.metadata	**false**	When set to **true** Debezium generates events with transaction boundaries and enriches data events envelope with transaction metadata. See Transaction Metadata for additional details.

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.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. (The SQL Server connector does use the new consumer.)

CHAPTER 6. 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.

6.1. MONITORING DEBEZIUM ON RHEL

6.1.1. Zookeeper JMX environment variables

Zookeeper has built-in support for JMX. When running Zookeeper using a local installation, the **zkServer.sh** script recognizes the following environment variables:

JMXPORT

Enables JMX and specifies the port number that will be used for JMX. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.port=$JMXPORT**.

JMXAUTH

Whether JMX clients must use password authentication when connecting. Must be either **true** or **false**. The default is **false**. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.authenticate=$JMXAUTH**.

JMXSSL

Whether JMX clients connect using SSL/TLS. Must be either **true** or **false**. The default is **false**. The value is used to specify the JVM parameter **-Dcom.sun.management.jmxremote.ssl=$JMXSSL**.

JMXLOG4J

Whether the Log4J JMX MBeans should be disabled. Must be either **true** (default) or **false**. The default is **true**. The value is used to specify the JVM parameter **-Dzookeeper.jmx.log4j.disable=$JMXLOG4J**.

6.1.2. 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**

6.1.3. 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**

6.2. 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 addition, you can use Prometheus and Grafana to monitor the JMX metrics. For more information, see Introducing Metrics.

CHAPTER 7. 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.

7.1. 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 will also have 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 will be 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.

7.2. UNDERSTANDING THE DEFAULT LOGGING CONFIGURATION

If you are running Debezium connectors in a Kafka Connect process, then Kafka Connect will use the Log4j configuration file (for example, **/opt/kafka/config/connect-log4j.properties**) in the Kafka installation. By default, this file contains the following configuration:

connect-log4j.properties

```
log4j.rootLogger=INFO, stdout (1)

log4j.appender.stdout=org.apache.log4j.ConsoleAppender (2)
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout (3)
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n (4)
...
```

(1) The root logger, which defines the default logger configuration. By default, loggers will include **INFO**, **WARN**, and **ERROR** messages. These log messages will be written to the **stdout** appender.

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

(3) The **stdout** appender will use a pattern matching algorithm to format the log messages.

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

Unless you configure other loggers, all of the loggers used by Debezium will inherit the **rootLogger** configuration.

7.3. CONFIGURING 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:

- Change the logging level
- Add mapped diagnostic contexts

NOTE

This section only covers a couple methods you can use to configure Debezium logging with Log4j. For more information about using Log4j, search for tutorials to set up and use appenders to send log messages to specific destinations.

7.3.1. Changing the logging level

The default Debezium logging level provides sufficient information to show whether a connector is healthy or not. However, if a connector is not healthy, you can change its logging level to troubleshoot the issue.

In general, Debezium connectors send their log messages to loggers with names that match the fully-qualified name of the Java class that is generating the log message. Debezium uses packages to organize code with similar or related functions. This means that you can control all of the log messages for a specific class or for all of the classes within or under a specific package.

Procedure

1. Open the **log4j.properties** file.
2. Configure a logger for the connector.
   This example configures loggers for the MySQL connector and the database history implementation used by the connector, and sets them to log **DEBUG** level messages:

   log4j.properties

```
...
```

```
log4j.logger.io.debezium.connector.mysql=DEBUG, stdout 1
log4j.logger.io.debezium.relational.history=DEBUG, stdout 2

log4j.additivity.io.debezium.connector.mysql=false 3
log4j.additivity.io.debezium.relational.history=false 4
...
```

1. Configures the logger named **io.debezium.connector.mysql** to send **DEBUG**, **INFO**, **WARN**, and **ERROR** messages to the **stdout** appender.

2. Configures the logger named **io.debezium.relational.history** to send **DEBUG**, **INFO**, **WARN**, and **ERROR** messages to the **stdout** appender.

3 4 Turns off *additivity*, which means that the log messages will not be sent to appenders of parent loggers (this can prevent seeing duplicate log messages when using multiple appenders).

3. If necessary, change the logging level for a specific subset of the classes within the connector. Increasing the logging level for the entire connector increases the log verbosity, which can make it difficult to understand what is happening. In these cases, you can change the logging level just for the subset of classes that are related to the issue that you are troubleshooting.

    a. Set the connector's logging level to either **DEBUG** or **TRACE**.

    b. Review the connector's log messages.
    Find the log messages that are related to the issue that you are troubleshooting. The end of each log message shows the name of the Java class that produced the message.

    c. Set the connector's logging level back to **INFO**.

    d. Configure a logger for each Java class that you identified.
    For example, consider a scenario in which you are unsure why the MySQL connector is skipping some events when it is processing the binlog. Rather than turn on **DEBUG** or **TRACE** logging for the entire connector, you can keep the connector's logging level at **INFO** and then configure **DEBUG** or **TRACE** on just the class that is reading the binlog:

    log4j.properties

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

    log4j.additivity.io.debezium.connector.mysql=false
    log4j.additivity.io.debezium.relational.history=false
    log4j.additivity.io.debezium.connector.mysql.BinlogReader=false
    ...
    ```

7.3.2. Adding mapped diagnostic contexts

Most Debezium connectors (and the Kafka Connect workers) use multiple threads to perform different activities. This can make it difficult to look at a log file and find only those log messages for a particular logical activity. To make the log messages easier to find, Debezium provides several *mapped diagnostic contexts* (MDC) that provide additional information for each thread.

Debezium provides the following MDC properties:

dbz.connectorType
: A short alias for the type of connector. For example, **MySql**, **Mongo**, **Postgres**, and so on. All threads associated with the same *type* of connector use the same value, so you can use this to find all log messages produced by a given type of connector.

dbz.connectorName
: The name of the connector or database server as defined in the connector's configuration. For example **products**, **serverA**, and so on. All threads associated with a specific *connector instance* use the same value, so you can find all of the log messages produced by a specific connector instance.

dbz.connectorContext
: A short name for an activity running as a separate thread running within the connector's task. For example, **main**, **binlog**, **snapshot**, and so on. In some cases, when a connector assigns threads to specific resources (such as a table or collection), the name of that resource could be used instead. Each thread associated with a connector would use a distinct value, so you can find all of the log messages associated with this particular activity.

To enable MDC for a connector, you configure an appender in the **log4j.properties** file.

Procedure

1. Open the **log4j.properties** file.

2. Configure an appender to use any of the supported Debezium MDC properties.
   In this 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
   ...
   ```

   This will produce log messages similar to these:

   ```
   ...
   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**).

7.4. DEBEZIUM LOGGING ON OPENSHIFT

If you are using Debezium on OpenShift, you can use the Kafka Connect loggers to configure the Debezium loggers and logging levels. For more information, see Kafka Connect loggers.

CHAPTER 8. 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.

- The **ByLogicalTableRouter** SMT re-routes data change event records to topics that you specify.
- The **ExtractNewRecordState** SMT flattens the complex structure of a data change event record into the simplified format that might be required by some Kafka consumers.
- Configuring Avro serialization for PostgreSQL, MongoDB, or SQL Server connectors makes it easier for change event record consumers to adapt to a changing record schema.
- The **CloudEventsConverter** enables a Debezium connector to emit change event records that conform to the CloudEvents specification.

8.1. ROUTING CHANGE 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 **ByLogicalTableRouter** 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 **ByLogicalTableRouter** transformation is a Kafka Connect SMT.

The following topics provide details:

- Section 8.1.1, "Use case for routing records to topics that you specify"
- Section 8.1.2, "Example of routing records for multiple tables to one topic"
- Section 8.1.3, "Ensuring unique keys across records routed to the same topic"
- Section 8.1.4, "Options for configuring **ByLogicalTableRouter** transformation"

8.1.1. Use case for routing records to topics that you specify

The default behavior is that a Debezium connector sends each change event record to a topic whose name is formed from the name of the database and the name of the table in which the change was made. In other words, a topic receives records for one physical table. When you want a topic to receive records for more than one physical table, you must configure the Debezium connector to re-route the records to that topic.

Logical tables

A logical table is a common use case for routing records for multiple physical tables to one topic. In a logical table, there are multiple physical tables that all have the same schema. For example, sharded tables have the same schema. A logical table might consist of two or more sharded tables: **db_shard1.my_table** and **db_shard2.my_table**. The tables are in different shards and are physically distinct but together they form a logical table. You can re-route change event records for tables in any of the shards to the same topic.

Partitioned PostgreSQL tables

When the Debezium PostgreSQL connector captures changes in a partitioned table, the default behavior is that change event records are routed to a different topic for each partition. To emit records from all partitions to one topic, configure the **ByLogicalTableRouter** 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.

8.1.2. Example of routing records for multiple tables to one topic

To route change event records for multiple physical tables to the same topic, configure the **ByLogicalTableRouter** transformation in the Kafka Connect configuration for the Debezium connector. Configuration of the **ByLogicalTableRouter** SMT requires you to specify regular expressions that determine:

- The tables for which to route records. These tables must all have the same schema.
- The destination topic name.

For example, configuration in a **.properties** file looks like this:

```
transforms=Reroute
transforms.Reroute.type=io.debezium.transforms.ByLogicalTableRouter
transforms.Reroute.topic.regex=(.*)customers_shard(.*)
transforms.Reroute.topic.replacement=$1customers_all_shards
```

topic.regex

Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
In the example, the regular expression, **(.)customers_shard(.)** matches records for changes to tables whose names include the **customers_shard** string. This would re-route records for tables with the following names:

myserver.mydb.customers_shard1
myserver.mydb.customers_shard2
myserver.mydb.customers_shard3

topic.replacement

Specifies a regular expression that represents the destination topic name. The transformation routes each matching record to the topic identified by this expression. In this example, records for the three sharded tables listed above would be routed to the **myserver.mydb.customers_all_shards** topic.

8.1.3. Ensuring unique keys across 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 **ByLogicalTableRouter** 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 **ByLogicalTableRouter** transformation to insert a different field into the key. To do this, specify the **key.field.name** option and set it to a field name that does not clash with existing primary key field names. For example:

```
transforms=Reroute
transforms.Reroute.type=io.debezium.transforms.ByLogicalTableRouter
transforms.Reroute.topic.regex=(.*)customers_shard(.*)
transforms.Reroute.topic.replacement=$1customers_all_shards
transforms.Reroute.key.field.name=shard_id
```

This example adds the **shard_id** field to the key structure in routed records.

If you want to adjust the value of the key's new field, configure both of these options:

key.field.regex
: Specifies a regular expression that the transformation applies to the default destination topic name to capture one or more groups of characters.

key.field.replacement
: Specifies a regular expression for determining the value of the inserted key field in terms of those captured groups.

For example:

```
transforms.Reroute.key.field.regex=(.*)customers_shard(.*)
transforms.Reroute.key.field.replacement=$2
```

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

myserver.mydb.customers_shard1
myserver.mydb.customers_shard2
myserver.mydb.customers_shard3

The transformation uses the values in the second captured group, the shard numbers, as the value of the key's new field. In this example, the inserted key field's values would be **1**, **2**, or **3**.

If your tables contain globally unique keys and you do not need to change the key structure, you can set the **key.enforce.uniqueness** property to **false**:

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

8.1.4. Options for configuring ByLogicalTableRouter transformation

Property	Default	Description
topic.regex		Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
topic.replacement		Specifies a regular expression that represents the destination topic name. The transformation routes each matching record to the topic identified by this expression. This expression can refer to groups captured by the regular expression that you specify for **topic.regex**. To refer to a group, specify **$1**, **$2**, and so on.
key.enforce.uniqueness	**true**	Indicates whether to add a field to the record's change event key. Adding a key field ensures that each event key is unique across the tables whose change event records go to the same topic. This helps to prevent collisions of change events for records that have the same key but that originate from different source tables. Specify **false** if you do not want the transformation to add a key field. For example, if you are routing records from a partitioned PostgreSQL table to one topic, you can configure **key.enforce.uniqueness=false** because unique keys are guaranteed in partitioned PostgreSQL tables.
key.field.name	**__dbz__physicalTableIdentifier**	Name of a field to be added to the change event key. The value of this field identifies the original table name. For the SMT to add this field, **key.enforce.uniqueness** must be **true**, which is the default.
key.field.regex		Specifies a regular expression that the transformation applies to the default destination topic name to capture one or more groups of characters. For the SMT to apply this expression, **key.enforce.uniqueness** must be **true**, which is the default.
key.field.replacement		Specifies a regular expression for determining the value of the inserted key field in terms of the groups captured by the expression specified for **key.field.regex**. For the SMT to apply this expression, **key.enforce.uniqueness** must be**true**, which is the default.

8.2. 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 **ExtractNewRecordState** 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 **ExtractNewRecordState** transformation is a Kafka Connect SMT.

The transformation is available to only SQL database connectors.

The following topics provide details:

- Section 8.2.1, “Description of Debezium change event structure”
- Section 8.2.2, “Behavior of Debezium **ExtractNewRecordState** transformation”
- Section 8.2.3, “Configuration of **ExtractNewRecordState** transformation”
- Section 8.2.4, “Example of adding metadata to the Kafka record”
- Section 8.2.5, “Options for configuring **ExtractNewRecordState** transformation”

8.2.1. 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, the structure of an **UPDATE** change event looks like this:

```
{
 "op": "u",
 "source": {
  ...
 },
 "ts_ms" : "...",
 "before" : {
  "field1" : "oldvalue1",
   "field2" : "oldvalue2"
  },
  "after" : {
   "field1" : "newvalue1",
   "field2" : "newvalue2"
  }
}
```

This complex format provides the most information about changes happening in the system. However, other connectors or other parts of the Kafka ecosystem usually expect the data in a simple format like this:

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

To provide the needed Kafka record format for consumers, configure the **ExtractNewRecordState** SMT.

8.2.2. Behavior of Debezium ExtractNewRecordState transformation

The **ExtractNewRecordState** 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 **ExtractNewRecordState** SMT for a Debezium connector or for a sink connector that consumes messages emitted by a Debezium connector. The advantage of configuring **ExtractNewRecordState** 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 **ExtractNewRecordData** SMT to do one of the following:

- Keep the record in the stream and edit it to have only the **"value": "null"** field.
- Keep the record in the stream and edit it to have a **value** field that contains the key/value pairs that were in the **before** field with an added **"__deleted": "true"** entry.

Similary, instead of dropping the tombstone record, you can configure the **ExtractNewRecordData** SMT to keep the tombstone record in the stream.

8.2.3. Configuration of ExtractNewRecordState transformation

Configure the Debezium **ExtractNewRecordState** SMT in a Kafka Connect source or sink connector by adding the SMT configuration details to your connector's configuration. To obtain the default behavior, in a **.properties** file, you would specify something like the following:

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
```

As for any Kafka Connect connector configuration, you can set **transforms=** to multiple, comma-separated, SMT aliases in the order in which you want Kafka Connect to apply the SMTs.

The following **.properties** example sets several **ExtractNewRecordState** 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.

8.2.4. Example of adding metadata to the Kafka record

The **ExtractNewRecordState** SMT can add original, change event metadata to the simplified Kafka record. For example, you might want the simplified record's header or value to contain any of the following:

- The type of operation that made the change
- The name of the database or table that was changed
- Connector-specific fields such as the Postgres LSN field

To add metadata to the simplified Kafka record's header, specify the **add.header** option. To add metadata to the simplified Kafka record's value, specify the **add.fields** option. Each of these options takes a comma separated list of change event field names. Do not specify spaces. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field. For example:

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
transforms.unwrap.add.fields=op,table,lsn,source.ts_ms
transforms.unwrap.add.headers=db
transforms.unwrap.delete.handling.mode=rewrite
```

With that configuration, a simplified Kafka record would contain something like the following:

```
{
 ...
 "__op" : "c",
 "__table": "MY_TABLE",
 "__lsn": "123456789",
 "__source_ts_ms" : "123456789",
 ...
}
```

Also, simplified Kafka records would have a **__db** header.

In the simplified Kafka record, the SMT prefixes the metadata field names with a double underscore. When you specify a struct, the SMT also inserts an underscore between the struct name and the field name.

To add metadata to a simplified Kafka record that is for a **DELETE** operation, you must also configure **delete.handling.mode=rewrite**.

8.2.5. Options for configuring ExtractNewRecordState transformation

The following table describes the options that you can specify for the **ExtractNewRecordState** SMT.

Property	Default	Description
drop.tombstones	**true**	Debezium generates a tombstone record for each **DELETE** operation. The default behavior is that **ExtractNewRecordState** removes tombstone records from the stream. To keep tombstone records in the stream, specify **drop.tombstones=false**.

delete.handling.mode	**drop**	Debezium generates a change event record for each **DELETE** operation. The default behavior is that **ExtractNewRecordState** 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.
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 **ExtractNewRecordState** SMT on a sink connector, setting this option might be useful when the destination topic name dictates the name of the database table that will be updated with the simplified change event record. If the topic name is not correct for your use case, you can configure **route.by.field** to re-route the event.

add.fields		Set this option to a comma-separated list, with no spaces, of metadata fields to add to the simplified Kafka record's value. When there are duplicate field names, to add metadata for one of those fields, specify the struct as well as the field, for example **source.ts_ms**. When the SMT adds metadata fields to the simplified record's value, it prefixes each metadata field name with a double underscore. For a struct specification, the SMT also inserts an underscore between the struct name and the field name. If you specify a field that is not in the change event record, the SMT still adds the field to the record's value.
add.headers		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**. 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.

8.3. AVRO SERIALIZATION

IMPORTANT

Using Avro to serialize record keys and values 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.

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 does the following:

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 change event record consumers 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 .

8.3.1. About the Red Hat Integration - Service Registry

Red Hat Integration - Service Registry provides several 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

  Since 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.

8.3.2. Deployment overview

To deploy a Debezium connector that uses Avro serialization, there are 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:

```
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.AutoRegisterIdStrategy
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.AutoRegisterIdStrategy
```

Internally, Kafka Connect always uses JSON key/value converters for storing configuration and offsets.

8.3.3. Deploying with Debezium containers

In your environment, you might want to use a provided Debezium container to deploy Debezium connectors that use Avro serializaion. Follow the procedure here to do that. In this procedure, you build a custom Kafka Connect container image for Debezium, and you configure the Debezium connector to use the Avro converter.

Prerequisites

- You have cluster administrator access to an OpenShift cluster.

- 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 archive(s) to create a directory structure for the connector plug-in(s). If you downloaded and extracted the archive for each Debezium connector, the structure looks like this:

```
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. While this duplicates the files, it removes 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-25:1.5.0** 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-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 debezium-container-with-avro:latest

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

[illegible]

- 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
  kind: KafkaConnect
  metadata:
    name: my-connect-cluster
  spec:
    #...
    image: debezium-container-with-avro
  ```

- In the **install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml** file, edit the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable to point to the new container image and reinstall the Cluster Operator. If you edit this file you will need to apply it to your OpenShift cluster.

5. Deploy each Debezium connector that is configured to use the Avro converter. For each Debezium connector:

   a. Create a Debezium connector instance. The following **inventory-connector.yaml** file example creates a **KafkaConnector** custom resource that defines a MySQL connector instance that is configured to use the Avro converter:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnector
   metadata:
     name: inventory-connector
     labels:
       strimzi.io/cluster: my-connect-cluster
   spec:
     class: io.debezium.connector.mysql.MySqlConnector
     tasksMax: 1
     config:
       database.hostname: mysql
       database.port: 3306
       database.user: debezium
       database.password: dbz
       database.server.id: 184054
       database.server.name: dbserver1
       database.whitelist: 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.AutoRegisterIdStrategy
       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.AutoRegisterIdStrategy
   ```

   b. Apply the connector instance, for example:
   oc apply -f inventory-connector.yaml

[illegible]

This registers **inventory-connector** and the connector starts to run against the **inventory** database.

6. Verify that the connector was created and has started to track changes in the specified database. You can verify the connector instance by watching the Kafka Connect log output as, for example, **inventory-connector** starts.

   a. Display the Kafka Connect log output:

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

   b. Review the log output to verify that the initial snapshot has been executed. You should see something like the following lines:

   ```
   ...
   2020-02-21 17:57:30,801 INFO Starting snapshot for jdbc:mysql://mysql:3306/?
   useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnicode=
   true&characterEncoding=UTF-8&characterSetResults=UTF-
   8&zeroDateTimeBehavior=CONVERT_TO_NULL&connectTimeout=30000 with user
   'debezium' with locking mode 'minimal' (io.debezium.connector.mysql.SnapshotReader)
   [debezium-mysqlconnector-dbserver1-snapshot]
   2020-02-21 17:57:30,805 INFO Snapshot is using user 'debezium' with these MySQL
   grants: (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-
   dbserver1-snapshot]
   ...
   ```

   Taking the snapshot involves a number of steps:

   ```
   ...
   2020-02-21 17:57:30,822 INFO Step 0: disabling autocommit, enabling repeatable read
   transactions, and setting lock wait timeout to 10
   (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   2020-02-21 17:57:30,836 INFO Step 1: flush and obtain global read lock to prevent
   writes to database (io.debezium.connector.mysql.SnapshotReader) [debezium-
   mysqlconnector-dbserver1-snapshot]
   2020-02-21 17:57:30,839 INFO Step 2: start transaction with consistent snapshot
   (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   2020-02-21 17:57:30,840 INFO Step 3: read binlog position of MySQL master
   (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   2020-02-21 17:57:30,843 INFO   using binlog 'mysql-bin.000003' at position '154' and gtid
   '' (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   ...
   2020-02-21 17:57:34,423 INFO Step 9: committing transaction
   (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   2020-02-21 17:57:34,424 INFO Completed snapshot in 00:00:03.632
   (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
   snapshot]
   ...
   ```

After completing the snapshot, Debezium begins tracking changes in, for example, the **inventory** database's **binlog** for change events:

```
...
2020-02-21 17:57:35,584 INFO Transitioning from the snapshot reader to the binlog
reader (io.debezium.connector.mysql.ChainedReader) [task-thread-inventory-connector-
0]
2020-02-21 17:57:35,613 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [task-thread-inventory-connector-0]
2020-02-21 17:57:35,630 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [blc-mysql:3306]
Feb 21, 2020 5:57:35 PM com.github.shyiko.mysql.binlog.BinaryLogClient connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:5)
2020-02-21 17:57:35,775 INFO Connected to MySQL binlog at mysql:3306, starting at
binlog file 'mysql-bin.000003', pos=154, skipping 0 events plus 0 rows
(io.debezium.connector.mysql.BinlogReader) [blc-mysql:3306]
...
```

8.3.4. Naming

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.

8.4. EXPORTING CLOUDEVENTS

CloudEvents is a specification for describing event data in a common way. Its aim is to provide interoperability across services, platforms and systems. Debezium enables you to configure a MongoDB, MySQL, PostgreSQL, or SQL Server connector to emit change event records that conform to the CloudEvents specification.

IMPORTANT

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

The CloudEvents specification defines:

- A set of standardized event attributes

- Rules for defining custom attributes
- Encoding rules for mapping event formats to serialized representations such as JSON
- 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 must be JSON and the **data** format must be JSON. It is expected that a future Debezium release will support binary mapping mode.

8.4.1. Example event format

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

```
{
  "id" : "name:test_server;lsn:29274832;txId:565",   (1)
  "source" : "/debezium/postgresql/test_server",   (2)
  "specversion" : "1.0",   (3)
  "type" : "io.debezium.postgresql.datachangeevent",   (4)
  "time" : "2020-01-13T13:55:39.738Z",   (5)
  "datacontenttype" : "application/json",   (6)
  "iodebeziumop" : "r",   (7)
  "iodebeziumversion" : "1.1.2.Final",   (8)
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578923739738",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "565",
  "iodebeziumlsn" : "29274832",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "565",   (9)
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : {   (10)
    "before" : null,
    "after" : {
      "pk" : 1,
      "name" : "Bob"
    }
  }
}
```

(1) 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.

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.

8.4.2. Example configuration

Configure **io.debezium.converters.CloudEventsConverter** in your Debezium connector configuration. Following is an example of configuring **CloudEventsConverter**. In this example, you could omit the specification of **serializer.type** because **json** is the default.

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

CloudEventsConverter 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**, or **JsonConverter**.

8.4.3. Configuration properties

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

Property	Default	Description
serializer.type	**json**	The encoding type to use for the CloudEvents envelope structure. **json** is the only supported value.
data.serializer.type	**json**	The encoding type to use for the **data** attribute. **json** is the only supported value.

json. ...	N/A	Any configuration properties to be passed through to the underlying converter when using JSON. The **json.** prefix is removed.