Red Hat Integration 2020-Q3

Debezium User Guide

For use with Debezium 1.2

Last Updated: 2020-10-20

Red Hat Integration 2020-Q3 Debezium User Guide

For use with Debezium 1.2

Legal Notice

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 6, *Debezium connector for Db2*
- Chapter 8, *Debezium logging*
- Chapter 7, *Monitoring Debezium*
- Chapter 9, *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, “Description of 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** (for example, 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, for example, for sensitive data
- **Monitoring:** most connectors can be monitored using JMX
- Different ready-to-use **message transformations**: for example, for message routing, extraction of new record state (relational connectors, MongoDB) and routing of events from a transactional outbox table

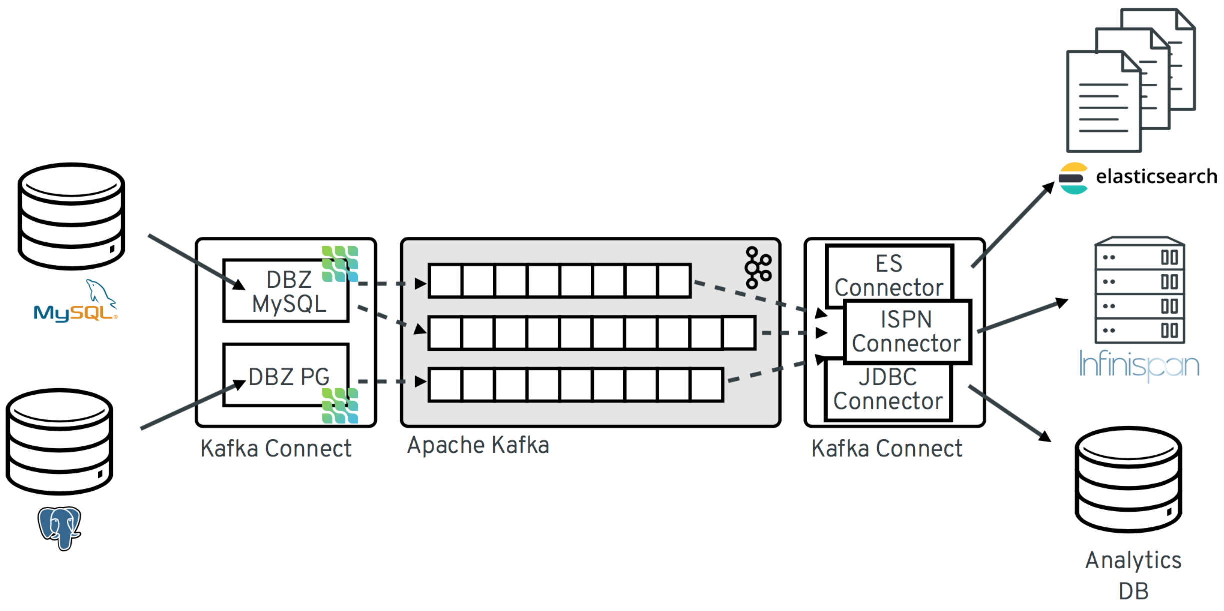
Refer to the connector documentation for a list of all supported databases and detailed information about the features and configuration options of each connector.

1.2. DESCRIPTION OF DEBEZIUM ARCHITECTURE

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

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

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



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

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

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

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

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

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

CHAPTER 2. 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 exposes schema changes.

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 exposes schema changes

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_ms"
        },
        {
          "type": "string",
          "optional": true,
          "field": "gtid"
        },
        {
          "type": "string",
          "optional": false,
          "field": "file"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "pos"
        },
        {
          "type": "int32",
          "optional": false,
          "field": "row"
        },
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "int64",
          "optional": true,
          "field": "thread"
        },
        {
          "type": "string",
          "optional": true,
          "field": "db"
        },
        {
          "type": "string",
          "optional": true,
          "field": "table"
        },
        {
          "type": "string",
          "optional": true,
          "field": "query"
        }
      ]
    }
  ]
 },
 "payload": {
   "databaseName": "inventory",
   "ddl": "CREATE TABLE products ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255) NOT NULL, description VARCHAR(512), weight FLOAT ); ALTER TABLE
products AUTO_INCREMENT = 101;",
   "source" : {
     "version": "1.2.4.Final",
     "name": "mysql-server-1",
     "server_id": 0,
     "ts_ms": 0,
     "gtid": null,
     "file": "mysql-bin.000003",
     "pos": 154,
     "row": 0,
     "snapshot": true,
     "thread": null,
     "db": null,
     "table": null,
     "query": null
   }
 }
}
```

2.1.3.2. Important tips about the schema change topic

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

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

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

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

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

Table 2.1. Overview of change event basic content

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

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

WARNING

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

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

2.1.4.1. Change event keys

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

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

Example table

```
CREATE TABLE customers (
  id INTEGER 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

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

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

Table 2.2. Description of change event key

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

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

2.1.4.2. Change event values

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

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

```
CREATE TABLE customers (
  id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE KEY
) AUTO_INCREMENT=1001;
```

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

- *create* events
- *update* events

- *delete* events

2.1.4.2.1. *create* events

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

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "mysql-server-1.inventory.customers.Value",
        "field": "after"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          },
          {
            "type": "string",
            "optional": true,
            "field": "gtid"
          },
          {
            "type": "string",
            "optional": false,
            "field": "file"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "pos"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "row"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "query"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source", (3)
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mysql-server-1.inventory.customers.Envelope" (4)
  },
  "payload": { (5)
    "op": "c", (6)
    "ts_ms": 1465491411815, (7)
    "before": null, (8)
    "after": { (9)
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 10
      "version": "1.2.4.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
    }
  }
}
```

Table 2.3. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **mysql-server-1.inventory.customers.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.mysql.Source** is the schema for the payload's **source** field. This schema is specific to the MySQL connector. The connector uses it for all events that it generates.
4	**name**	**mysql-server-1.inventory.customers.Envelope** is the schema for the overall structure of the payload, where **mysql-server-1** is the connector name, **inventory** is the database, and **customers** is the table.

Item	Field name	Description
5	**payload**	The value’s actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: ● **c** = create ● **u** = update ● **d** = delete ● **r** = read (applies to only snapshots)
7	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
8	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content.
9	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row’s **id**, **first_name**, **last_name**, and **email** columns.

Item	Field name	Description
10	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: ● Debezium version ● Connector name ● binlog name where the event was recorded ● binlog position ● Row within the event ● If the event was part of a snapshot ● Name of the database and table that contain the new row ● ID of the MySQL thread that created the event (non-snapshot only) ● MySQL server ID (if available) ● Timestamp for when the change was made in the database If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration **include.query** property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event.

2.1.4.2.2. *update* events

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

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { 2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 3
```

```
        "version": "1.2.4.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
    }
  }
```

Table 2.4. Descriptions of *update* event value fields

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

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different, for example, the sample *update* event is from a different position in the binlog. The source metadata includes: ● Debezium version ● Connector name ● binlog name where the event was recorded ● binlog position ● Row within the event ● If the event was part of a snapshot ● Name of the database and table that contain the updated row ● ID of the MySQL thread that created the event (non-snapshot only) ● MySQL server ID (if available) ● Timestamp for when the change was made in the database If the **binlog_rows_query_log_events** MySQL configuration option is enabled and the connector configuration **include.query** property is enabled, the **source** field also provides the **query** field, which contains the original SQL statement that caused the change event.
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

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

2.1.4.2.3. Primary key updates

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

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

2.1.4.2.4. *delete* events

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

```
{
  "schema": { ... },
  "payload": {
    "before": { (1)
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, (2)
    "source": { (3)
      "version": "1.2.4.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)
  }
}
```

Table 2.5. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit.

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

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

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

Tombstone events

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

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*
BIT(>1)	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits. The **byte[]** contains the bits in *little-endian* form and is sized to contain the specified number of bits. For example, where *n* is bits: `numBytes = n/8 + (n%8== 0 ? 0 : 1)`
TINYINT	**INT16**	*n/a*
SMALLINT[(M)]	**INT16**	*n/a*
MEDIUMINT[(M)]	**INT32**	*n/a*
INT, INTEGER[(M)]	**INT32**	*n/a*

MySQL type	Literal type	Semantic type
BIGINT[(M)]	**INT64**	*n/a*
REAL[(M,D)]	**FLOAT32**	*n/a*
FLOAT[(M,D)]	**FLOAT64**	*n/a*
DOUBLE[(M,D)]	**FLOAT64**	*n/a*
CHAR(M)]	**STRING**	*n/a*
VARCHAR(M)]	**STRING**	*n/a*
BINARY(M)]	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
VARBINARY(M)]	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
TINYBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
TINYTEXT	**STRING**	*n/a*
BLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
TEXT	**STRING**	*n/a*
MEDIUMBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
MEDIUMTEXT	**STRING**	*n/a*

MySQL type	Literal type	Semantic type
LONGBLOB	**BYTES** or **STRING**	*n/a* Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the binary handling mode setting
LONGTEXT	**STRING**	*n/a*
JSON	**STRING**	**io.debezium.data.Json** Contains the string representation of a **JSON** document, array, or scalar.
ENUM	**STRING**	**io.debezium.data.Enum** The **allowed** schema parameter contains the comma-separated list of allowed values.
SET	**STRING**	**io.debezium.data.EnumSet** The **allowed** schema parameter contains the comma-separated list of allowed values.
YEAR[(2|4)]	**INT32**	**io.debezium.time.Year**
TIMESTAMP[(M)]	**STRING**	**io.debezium.time.ZonedTimestamp** In ISO 8601 format with microsecond precision. MySQL allows **M** to be in the range of**0-6**.

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** Represents the number of days since the epoch.
TIME[(M)]	**INT64**	**io.debezium.time.MicroTime** Represents the time value in microseconds and does not include time zone information. MySQL allows **M** to be in the range of **0-6**.
DATETIME, DATETIME(0), DATETIME(1), DATETIME(2), DATETIME(3)	**INT64**	**io.debezium.time.Timestamp** Represents the number of milliseconds since the epoch and does not include time zone information.
DATETIME(4), DATETIME(5), DATETIME(6)	**INT64**	**io.debezium.time.MicroTimestamp** Represents the number of microseconds since the 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** Represents the number of days since the epoch.
TIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Time** Represents the time value in microseconds since midnight and does not include time zone information.
DATETIME[(M)]	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since epoch, and does not include time zone information.

2.1.5.2. Decimal values

Decimals are handled via the **decimal.handling.mode** property. 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** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.
DECIMAL[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer that represents how many digits the decimal point shifted.

decimal.handling.mode=double

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*

2.1.5.3. 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
```

2.1.5.4. Spatial data types

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

MySQL type	Literal type	Semantic type
GEOMETRY, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	**STRUCT**	**io.debezium.data.geometry.Geometry** Contains a structure with two fields: ● **srid (INT32**: 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:

serverName.databaseName.tableName

For example, suppose that **fulfillment** is the server name and **inventory** is the database that contains three tables: **orders**, **customers**, and **products**. The Debezium MySQL connector emits events to three Kafka topics, one for each table in the database:

```
fulfillment.inventory.orders
fulfillment.inventory.customers
fulfillment.inventory.products
```

2.1.7. 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';
```

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;
```

Table 2.6. 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	When performing a snapshot, enables the connector to use the **FLUSH** statement to clear or reload internal caches, flush tables, or acquire locks.
SHOW DATABASES	When performing a snapshot, enables the connector to see database names by issuing the **SHOW DATABASE** statement.

Permission/item	Description
REPLICATION SLAVE	Enables the connector to connect to and read the MySQL server binlog.
REPLICATION CLIENT	Enables the connector to run the following commands: • **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 binlog config properties:

```
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:

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

Table 2.7. 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. 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    |
+--------------------------+-------+
```

Table 2.8. Options explained

Permission/item	Description
gtid_mode	Boolean that specifies whether GTID mode of the MySQL server is enabled or not. • **ON** = enabled • **OFF** = disabled
enforce_gtid_consistency	Boolean that instructs the server whether 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>
```

Table 2.9. Options explained

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

- Enable **binlog_rows_query_log_events**:

```
mysql> binlog_rows_query_log_events=ON
```

Additional information

binlog_rows_query_log_events is set to a Boolean value that 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

- Kafka and Kafka Connect are installed.
- MySQL Server is installed and set up to run the Debezium MySQL connector.

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
   ```

   This 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)
```

Table 2.10. Descriptions of connector configuration settings

Item	Description
1	The name of the connector.
2	Only one task should operate at any one time. Because the MySQL connector reads the MySQL server's **binlog**, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.
3	The connector's configuration.
4	The database host, which is the name of the container running the MySQL server (**mysql**).
5	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.
6	Only changes in the **inventory** database will be detected.
7	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. Adding MySQL connector configuration to Kafka Connect

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

Prerequisites

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

Procedure

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

   ```
   tree ./my-plugins/
   ./my-plugins/
   ├── debezium-connector-mysql
   │   ├── ...
   ```

2. Create and publish a custom image for running your Debezium connector:

   a. Create a new **Dockerfile** by using **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** 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-rhel7:1.5.0
   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 container image. For example, if you saved the **Dockerfile** that you created in the previous step as **debezium-container-for-mysql**, and if the **Dockerfile** is in the current directory, then you would run the following command:
   podman build -t debezium-container-for-mysql:latest .

   c. Push your custom image to your container registry, for example:
   podman push debezium-container-for-mysql:latest

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

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

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

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

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

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

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

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

   a. Display the Kafka Connect log output:

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

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

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

Results

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

2.3.4. 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.

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.)

Property	Default	Description
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.
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.

Property	Default	Description
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**.
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*.

Property	Default	Description
column.mask.with .*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns 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*.
column.mask .hash.*hashAlgorithm* .with.salt.*salt*	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in the change event message values with a field value consisting of the hashed value using the algorithm ***hashAlgorithm*** and salt ***salt***. Based on the used hash function referential integrity is kept while data is pseudonymized. Supported hash functions are described in the {link-java7-standard-names}[MessageDigest section] of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash is automatically shortened to the length of the column. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form *databaseName.tableName.columnName*. Example: `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` where **CzQMA0cB5K** is a randomly selected salt. Note: Depending on the ***hashAlgorithm*** used, the ***salt*** selected and the actual data set, the resulting masked data set may not be completely anonymized.

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*.
binary.handling.mode	bytes	Specifies how binary (**blob**, **binary**, **varbinary**, etc.) columns should be represented in change events, including: **bytes** represents binary data as byte array (default), **base64** represents binary data as base64-encoded String, **hex** represents binary data as hex-encoded (base16) String

2.3.4.1. 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.

Property	Default	Description
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.
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.

Property	Default	Description
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.
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.

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 **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.
skipped.operations		comma-separated list of oplog operations that will be skipped during streaming. The operations include: **c** for inserts, **u** for updates, and **d** for deletes. By default, no operations are skipped.

2.3.4.2. 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
```

2.3.4.3. 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.

2.3.5. 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.

2.3.5.1. Snapshot metrics

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

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

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

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

Attribute	Type	Description
HoldingGlobalLock	**boolean**	Whether the connector currently holds a global or table write lock.

2.3.5.2. 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.

Attributes	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.

Attributes	Type	Description
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

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

Attribute	Type	Description
BinlogFilename	**string**	The name of the binlog filename that the connector has most recently read.
BinlogPosition	**long**	The most recent position (in bytes) within the binlog that the connector has read.
IsGtidModeEnabled	**boolean**	Flag that denotes whether the connector is currently tracking GTIDs from MySQL server.
GtidSet	**string**	The string representation of the most recent GTID set seen by the connector when reading the binlog.
NumberOfSkippedEvents	**long**	The number of events that have been skipped by the MySQL connector. Typically events are skipped due to a malformed or unparseable event from MySQL's binlog.
NumberOfDisconnects	**long**	The number of disconnects by the MySQL connector.
NumberOfRolledBackTransactions	**long**	The number of processed transactions that were rolled back and not streamed.

Attribute	Type	Description
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.

2.3.5.3. Schema history metrics

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

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

Attributes	Type	Description
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 captures row-level changes in the schemas of a PostgreSQL database. PostgreSQL versions 10, 11, and 12 are supported.

The first time it connects to a PostgreSQL server or cluster, the connector takes a consistent snapshot of all schemas. After that snapshot is complete, the connector continuously captures row-level changes that insert, update, and delete database content and that were committed to a PostgreSQL database. The connector generates data change event records and streams them to Kafka topics. For each table, the default behavior is that the connector streams all generated events to a separate Kafka topic for that table. Applications and services consume data change event records from that topic.

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

3.1. OVERVIEW OF DEBEZIUM POSTGRESQL CONNECTOR

PostgreSQL's *logical decoding* feature was introduced in version 9.4. It is a mechanism that allows the extraction of the changes that were committed to the transaction log and the processing of these changes in a user-friendly manner with the help of an *output plug-in*. The output plug-in enables clients to consume the changes.

The PostgreSQL connector contains two main parts that work together to read and process database changes:

- **pgoutput** is the standard logical decoding output plug-in in PostgreSQL 10+. This is the only supported logical decoding output plug-in in this Debezium release. This plug-in is maintained by the PostgreSQL community, and used by PostgreSQL itself for logical replication. This plug-in is always present so no additional libraries need to be installed. The Debezium connector interprets the raw replication event stream directly into change events.

- Java code (the actual Kafka Connect connector) that reads the changes produced by the logical decoding output plug-in by using PostgreSQL's *streaming replication protocol* and the PostgreSQL *JDBC driver*.

The connector produces a *change event* for every row-level insert, update, and delete operation that was captured and sends change event records for each table in a separate Kafka topic. Client applications read the Kafka topics that correspond to the database tables of interest, and can react to every row-level event they receive from those topics.

PostgreSQL normally purges write-ahead log (WAL) segments after some period of time. This means that the connector does not have the complete history of all changes that have been made to the database. Therefore, when the PostgreSQL connector first connects to a particular PostgreSQL database, it starts by performing a *consistent snapshot* of each of the database schemas. After the connector completes the snapshot, it continues streaming changes from the exact point at which the snapshot was made. This way, the connector starts with a consistent view of all of the data, and does not omit any changes that were made while the snapshot was being taken.

The connector is tolerant of failures. As the connector reads changes and produces events, it records the WAL position for each event. If the connector stops for any reason (including communication failures, network problems, or crashes), upon restart the connector continues reading the WAL where it last left off. This includes snapshots. If the connector stops during a snapshot, the connector begins a new snapshot when it restarts.

IMPORTANT

The connector relies on and reflects the PostgreSQL logical decoding feature, which has the following limitations:

- Logical decoding does not support DDL changes. This means that the connector is unable to report DDL change events back to consumers.
- Logical decoding replication slots are supported on only **primary** servers. When there is a cluster of PostgreSQL servers, the connector can run on only the active **primary** server. It cannot run on **hot** or **warm** standby replicas. If the **primary** server fails or is demoted, the connector stops. After the **primary** server has recovered, you can restart the connector. If a different PostgreSQL server has been promoted to **primary**, adjust the connector configuration before restarting the connector.

Behavior when things go wrong describes what the connector does when there is a problem.

IMPORTANT

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

3.2. HOW DEBEZIUM POSTGRESQL CONNECTORS WORK

To optimally configure and run a Debezium PostgreSQL connector, it is helpful to understand how the connector performs snapshots, streams change events, determines Kafka topic names, and uses metadata.

Details are in the following topics:

- Section 3.2.1, “How Debezium PostgreSQL connectors perform database snapshots”
- Section 3.2.2, “How Debezium PostgreSQL connectors stream change event records”
- Section 3.2.3, “Default names of Kafka topics that receive Debezium PostgreSQL change event records”
- Section 3.2.4, “Metadata in Debezium PostgreSQL change event records”
- Section 3.2.5, “Debezium PostgreSQL connector-generated events that represent transaction boundaries”

3.2.1. How Debezium PostgreSQL connectors perform database snapshots

Most PostgreSQL servers are configured to not retain the complete history of the database in the WAL segments. This means that the PostgreSQL connector would be unable to see the entire history of the database by reading only the WAL. Consequently, the first time that the connector starts, it performs an initial *consistent snapshot* of the database. The default behavior for performing a snapshot consists of the following steps. You can change this behavior by setting the **snapshot.mode** connector configuration property to a value other than **initial**.

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

2. Obtain an **ACCESS SHARE MODE** lock on each of the tables being tracked to ensure that no structural changes can occur to any of the tables while the snapshot is taking place. These locks do not prevent table **INSERT**, **UPDATE** and **DELETE** operations from taking place during the snapshot.
   *This step is omitted when **snapshot.mode** is set to **exported**, which allows the connector to perform a lock-free snapshot.*

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

4. Scan the database tables and schemas, generate a **READ** event for each row and write that event to the appropriate table-specific Kafka topic.

5. Commit the transaction.

6. Record the successful completion of the snapshot in the connector offsets.

If the connector fails, is rebalanced, or stops after Step 1 begins but before Step 6 completes, upon restart the connector begins a new snapshot. After the connector completes its initial snapshot, the PostgreSQL connector continues streaming from the position that it read in step 3. This ensures that the connector does not miss any updates. If the connector stops again for any reason, upon restart, the connector continues streaming changes from where it previously left off.

WARNING

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

Table 3.1. Settings for **snapshot.mode** connector configuration property

Setting	Description

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

3.2.2. How Debezium PostgreSQL connectors stream change event records

The PostgreSQL connector typically spends the vast majority of its time streaming changes from the PostgreSQL server to which it is connected. This mechanism relies on *PostgreSQL's replication protocol*. This protocol enables clients to receive changes from the server as they are committed in the server's transaction log at certain positions, which are referred to as Log Sequence Numbers (LSNs).

Whenever the server commits a transaction, a separate server process invokes a callback function from the logical decoding plug-in. This function processes the changes from the transaction, converts them to a specific format (Protobuf or JSON in the case of Debezium plug-in) and writes them on an output stream, which can then be consumed by clients.

The Debezium PostgreSQL connector acts as a PostgreSQL client. When the connector receives changes it transforms the events into Debezium *create*, *update*, or *delete* events that include the LSN of the event. The PostgreSQL connector forwards these change events in records to the Kafka Connect framework, which is running in the same process. The Kafka Connect process asynchronously writes the change event records in the same order in which they were generated to the appropriate Kafka topic.

Periodically, Kafka Connect records the most recent *offset* in another Kafka topic. The offset indicates source-specific position information that Debezium includes with each event. For the PostgreSQL connector, the LSN recorded in each change event is the offset.

When Kafka Connect gracefully shuts down, it stops the connectors, flushes all event records to Kafka, and records the last offset received from each connector. When Kafka Connect restarts, it reads the last recorded offset for each connector, and starts each connector at its last recorded offset. When the connector restarts, it sends a request to the PostgreSQL server to send the events starting just after that position.

NOTE

The PostgreSQL connector retrieves schema information as part of the events sent by the logical decoding plug-in. However, the connector does not retrieve information about which columns compose the primary key. The connector obtains this information from the JDBC metadata (side channel). If the primary key definition of a table changes (by adding, removing or renaming primary key columns), there is a tiny period of time when the primary key information from JDBC is not synchronized with the change event that the logical decoding plug-in generates. During this tiny period, a message could be created with an inconsistent key structure. To prevent this inconsistency, update primary key structures as follows:

1. Put the database or an application into a read-only mode.
2. Let Debezium process all remaining events.
3. Stop Debezium.
4. Update the primary key definition in the relevant table.
5. Put the database or the application into read/write mode.
6. Restart Debezium.

PostgreSQL 10+ logical decoding support (**pgoutput**)

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

See Setting up PostgreSQL for more details.

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

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

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

For example, suppose that **fulfillment** is the logical server name in the configuration for a connector that is capturing changes in a PostgreSQL installation that has a **postgres** database and an **inventory** schema that contains four tables: **products**, **products_on_hand**, **customers**, and **orders**. The connector would stream records to these four Kafka topics:

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

Now suppose that the tables are not part of a specific schema but were created in the default **public** PostgreSQL schema. The names of the Kafka topics would be:

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

3.2.4. Metadata in Debezium PostgreSQL change event records

In addition to a *database change event*, each record produced by a PostgreSQL connector contains some metadata. Metadata includes where the event occurred on the server, the name of the source partition and the name of the Kafka topic and partition where the event should go, for example:

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

- **sourcePartition** always defaults to the setting of the **database.server.name** connector configuration property.
- **sourceOffset** contains information about the location of the server where the event occurred:
  - **lsn** represents the PostgreSQL Log Sequence Number or **offset** in the transaction log.
  - **txId** represents the identifier of the server transaction that caused the event.
  - **ts_ms** represents the server time at which the transaction was committed in the form of the number of milliseconds since the epoch.
- **kafkaPartition** with a setting of **null** means that the connector does not use a specific Kafka partition. The PostgreSQL connector uses only one Kafka Connect partition and it places the generated events into one Kafka partition.

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

Debezium can generate events that represent transaction boundaries and that enrich data change event messages. For every transaction **BEGIN** and **END**, Debezium generates an event that contains the following fields:

- **status** - **BEGIN** or **END**
- **id** - string representation of unique transaction identifier
- **event_count** (for **END** events) - total number of events emitted by the transaction
- **data_collections** (for **END** events) - an array of pairs of **data_collection** and **event_count** that provides the number of events emitted by changes originating from given data collection

Example

```
{
  "status": "BEGIN",
  "id": "571",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "571",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "s1.a",
      "event_count": 1
    },
    {
      "data_collection": "s2.a",
      "event_count": 1
    }
  ]
}
```

Transaction events are written to the topic named ***database.server.name*****.transaction**.

Change data event enrichment

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

- **id** - string representation of unique transaction identifier
- **total_order** - absolute position of the event among all events generated by the transaction
- **data_collection_order** - the per-data collection position of the event among all events that were emitted by the transaction

Following is an example of a message:

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

3.3. DESCRIPTIONS OF DEBEZIUM POSTGRESQL CONNECTOR DATA CHANGE EVENTS

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

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

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

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

Table 3.2. Overview of change event basic content

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

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

NOTE

Starting with Kafka 0.10, Kafka can optionally record the event key and value with the *timestamp* at which the message was created (recorded by the producer) or written to the log by Kafka.

WARNING

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

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

Details are in the following topics:

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

3.3.1. About keys in Debezium PostgreSQL change events

For a given table, the change event’s key has a structure that contains a field for each column in the primary key of the table at the time the event was created. Alternatively, if the table has **REPLICA IDENTITY** set to **FULL** or **USING INDEX** there is a field for each unique key constraint.

Consider a **customers** table defined in the **public** database schema and the example of a change event key for that table.

Example table

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

Example change event key

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

```
{
  "schema": { ❶
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key", ❷
    "optional": false, ❸
    "fields": [ ❹
          {
              "name": "id",
              "index": "0",
              "schema": {
                  "type": "INT32",
                  "optional": "false"
              }
          }
      ]
  },
  "payload": { ❺
      "id": "1"
  },
}
```

Table 3.3. Description of change event key

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

NOTE

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

WARNING

If the table does not have a primary or unique key, then the change event's key is null. The rows in a table without a primary or unique key constraint cannot be uniquely identified.

3.3.2. About values in Debezium PostgreSQL change events

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

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

```
CREATE TABLE customers (
  id SERIAL,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  PRIMARY KEY(id)
);
```

The value portion of a change event for a change to this table varies according to the **REPLICA IDENTITY** setting and the operation that the event is for.

Details follow in these sections:

- Replica identity
- *create* events
- *update* events
- Primary key updates
- *delete* events
- Tombstone events

Replica identity

REPLICA IDENTITY is a PostgreSQL-specific table-level setting that determines the amount of information that is available to the logical decoding plug-in for **UPDATE** and **DELETE** events. More specifically, the setting of **REPLICA IDENTITY** controls what (if any) information is available for the previous values of the table columns involved, whenever an **UPDATE** or **DELETE** event occurs.

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

- **DEFAULT** - The default behavior is that **UPDATE** and **DELETE** events contain the previous values for the primary key columns of a table if that table has a primary key. For an **UPDATE** event, only the primary key columns with changed values are present.
  If a table does not have a primary key, the connector does not emit **UPDATE** or **DELETE** events for that table. For a table without a primary key, the connector emits only *create* events. Typically, a table without a primary key is used for appending messages to the end of the table, which means that **UPDATE** and **DELETE** events are not useful.

- **NOTHING** - Emitted events for **UPDATE** and **DELETE** operations do not contain any information about the previous value of any table column.

- **FULL** - Emitted events for **UPDATE** and **DELETE** operations contain the previous values of all columns in the table.

- **INDEX** *index-name* - Emitted events for **UPDATE** and **DELETE** operations contain the previous values of the columns contained in the specified index. **UPDATE** events also contain the indexed columns with the updated values.

create events

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

```
{
  "schema": { 1
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "PostgreSQL_server.inventory.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
                "field": "email"
            }
        ],
        "optional": true,
        "name": "PostgreSQL_server.inventory.customers.Value",
        "field": "after"
    },
    {
        "type": "struct",
        "fields": [
            {
                "type": "string",
                "optional": false,
                "field": "version"
            },
            {
                "type": "string",
                "optional": false,
                "field": "connector"
            },
            {
                "type": "string",
                "optional": false,
                "field": "name"
            },
            {
                "type": "int64",
                "optional": false,
                "field": "ts_ms"
            },
            {
                "type": "boolean",
                "optional": true,
                "default": false,
                "field": "snapshot"
            },
            {
                "type": "string",
                "optional": false,
                "field": "db"
            },
            {
                "type": "string",
                "optional": false,
                "field": "schema"
            },
            {
                "type": "string",
                "optional": false,
                "field": "table"
            },
            {
                "type": "int64",
                "optional": true,
                "field": "txId"
            },
          {
            "type": "int64",
            "optional": true,
            "field": "lsn"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "xmin"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.postgresql.Source", 3
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "PostgreSQL_server.inventory.customers.Envelope" 4
  },
  "payload": { 5
    "before": null, 6
    "after": { 7
      "id": 1,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 8
      "version": "1.2.4.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", 9
    "ts_ms": 1559033904863 10
  }
}
```

Table 3.4. Descriptions of *create* event value fields

Item	Field name	Description
1	**schema**	The value's schema, which describes the structure of the value's payload. A change event's value schema is the same in every change event that the connector generates for a particular table.
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **PostgreSQL_server.inventory.customers.Value** is the schema for the payload's **before** and **after** fields. This schema is specific to the **customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.postgresql.Source** is the schema for the payload's **source** field. This schema is specific to the PostgreSQL connector. The connector uses it for all events that it generates.
4	**name**	**PostgreSQL_server.inventory.customers.Envelope** is the schema for the overall structure of the payload, where **PostgreSQL_server** is the connector name, **inventory** is the database, and **customers** is the table.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content. NOTE Whether or not this field is available is dependent on the **REPLICA IDENTITY** setting for each table.
7	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **id**, **first_name**, **last_name**, and **email** columns.

Item	Field name	Description
8	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: • Debezium version • Connector type and name • Database and table that contains the new row • Schema name • If the event was part of a snapshot • ID of the transaction in which the operation was performed • Offset of the operation in the database log • Timestamp for when the change was made in the database
9	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots)
10	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

update events

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

```
{
    "schema": { ... },
    "payload": {
        "before": { (1)
            "id": 1
        },
        "after": { (2)
            "id": 1,
            "first_name": "Anne Marie",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": { (3)
            "version": "1.2.4.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", (4)
        "ts_ms": 1465584025523 (5)
    }
}
```

Table 3.5. Descriptions of *update* event value fields

Item	Field name	Description
1	**before**	An optional field that contains values that were in the row before the database commit. In this example, only the primary key column, **id**, is present because the table's **REPLICA IDENTITY** setting is, by default, **DEFAULT**. + For an *update* event to contain the previous values of all columns in the row, you would have to change the **customers** table by running **ALTER TABLE customers REPLICA IDENTITY FULL**.
2	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **first_name** value is now **Anne Marie**.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different. The source metadata includes: ● Debezium version ● Connector type and name ● Database and table that contains the new row ● Schema name ● If the event was part of a snapshot ● ID of the transaction in which the operation was performed ● Offset of the operation in the database log ● Timestamp for when the change was made in the database
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

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

Primary key updates

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

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

delete events

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

```
{
    "schema": { ... },
    "payload": {
        "before": { 1
            "id": 1
        },
        "after": null, 2
        "source": { 3
            "version": "1.2.4.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", 4
        "ts_ms": 1465581902461 5
    }
}
```

Table 3.6. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit. In this example, the **before** field contains only the primary key column because the table's **REPLICA IDENTITY** setting is **DEFAULT**.
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and **lsn** field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector type and name • Database and table that contains the new row • Schema name • If the event was part of a snapshot • ID of the transaction in which the operation was performed • Offset of the operation in the database log • Timestamp for when the change was made in the database
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

A *delete* change event record provides a consumer with the information it needs to process the removal of this row.

WARNING

For a consumer to be able to process a *delete* event generated for a table that does not have a primary key, set the table's **REPLICA IDENTITY** to **FULL**. When a table does not have a primary key and the table's **REPLICA IDENTITY** is set to **DEFAULT** or **NOTHING**, a *delete* event has no **before** field.

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

Tombstone events

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

3.4. HOW DEBEZIUM POSTGRESQL CONNECTORS MAP DATA TYPES

The PostgreSQL connector represents changes to rows with events that are structured like the table in which the row exists. The event contains a field for each column value. How that value is represented in the event depends on the PostgreSQL data type of the column. This section describes these mappings.

Details are in the following sections:

- Basic types
- Temporal types
- TIMESTAMP type
- Decimal types
- HSTORE type
- Domain types
- Network address types
- PostGIS types
- Toasted values

Basic types

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

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

Table 3.7. Mappings for PostgreSQL basic data types

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
BOOLEAN	**BOOLEAN**	n/a
BIT(1)	**BOOLEAN**	n/a

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
BIT(> 1)	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits. The resulting **byte[]** contains the bits in little-endian form and is sized to contain the specified number of bits. For example, **numBytes = n/8 + (n % 8 == 0 ? 0 : 1)** where **n** is the number of bits.
BIT VARYING[(M)]	**BYTES**	**io.debezium.data.Bits** The **length** schema parameter contains an integer that represents the number of bits (2^31 - 1 in case no length is given for the column). The resulting **byte[]** contains the bits in little-endian form and is sized based on the content. The specified size **(M)** is stored in the length parameter of the **io.debezium.data.Bits** type.
SMALLINT, **SMALLSERIAL**	**INT16**	n/a
INTEGER, **SERIAL**	**INT32**	n/a
BIGINT, **BIGSERIAL**	**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.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
TIMETZ, **TIME WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTime** A string representation of a time value with timezone information, where the timezone is GMT.
INTERVAL [P]	**INT64**	**io.debezium.time.MicroDuration** (default) The approximate number of microseconds for a time interval using the **365.25 / 12.0** formula for days per month average.
INTERVAL [P]	**STRING**	**io.debezium.time.Interval** (when **interval.handling.mode** is set to **string**) The string representation of the interval value that follows the pattern **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**, for example, **P1Y2M3DT4H5M6.78S**.
BYTEA	**BYTES** or **STRING**	n/a Either the raw bytes (the default), a base64-encoded string, or a hex-encoded string, based on the connector's binary handling mode setting.
JSON, **JSONB**	**STRING**	**io.debezium.data.Json** Contains the string representation of a JSON document, array, or scalar.
XML	**STRING**	**io.debezium.data.Xml** Contains the string representation of an XML document.
UUID	**STRING**	**io.debezium.data.Uuid** Contains the string representation of a PostgreSQL UUID value.
POINT	**STRUCT**	**io.debezium.data.geometry.Point** Contains a structure with two **FLOAT64** fields, **(x,y)**. Each field represents the coordinates of a geometric point.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
LTREE	**STRING**	**io.debezium.data.Ltree** Contains the string representation of a PostgreSQL LTREE value.
CITEXT	**STRING**	n/a
INET	**STRING**	n/a
INT4RANGE	**STRING**	n/a Range of integer.
INT8RANGE	**STRING**	n/a Range of **bigint**.
NUMRANGE	**STRING**	n/a Range of **numeric**.
TSRANGE	**STRING**	n/a Contains the string representation of a timestamp range without a time zone.
TSTZRANGE	**STRING**	n/a Contains the string representation of a timestamp range with the local system time zone.
DATERANGE	**STRING**	n/a Contains the string representation of a date range. It always has an exclusive upper-bound.
ENUM	**STRING**	**io.debezium.data.Enum** Contains the string representation of the PostgreSQL **ENUM** value. The set of allowed values is maintained in the **allowed** schema parameter.

Temporal types

Other than PostgreSQL's **TIMESTAMPTZ** and **TIMETZ** data types, which contain time zone information, how temporal types are mapped depends on the value of the **time.precision.mode** connector configuration property. The following sections describe these mappings:

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

time.precision.mode=adaptive

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

Table 3.8. Mappings when **time.precision.mode** is **adaptive**

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

time.precision.mode=adaptive_time_microseconds

When the **time.precision.mode** configuration property is set to **adaptive_time_microseconds**, the connector determines the literal type and semantic type for temporal types based on the column's data type definition. This ensures that events *exactly* represent the values in the database, except all **TIME** fields are captured as microseconds.

Table 3.9. Mappings when **time.precision.mode** is **adaptive_time_microseconds**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
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.

time.precision.mode=connect

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

Table 3.10. Mappings when**time.precision.mode** is **connect**

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**org.apache.kafka.connect.data.Time** Represents the number of milliseconds since midnight, and does not include timezone information. PostgreSQL allows **P** to be in the range 0-6 to store up to microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
TIMESTAMP([P])	**INT64**	**org.apache.kafka.connect.data.Timestamp** Represents the number of milliseconds since the epoch, and does not include timezone information. PostgreSQL allows **P** to be in the range 0-6 to store up to microsecond precision, though this mode results in a loss of precision when **P** is greater than 3.

TIMESTAMP type

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

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

Decimal types

The setting of the PostgreSQL connector configuration property, **decimal.handling.mode** determines how the connector maps decimal types.

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

Table 3.11. Mappings when **decimal.handling.mode** is **precise**

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

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

Table 3.12. Mappings of decimal types when there are no scale constraints

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

When the **decimal.handling.mode** property is set to **double**, the connector represents all **DECIMAL** and **NUMERIC** values as Java double values and encodes them as shown in the following table.

Table 3.13. Mappings when **decimal.handling.mode** is **double**

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

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

Table 3.14. Mappings when **decimal.handling.mode** is **string**

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

PostgreSQL supports **NaN** (not a number) as a special value to be stored in **DECIMAL**/**NUMERIC** values when the setting of **decimal.handling.mode** is **string** or **double**. In this case, the connector encodes **NaN** as either **Double.NaN** or the string constant **NAN**.

HSTORE type

When the **hstore.handling.mode** connector configuration property is set to **json** (the default), the connector represents **HSTORE** values as string representations of JSON values and encodes them as shown in the following table. When the **hstore.handling.mode** property is set to **map**, the connector uses the **MAP** schema type for **HSTORE** values.

Table 3.15. Mappings for **HSTORE** data type

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

Domain types

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

IMPORTANT

Capturing changes in columns that use PostgreSQL domain types requires special consideration. When a column is defined to contain a domain type that extends one of the default database types and the domain type defines a custom length or scale, the generated schema inherits that defined length or scale.

When a column is defined to contain a domain type that extends another domain type that defines a custom length or scale, the generated schema does **not** inherit the defined length or scale because that information is not available in the PostgreSQL driver's column metadata.

Network address types

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

Table 3.16. Mappings for network address types

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
INET	**STRING**	n/a IPv4 and IPv6 networks
CIDR	**STRING**	n/a IPv4 and IPv6 hosts and networks

PostgreSQL data type	Literal type (schema type)	Semantic type (schema name)
MACADDR	**STRING**	n/a MAC addresses
MACADDR8	**STRING**	n/a MAC addresses in EUI-64 format

PostGIS types

The PostgreSQL connector supports all PostGIS data types.

Table 3.17. Mappings of PostGIS data types

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

Toasted values

PostgreSQL has a hard limit on the page size. This means that values that are larger than around 8 KBs need to be stored by using link::https://www.postgresql.org/docs/current/storage-toast.html[TOAST storage]. This impacts replication messages that are coming from the database. Values that were stored by using the TOAST mechanism and that have not been changed are not included in the message, unless they are part of the table's replica identity. There is no safe way for Debezium to read the missing value out-of-bands directly from the database, as this would potentially lead to race conditions. Consequently, Debezium follows these rules to handle toasted values:

- Tables with **REPLICA IDENTITY FULL** - TOAST column values are part of the **before** and **after** fields in change events just like any other column.
- Tables with **REPLICA IDENTITY DEFAULT** - When receiving an **UPDATE** event from the database, any unchanged TOAST column value that is not part of the replica identity is not contained in the event. Similarly, when receiving a **DELETE** event, no TOAST columns, if any, are in the **before** field. As Debezium cannot safely provide the column value in this case, the connector returns a placeholder value as defined by the connector configuration property, **toasted.value.placeholder**.

3.5. SETTING UP POSTGRESQL TO RUN A DEBEZIUM CONNECTOR

This release of Debezium supports only the native **pgoutput** logical replication stream. To set up PostgreSQL so that it uses the **pgoutput** plug-in, you must enable a replication slot, and configure a user with sufficient privileges to perform the replication.

Details are in the following topics:

- Section 3.5.1, "Configuring a replication slot for the Debezium **pgoutput** plug-in"
- Section 3.5.2, "Setting up PostgreSQL permissions required by Debezium connectors"
- Section 3.5.3, "Configuring PostgreSQL to manage Debezium WAL disk space consumption"

3.5.1. Configuring a replication slot for the Debezium pgoutput plug-in

PostgreSQL's logical decoding uses replication slots. To configure a replication slot, specify the following in the **postgresql.conf** file:

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

These settings instruct the PostgreSQL server as follows:

- **wal_level** - Use logical decoding with the write-ahead log.
- **max_wal_senders** - Use a maximum of one separate process for processing WAL changes.
- **max_replication_slots** - Allow a maximum of one replication slot to be created for streaming WAL changes.

Replication slots are guaranteed to retain all WAL entries that are required for Debezium even during Debezium outages. Consequently, it is important to closely monitor replication slots to avoid:

- Too much disk consumption

- Any conditions, such as catalog bloat, that can happen if a replication slot stays unused for too long

For more information, see the PostgreSQL documentation for replication slots.

NOTE

Familiarity with the mechanics and configuration of the PostgreSQL write-ahead log is helpful for using the Debezium PostgreSQL connector.

3.5.2. Setting up PostgreSQL permissions required by Debezium connectors

Setting up a PostgreSQL server to run a Debezium connector requires a database user who can perform replications. Replication can be performed only by a database user who has appropriate permissions and only for a configured number of hosts. Also, you must configure the PostgreSQL server to allow replication to take place between the server machine and the host on which the PostgreSQL connector is running.

Prerequisites

- PostgreSQL administrative permissions.

Procedure

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

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

   By default, superusers have both of the above roles.

2. 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 file example:

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

   Table 3.18. Description of entries

Item	Description
1	Instructs the server to allow replication for **<youruser>** locally, that is, on the server machine.
2	Instructs the server to allow **<youruser>** on **localhost** to receive replication changes using **IPV4**.

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

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

3.5.3. Configuring PostgreSQL to manage Debezium WAL disk space consumption

In certain cases, it is possible for PostgreSQL disk space consumed by WAL files to spike or increase out of usual proportions. There are several possible reasons for this situation:

- The LSN up to which the connector has received data is available in the **confirmed_flush_lsn** column of the server's **pg_replication_slots** view. Data that is older than this LSN is no longer available, and the database is responsible for reclaiming the disk space.
  Also in the **pg_replication_slots** view, the **restart_lsn** column contains the LSN of the oldest WAL that the connector might require. If the value for **confirmed_flush_lsn** is regularly increasing and the value of **restart_lsn** lags then the database needs to reclaim the space.

  The database typically reclaims disk space in batch blocks. This is expected behavior and no action by a user is necessary.

- There are many updates in a database that is being tracked but only a tiny number of updates are related to the table(s) and schema(s) for which the connector is capturing changes. This situation can be easily solved with periodic heartbeat events. Set the **heartbeat.interval.ms** connector configuration property.

- The PostgreSQL instance contains multiple databases and one of them is a high-traffic database. Debezium captures changes in another database that is low-traffic in comparison to the other database. Debezium then cannot confirm the LSN as replication slots work per-database and Debezium is not invoked. As WAL is shared by all databases, the amount used tends to grow until an event is emitted by the database for which Debezium is capturing changes. To overcome this, it is necessary to:

  - Enable periodic heartbeat record generation with the **heartbeat.interval.ms** connector configuration property.

  - Regularly emit change events from the database for which Debezium is capturing changes.

  A separate process would then periodically update the table by either inserting a new row or repeatedly updating the same row. PostgreSQL then invokes Debezium, which confirms the latest LSN and allows the database to reclaim the WAL space. This task can be automated by means of the **heartbeat.action.query** connector configuration property.

3.6. DEPLOYING AND MANAGING DEBEZIUM POSTGRESQL CONNECTORS

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

- Section 3.6.1, "Deploying Debezium PostgreSQL connectors"

- Section 3.6.2, "Monitoring Debezium PostgreSQL connector performance"

- Section 3.6.3, "Description of Debezium PostgreSQL connector configuration properties"

3.6.1. Deploying Debezium PostgreSQL connectors

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

- A **KafkaConnect** CR that configures your Kafka Connector and that specifies the name of the image that you created to run your Debezium connector. You apply this CR to the OpenShift Kafka instance.
- A **KafkaConnector** CR that configures your Debezium PostgreSQL connector. You apply this CR to the OpenShift instance where Red Hat AMQ Streams is deployed.

Prerequisites

- PostgreSQL is running and you performed the steps to set up PostgreSQL to run a Debezium connector.
- Red Hat AMQ Streams was used to set up and start running Apache Kafka and Kafka Connect on OpenShift. AMQ Streams offers operators and images that bring Kafka to OpenShift.
- Podman or Docker is installed.
- You have an account and permissions to create and manage containers in the container registry (such as **quay.io** or **docker.io**) to which you plan to add the container that will run your Debezium connector.

Procedure

1. Create the Debezium PostgreSQL container for Kafka Connect:
    a. Download the Debezium PostgreSQL connector archive.
    b. Extract the Debezium PostgreSQL connector archive to create a directory structure for the connector plug-in, for example:

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

    c. Create a Docker file that uses **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** as the base image. For example, from a terminal window, enter the following:

```
cat <<EOF >debezium-container-for-postgresql.yaml (1)
FROM {DockerKafkaConnect}
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) - You can specify any file name that you want.

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

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

d. Build the container image from the **debezium-container-for-postgresql.yaml** Docker file that you created in the previous step. From the directory that contains the file, run the following command:

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

This command builds a container image with the name **debezium-container-for-postgresql**.

e. Push your custom image to a container registry such as **quay.io** or any internal container registry. Ensure that this registry is reachable from your OpenShift instance. For example:

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

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations: strimzi.io/use-connector-resources: "true" 1
spec:
  image: debezium-container-for-postgresql 2
```

(1) - **metadata.annotations** indicates to the Cluster Operator that **KafkaConnector** resources are used to configure connectors in this Kafka Connect cluster.

(2) - **spec.image** specifies the name of the image that you created to run your Debezium connector. This property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable in the Cluster Operator.

g. Apply your **KafkaConnect** CR to the OpenShift Kafka instance by running the following command:

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

This updates your Kafka Connect environment in OpenShift to add a Kafka Connector instance that specifies the name of the image that you created to run your Debezium connector.

2. Create a **KafkaConnector** custom resource that configures your Debezium PostgreSQL connector instance.
You configure a Debezium PostgreSQL connector in a **.yaml** file that sets connector configuration properties. A connector configuration might instruct Debezium to produce events for a subset of the schemas and tables, or it might set properties so that Debezium ignores, masks, or truncates values in specified columns that are sensitive, too large, or not needed. See the complete list of PostgreSQL connector properties that can be specified in these configurations.

The following example configures a Debezium connector that connects to a PostgreSQL server host, **192.168.99.100**, on port **5432**. This host has a database named **sampledb**, a schema named **public**, and **fulfillment** is the server's logical name.

fulfillment-connector.yaml

```
apiVersion: kafka.strimzi.io/v1alpha1
  kind: KafkaConnector
  metadata:
    name: fulfillment-connector (1)
    labels:
      strimzi.io/cluster: my-connect-cluster
  spec:
    class: io.debezium.connector.postgresql.PostgresConnector
    tasksMax: 1 (2)
    config: (3)
      database.hostname: 192.168.99.100 (4)
      database.port: 5432
      database.user: debezium
      database.password: dbz
      database.dbname: sampledb
      database.server.name: fulfillment (5)
      schema.include.list: public (6)
      plugin.name: pgoutput (7)
```

(1) - 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 name of the database host that is running the PostgreSQL server. In this example, the database host name is **192.168.99.100**.

(5) - A unique server name. The server name is the logical identifier for the PostgreSQL server or cluster of servers. This name is used as the prefix for all Kafka topics that receive change event records.

(6) - The connector captures changes in only the **public** schema. It is possible to configure the connector to capture changes in only the tables that you choose. See **table.include.list** connector configuration property.

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

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

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

This registers **fulfillment-connector** and the connector starts to run against the **sampledb** database as defined in the **KafkaConnector** CR.

4. Verify that the connector was created and has started:

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

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

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

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

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

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

   ```
   oc get kafkatopics
   ```

Results

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

3.6.2. Monitoring Debezium PostgreSQL connector performance

The Debezium PostgreSQL connector provides two types of metrics that are in addition to the built-in support for JMX metrics that Zookeeper, Kafka, and Kafka Connect provide.

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

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

3.6.2.1. Monitoring Debezium during snapshots of PostgreSQL databases

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

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.

Attributes	Type	Description
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.
QueueTotalCapacity	**int**	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	**int**	The total number of tables that are being included in the snapshot.
RemainingTableCount	**int**	The number of tables that the snapshot has yet to copy.
SnapshotRunning	**boolean**	Whether the snapshot was started.
SnapshotAborted	**boolean**	Whether the snapshot was aborted.
SnapshotCompleted	**boolean**	Whether the snapshot completed.
SnapshotDurationInSeconds	**long**	The total number of seconds that the snapshot has taken so far, even if not complete.

Attributes	Type	Description
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.

3.6.2.2. Monitoring Debezium PostgreSQL connector record streaming

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

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

Attributes	Type	Description
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event’s timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

3.6.3. Description of Debezium PostgreSQL connector configuration properties

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

- Required configuration properties
- Advanced configuration properties
- Pass-through configuration properties

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

Table 3.19. Required connector configuration properties

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.

Property	Default	Description
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.postgresql.PostgresConnector** for the PostgreSQL connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The PostgreSQL connector always uses a single task and therefore does not use this value, so the default is always acceptable.
plugin.name	**decoderbufs**	The name of the PostgreSQL logical decoding plug-in installed on the PostgreSQL server. The only supported value is **pgoutput**. You must explicitly set **plugin.name** to **pgoutput**.
slot.name	**debezium**	The name of the PostgreSQL logical decoding slot that was created for streaming changes from a particular plug-in for a particular database/schema. The server uses this slot to stream events to the Debezium connector that you are configuring. Slot names must conform to PostgreSQL replication slot naming rules, which state: *"Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character."*
slot.drop.on.stop	**false**	Whether or not to delete the logical replication slot when the connector stops in a graceful, expected way. The default behavior is that the replication slot remains configured for the connector when the connector stops. When the connector restarts, having the same replication slot enables the connector to start processing where it left off. Set to **true** in only testing or development environments. Dropping the slot allows the database to discard WAL segments. When the connector restarts it performs a new snapshot or it can continue from a persistent offset in the Kafka Connect offsets topic.

Property	Default	Description
publication.name	**dbz_publication**	The name of the PostgreSQL publication created for streaming changes when using **pgoutput**. This publication is created at start-up if it does not already exist and it includes *all tables*. Debezium then applies its own whitelist/blacklist filtering, if configured, to limit the publication to change events for the specific tables of interest. The connector user must have superuser permissions to create this publication, so it is usually preferable to create the publication before starting the connector for the first time. If the publication already exists, either for all tables or configured with a subset of tables, Debezium uses the publication as it is defined.
database.hostname		IP address or hostname of the PostgreSQL database server.
database.port	**5432**	Integer port number of the PostgreSQL database server.
database.user		Name of the PostgreSQL database user for connecting to the PostgreSQL database server.
database.password		Password to use when connecting to the PostgreSQL database server.
database.dbname		The name of the PostgreSQL database from which to stream the changes.
database.server.name		Logical name that identifies and provides a namespace for the particular PostgreSQL database server or cluster in which Debezium is capturing changes. Only alphanumeric characters and underscores should be used in the database server logical name. The logical name should be unique across all other connectors, since it is used as a topic name prefix for all Kafka topics that receive records from this connector.

Property	Default	Description
schema.whitelist		An optional, comma-separated list of regular expressions that match names of schemas for which you **want** to capture changes. Any schema name not included in the whitelist is excluded from having its changes captured. By default, all non-system schemas have their changes captured. Do not also set the **schema.blacklist** property.
schema.blacklist		An optional, comma-separated list of regular expressions that match names of schemas for which you **do not** want to capture changes. Any schema whose name is not included in the blacklist has its changes captured, with the exception of system schemas. Do not also set the **schema.whitelist** property.
table.whitelist		An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you want to capture. Any table not included in the whitelist does not have its changes captured. Each identifier is of the form *schemaName*.*tableName*. By default, the connector captures changes in every non-system table in each schema whose changes are being captured. Do not also set the **table.blacklist** property.
table.blacklist		An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you **do not** want to capture. Any table not included in the blacklist has it changes captured. Each identifier is of the form *schemaName*.*tableName*. Do not also set the **table.whitelist** property.
column.whitelist		An optional, comma-separated list of regular expressions that match the fully-qualified names of columns that should be included in change event record values. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*. Do not also set the **column.blacklist** property.

Property	Default	Description
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 record values. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Do not also set the **column.whitelist** property.
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision: **adaptive** captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type. **adaptive_time_microseconds** captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type. An exception is **TIME** type fields, which are always captured as microseconds. **connect** always represents time and timestamp values by using Kafka Connect's built-in representations for **Time**, **Date**, and **Timestamp**, which use millisecond precision regardless of the database columns' precision. See temporal values.
decimal.handling.mode	**precise**	Specifies how the connector should handle values for **DECIMAL** and **NUMERIC** columns: **precise** represents values by using **java.math.BigDecimal** to represent values in binary form in change events. **double** represents values by using **double** values, which might result in a loss of precision but which is easier to use. **string** encodes values as formatted strings, which are easy to consume but semantic information about the real type is lost. See Decimal types.

Property	Default	Description
hstore.handling.mode	**map**	Specifies how the connector should handle values for **hstore** columns: **map** represents values by using **MAP**. **json** represents values by using **json string**. This setting encodes values as formatted strings such as **{"key" : "val"}**. See PostgreSQL **HSTORE** type.
interval.handling.mode	**numeric**	Specifies how the connector should handle values for **interval** columns: **numeric** represents intervals using approximate number of microseconds. **string** represents intervals exactly by using the string pattern representation **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**. For example: **P1Y2M3DT4H5M6.78S**. See PostgreSQL basic types.
database.sslmode	**disable**	Whether to use an encrypted connection to the PostgreSQL server. Options include: **disable** uses an unencrypted connection. **require** uses a secure (encrypted) connection, and fails if one cannot be established. **verify-ca** behaves like **require** but also verifies the server TLS certificate against the configured Certificate Authority (CA) certificates, or fails if no valid matching CA certificates are found. **verify-full** behaves like **verify-ca** but also verifies that the server certificate matches the host to which the connector is trying to connect. See the PostgreSQL documentation for more information.
database.sslcert		The path to the file that contains the SSL certificate for the client. See the PostgreSQL documentation for more information.
database.sslkey		The path to the file that contains the SSL private key of the client. See the PostgreSQL documentation for more information.

Property	Default	Description
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 that contains the root certificate(s) against which the server is validated. See the PostgreSQL documentation for more information.
database.tcpKeepAlive	**true**	Enable TCP keep-alive probe to verify that the database connection is still alive. See the PostgreSQL documentation for more information.
tombstones.on.delete	**true**	Controls whether a tombstone event should be generated after a *delete* event. **true** - delete operations are represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is sent. After a *delete* operation, emitting a tombstone event enables Kafka to delete all change event records that have the same key as the deleted row.
column.truncate.to.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event records, values in these columns are truncated if they are longer than the number of characters specified by *length* in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, **column.truncate.to.20.chars**.

Property	Default	Description
column.mask.with .*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*. In change event values, the values in the specified table columns are replaced with *length* number of asterisk (*****) characters. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer or zero. When you specify zero, the connector replaces a value with an empty string.
column.mask .hash.*hashAlgorithm* .with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*. In change event values, the values in the specified columns are replaced with pseudonyms. A pseudonym consists of the hashed value that results from applying the specifed *hashAlgorithm* and *salt*. Based on the hash function that is used, referential integrity is kept while column values are replaced with pseudonyms. Supported hash functions are described in the {link-java7-standard-names}[MessageDigest section] of the Java Cryptography Architecture Standard Algorithm Name Documentation. If necessary, the pseudonym is automatically shortened to the length of the column. You can specify multiple properties with different hash algorithms and salts in a single configuration. In the following example, **CzQMA0cB5K** is a randomly selected salt. **column.mask.hash.SHA-256.with.salt.CzQMA0cB5K =inventory.orders.customerName,inventory.shipment.customerName** Depending on the *hashAlgorithm* used, the *salt* selected, and the actual data set, the resulting masked data set might not be completely masked.

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

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

Property	Default	Description
publication .autocreate.mode	*all_tables*	Applies only when streaming changes by using the **pgoutput** plug-in. The setting determines how creation of a publication should work. Possible settings are: **all_tables** - If a publication exists, the connector uses it. If a publication does not exist, the connector creates a publication for all tables in the database for which the connector is capturing changes. This requires that the database user who has permission to perform replications also has permission to create a publication. This is granted with **CREATE PUBLICATION <publication_name> FOR ALL TABLES;**. **disabled** - The connector does not attempt to create a publication. A database administrator or the user configured to perform replications must have created the publication before running the connector. If the connector cannot find the publication, the connector throws an exception and stops. **filtered** - If a publication exists, the connector uses it. If no publication exists, the connector creates a new publication for tables that match the current filter configuration as specified by the **database.exclude.list**, **database.include.list**, **table.exclude.list**, and **table.include.list** connector configuration properties. For example: **CREATE PUBLICATION <publication_name> FOR TABLE <tbl1, tbl2, tbl3>**.
binary.handling.mode	bytes	Specifies how binary (**bytea**) columns should be represented in change events: **bytes** represents binary data as byte array. **base64** represents binary data as base64-encoded strings. **hex** represents binary data as hex-encoded (base16) strings.

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

Table 3.20. Advanced connector configuration properties

Property	Default	Description
snapshot.mode	**initial**	Specifies the criteria for performing a snapshot when the connector starts: **initial** - The connector performs a snapshot only when no offsets have been recorded for the logical server name. **always** - The connector performs a snapshot each time the connector starts. **never** - The connector never performs snapshots. When a connector is configured this way, its behavior when it starts is as follows. If there is a previously stored LSN in the Kafka offsets topic, the connector continues streaming changes from that position. If no LSN has been stored, the connector starts streaming changes from the point in time when the PostgreSQL logical replication slot was created on the server. The **never** snapshot mode is useful only when you know all data of interest is still reflected in the WAL. **initial_only** - The connector performs an initial snapshot and then stops, without processing any subsequent changes. **exported** - The connector performs a snapshot based on the point in time when the replication slot was created. This is an excellent way to perform the snapshot in a lock-free way. Thereference table for snapshot mode settings has more details.
snapshot.lock.timeout.ms	**10000**	Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this time interval, the snapshot fails. How the connector performs snapshots provides details.

Property	Default	Description
snapshot.select .statement.overrides		Controls which table rows are included in snapshots. This property affects snapshots only. It does not affect events that are generated by the logical decoding plug-in. Specify a comma-separated list of fully-qualified table names in the form *databaseName.tableName*. For each table that you specify, also specify another configuration property: **snapshot.select.statement.overrides.*DB_NAME.TABLE_NAME***, for example: **snapshot.select.statement.overrides.customers.orders**. Set this property to a **SELECT** statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this **SELECT** statement to retrieve data from that table. A possible use case for setting these properties is large, append-only tables. You can specify a **SELECT** statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.
event.processing .failure.handling.mode	**fail**	Specifies how the connector should react to exceptions during processing of events: **fail** propagates the exception, indicates the offset of the problematic event, and causes the connector to stop. **warn** logs the offset of the problematic event, skips that event, and continues processing. **skip** skips the problematic event and continues processing.
max.queue.size	**20240**	Positive integer value for the maximum size of the blocking queue. The connector places change events received from streaming replication in the blocking queue before writing them to Kafka. This queue can provide backpressure when, for example, writing records to Kafka is slower that it should be or Kafka is not available.

Property	Default	Description
max.batch.size	**10240**	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait for new change events to appear before it starts processing a batch of events. Defaults to 1000 milliseconds, or 1 second.
include.unknown .datatypes	**false**	Specifies connector behavior when the connector encounters a field whose data type is unknown. The default behavior is that the connector omits the field from the change event and logs a warning. Set this property to **true** if you want the change event to contain an opaque binary representation of the field. This lets consumers decode the field. You can control the exact representation by setting the **binary handling mode** property. Consumers risk backward compatibility issues when **include.unknown.datatypes** is set to **true**. Not only may the database-specific binary representation change between releases, but if the data type is eventually supported by Debezium, the data type will be sent downstream in a logical type, which would require adjustments by consumers. In general, when encountering unsupported data types, create a feature request so that support can be added.

Property	Default	Description
database.initial .statements		A semicolon separated list of SQL statements that the connector executes when it establishes a JDBC connection to the database. To use a semicolon as a character and not as a delimiter, specify two consecutive semicolons, **;;**. The connector may establish JDBC connections at its own discretion. Consequently, this property is useful for configuration of session parameters only, and not for executing DML statements. The connector does not execute these statements when it creates a connection for reading the transaction log.

Property	Default	Description
heartbeat.interval.ms	**0**	Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages. Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are needed when there are many updates in a database that is being tracked but only a tiny number of updates are related to the table(s) and schema(s) for which the connector is capturing changes. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that no offset updates are committed to Kafka and the connector does not have an opportunity to send the latest retrieved LSN to the database. The database retains WAL files that contain events that have already been processed by the connector. Sending heartbeat messages enables the connector to send the latest retrieved LSN to the database, which allows the database to reclaim disk space being used by no longer needed WAL files.
heartbeat.topics.prefix	**__debezium-heartbeat**	Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern: *<heartbeat.topics.prefix>.<server.name>* For example, if the database server name is **fullfillment**, the default topic name is **__debezium-heartbeat.fulfillment**.

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

Property	Default	Description
snapshot.delay.ms		An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
snapshot.fetch.size	**10240**	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
slot.stream.params		Semicolon separated list of parameters to pass to the configured logical decoding plug-in. For example, **add-tables=public.table,public.table2;include-lsn=true**.
sanitize.field.names	**true** if connector configuration sets the **key.converter** or **value.converter** property to the Avro converter. **false** if not.	Indicates whether field names are sanitized to adhere to Avro naming requirements.
slot.max.retries	**6**	If connecting to a replication slot fails, this is the maximum number of consecutive attempts to connect.
slot.retry.delay.ms	**10000** (10 seconds)	The number of milliseconds to wait between retry attempts when the connector fails to connect to a replication slot.
toasted.value.placeholder	**__debezium_unavailable_value**	Specifies the constant that the connector provides to indicate that the original value is a toasted value that is not provided by the database. If the setting of **toasted.value.placeholder** starts with the **hex:** prefix it is expected that the rest of the string represents hexadecimally encoded octets. See toasted values for additional details.

Property	Default	Description
provide.transaction .metadata	**false**	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify **true** if you want the connector to do this. See Transaction metadata for details.

Pass-through connector configuration properties

The connector also supports *pass-through* configuration properties that are used when creating the Kafka producer and consumer.

Be sure to consult the Kafka documentation for all of the configuration properties for Kafka producers and consumers. The PostgreSQL connector does use the new consumer configuration properties.

3.7. HOW DEBEZIUM POSTGRESQL CONNECTORS HANDLE FAULTS AND PROBLEMS

Debezium is a distributed system that captures all changes in multiple upstream databases; it never misses or loses an event. When the system is operating normally or being managed carefully then Debezium provides *exactly once* delivery of every change event record.

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

Details are in the following sections:

- Configuration and startup errors
- PostgreSQL becomes unavailable
- Cluster failures
- Kafka Connect process stops gracefully
- Kafka Connect process crashes
- Kafka becomes unavailable
- Connector is stopped for a duration

Configuration and startup errors

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

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

- The connector is restarting from a previously-recorded position in the PostgreSQL WAL (by using the LSN) and PostgreSQL no longer has that history available.

In these cases, the error message has details about the problem and possibly a suggested workaround. After you correct the configuration or address the PostgreSQL problem, restart the connector.

PostgreSQL becomes unavailable

When the connector is running, the PostgreSQL server that it is connected to could become unavailable for any number of reasons. If this happens, the connector fails with an error and stops. When the server is available again, restart the connector.

The PostgreSQL connector externally stores the last processed offset in the form of a PostgreSQL LSN. After a connector restarts and connects to a server instance, the connector communicates with the server to continue streaming from that particular offset. This offset is available as long as the Debezium replication slot remains intact. Never drop a replication slot on the primary server or you will lose data. See the next section for failure cases in which a slot has been removed.

Cluster failures

As of release 12, PostgreSQL allows logical replication slots *only on primary servers*. This means that you can point a Debezium PostgreSQL connector to only the active primary server of a database cluster. Also, replication slots themselves are not propagated to replicas. If the primary server goes down, a new primary must be promoted.

The new primary must have a replication slot that is configured for use by the **pgoutput** plug-in and the database in which you want to capture changes. Only then can you point the connector to the new server and restart the connector.

There are important caveats when failovers occur and you should pause Debezium until you can verify that you have an intact replication slot that has not lost data. After a failover:

- There must be a process that re-creates the Debezium replication slot before allowing the application to write to the **new** primary. This is crucial. Without this process, your application can miss change events.
- You might need to verify that Debezium was able to read all changes in the slot **before the old primary failed**.

One reliable method of recovering and verifying whether any changes were lost is to recover a backup of the failed primary to the point immediately before it failed. While this can be administratively difficult, it allows you to inspect the replication slot for any unconsumed changes.

Kafka Connect process stops gracefully

Suppose that Kafka Connect is being run in distributed mode and a Kafka Connect process is stopped gracefully. Prior to shutting down that process, Kafka Connect migrates the process's connector tasks to another Kafka Connect process in that group. The new connector tasks start processing exactly where the prior tasks stopped. There is a short delay in processing while the connector tasks are stopped gracefully and restarted on the new processes.

Kafka Connect process crashes

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

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

In each change event record, Debezium connectors insert source-specific information about the origin of the event, including the PostgreSQL server's time of the event, the ID of the server transaction, and the position in the write-ahead log where the transaction changes were written. Consumers can keep track of this information, especially the LSN, to determine whether an event is a duplicate.

Kafka becomes unavailable

As the connector generates change events, the Kafka Connect framework records those events in Kafka by using the Kafka producer API. Periodically, at a frequency that you specify in the Kafka Connect configuration, Kafka Connect records the latest offset that appears in those change events. If the Kafka brokers become unavailable, the Kafka Connect process that is running the connectors repeatedly tries to reconnect to the Kafka brokers. In other words, the connector tasks pause until a connection can be re-established, at which point the connectors resume exactly where they left off.

Connector is stopped for a duration

If the connector is gracefully stopped, the database can continue to be used. Any changes are recorded in the PostgreSQL WAL. When the connector restarts, it resumes streaming changes where it left off. That is, it generates change event records for all database changes that were made while the connector was stopped.

A properly configured Kafka cluster is able to handle massive throughput. Kafka Connect is written according to Kafka best practices, and given enough resources a Kafka Connect connector can also handle very large numbers of database change events. Because of this, after being stopped for a while, when a Debezium connector restarts, it is very likely to catch up with the database changes that were made while it was stopped. How quickly this happens depends on the capabilities and performance of Kafka and the volume of changes being made to the data in PostgreSQL.

CHAPTER 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. Data change events

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

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

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

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

Table 4.1. Overview of change event basic content

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

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

WARNING

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

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

4.4.6.1. Change event keys

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

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

Example document

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

Example change event key

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

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

Table 4.2. Description of change event key

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

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

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

Type	MongoDB **_id** Value	Key's payload
Integer	1234	**{ "id" : "1234" }**
Float	12.34	**{ "id" : "12.34" }**

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

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

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

Example document

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

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

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

4.4.6.3. *create* events

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

```
{
    "schema": { 1
      "type": "struct",
      "fields": [
{
  "type": "string",
  "optional": true,
  "name": "io.debezium.data.Json", (2)
  "version": 1,
  "field": "after"
},
{
  "type": "string",
  "optional": true,
  "name": "io.debezium.data.Json",
  "version": 1,
  "field": "patch"
},
{
  "type": "string",
  "optional": true,
  "name": "io.debezium.data.Json",
  "version": 1,
  "field": "filter"
},
{
  "type": "struct",
  "fields": [
    {
      "type": "string",
      "optional": false,
      "field": "version"
    },
    {
      "type": "string",
      "optional": false,
      "field": "connector"
    },
    {
      "type": "string",
      "optional": false,
      "field": "name"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "ts_ms"
    },
    {
      "type": "boolean",
      "optional": true,
      "default": false,
      "field": "snapshot"
    },
    {
      "type": "string",
      "optional": false,
      "field": "db"
    },
    {
```

```
              "type": "string",
              "optional": false,
              "field": "rs"
            },
            {
              "type": "string",
              "optional": false,
              "field": "collection"
            },
            {
              "type": "int32",
              "optional": false,
              "field": "ord"
            },
            {
              "type": "int64",
              "optional": true,
              "field": "h"
            }
          ],
          "optional": false,
          "name": "io.debezium.connector.mongo.Source", (3)
          "field": "source"
        },
        {
          "type": "string",
          "optional": true,
          "field": "op"
        },
        {
          "type": "int64",
          "optional": true,
          "field": "ts_ms"
        }
      ],
      "optional": false,
      "name": "dbserver1.inventory.customers.Envelope" (4)
      },
    "payload": { (5)
      "after": "{\"_id\" : {\"$numberLong\" : \"1004\"},\"first_name\" : \"Anne\",\"last_name\" :
\"Kretchmar\",\"email\" : \"annek@noanswer.org\"}", (6)
      "patch": null,
      "source": { (7)
        "version": "1.2.4.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": false,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 31,
        "h": 1546547425148721999
      },
      "op": "c", (8)
      "ts_ms": 1558965515240 9
    }
  }
```

Table 4.4. Descriptions of *create* event value fields

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

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

4.4.6.4. *update* events

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

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

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

```
{
    "schema": { ... },
    "payload": {
      "op": "u", 1
      "ts_ms": 1465491461815, 2
      "patch": "{\"$set\":{\"first_name\":\"Anne Marie\"}}", 3
      "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", 4
      "source": { 5
        "version": "1.2.4.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": true,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

Table 4.5. Descriptions of *update* event value fields

Item	Field name	Description
1	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **u** indicates that the operation updated a document.
2	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
3	**patch**	Contains the JSON string representation of the actual MongoDB idempotent change to the document. In this example, the update changed the **first_name** field to a new value. An *update* event value does not contain an**after** field.

Item	Field name	Description
4	**filter**	Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be updated.
5	**source**	Mandatory field that describes the source metadata for the event. This field contains the same information as a *create* event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes: ● Debezium version. ● Name of the connector that generated the event. ● Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. ● Names of the collection and database that contain the updated document. ● If the event was part of a snapshot. ● Timestamp for when the change was made in the database and ordinal of the event within the timestamp. ● Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the **h** field in the oplog event, or a field named **stxnid**, which represents the **lsid** and **txnNumber** fields from the oplog event.

WARNING

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

NOTE

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

4.4.6.5. *delete* events

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

```
{
  "schema": { ... },
  "payload": {
    "op": "d", ❶
    "ts_ms": 1465495462115, ❷
    "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", ❸
    "source": { ❹
      "version": "1.2.4.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 6,
      "h": 1546547425148721999
    }
  }
}
```

Table 4.6. Descriptions of *delete* event value fields

Item	Field name	Description
1	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this document was deleted.
2	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.
3	**filter**	Contains the JSON string representation of the MongoDB selection criteria that was used to identify the document to be deleted.

Item	Field name	Description
4	**source**	Mandatory field that describes the source metadata for the event. This field contains the same information as a *create* or *update* event for the same collection, but the values are different since this event is from a different position in the oplog. The source metadata includes: • Debezium version. • Name of the connector that generated the event. • Logical name of the MongoDB replica set, which forms a namespace for generated events and is used in Kafka topic names to which the connector writes. • Names of the collection and database that contained the deleted document. • If the event was part of a snapshot. • Timestamp for when the change was made in the database and ordinal of the event within the timestamp. • Unique identifier of the MongoDB operation, which depends on the version of MongoDB. It is either the **h** field in the **oplog** event, or a field named **stxnid**, which represents the **lsid** and **txnNumber** fields from the **oplog** event.

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

Tombstone events

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

4.4.7. Transaction Metadata

Debezium can generate events that represents 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": "1462833718356672513",
  "event_count": null,
  "data_collections": null
}

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

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

4.4.7.2. Data events enrichment

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

- **id** - string representation of unique transaction identifier
- **total_order** - the absolute position of the event among all events generated by the transaction
- **data_collection_order** - the per-data collection position of the event among all events that were emitted by the transaction

Following is an example of what a message looks like:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
...
  },
  "op": "c",
    "ts_ms": "1580390884335",
    "transaction": {
      "id": "1462833718356672513",
      "total_order": "1",
      "data_collection_order": "1"
    }
  }
```

4.5. DEPLOYING THE MONGODB CONNECTOR

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

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

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

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

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

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

Additional resources

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

- Installing Debezium on OpenShift
- Installing Debezium on RHEL

4.5.1. Example configuration

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

Following is an example of the configuration for a MongoDB connector that monitors a MongoDB replica set **rs0** at port 27017 on 192.168.99.100, which we logically name **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
```

Table 4.7. Description of settings

Item	Description
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 converter 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. Adding connector configuration

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

Prerequisites

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

Procedure

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

   ```
   tree ./my-plugins/
   ./my-plugins/
   ├── debezium-connector-mongodb
   │   ├── ...
   ```

2. Create and publish a custom image for running your Debezium connector:

   a. Create a new **Dockerfile** by using **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** 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-rhel7:1.5.0
   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 container image. For example, if you saved the **Dockerfile** that you created in the previous step as **debezium-container-for-mongodb**, and if the **Dockerfile** is in the current directory, then you would run the following command:
   podman build -t debezium-container-for-mongodb:latest .

   c. Push your custom image to your container registry, for example:
   podman push debezium-container-for-mongodb:latest

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

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

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

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

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

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

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

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

    a. Display the Kafka Connect log output:

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

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

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

Results

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

4.5.3. Monitoring

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

- snapshot metrics; for monitoring the connector when performing snapshots
- streaming metrics; for monitoring the connector when processing oplog events

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

4.5.3.1. Snapshot Metrics

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

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

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

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

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

4.5.3.2. Streaming Metrics

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

Attributes	Type	Description
LastEvent	**string**	The last streaming event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.
TotalNumberOfEventsSeen	**long**	The total number of events that this connector has seen since last started or reset.
NumberOfEventsFiltered	**long**	The number of events that have been filtered by whitelist or blacklist filtering rules configured on the connector.
MonitoredTables	**string[]**	The list of tables that are monitored by the connector.
QueueTotalCapacity	**int**	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	**int**	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.

Attributes	Type	Description
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

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.

4.5.4. 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.mongodb.MongoDbConnector** for the MongoDB connector.

Property	Default	Description
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.
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**.

Property	Default	Description
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.
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.

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

Property	Default	Description
snapshot.delay.ms		An interval in milli-seconds that the connector should wait before taking a snapshot after starting up; Can be used to avoid snapshot interruptions when starting multiple connectors in a cluster, which may cause re-balancing of connectors.
snapshot.fetch.size	**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).

Property	Default	Description
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.

Property	Default	Description
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

Reconnection attempt number	Delay before attempt, in seconds	Total delay before attempt, in minutes and seconds
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. Schema change topic

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

WARNING

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

Debezium emits a message to the schema change topic when:

- You enable CDC for a table.
- You disable CDC for a table.
- You alter the structure of a table for which CDC is enabled by following the schema evolution procedure.

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

```
{
 "schema": {
 ...
 },
 "payload": {
  "source": {
   "version": "1.2.4.Final",
   "connector": "sqlserver",
   "name": "server1",
   "ts_ms": 1588252618953,
   "snapshot": "true",
   "db": "testDB",
   "schema": "dbo",
   "table": "customers",
   "change_lsn": null,
   "commit_lsn": "00000025:00000d98:00a2",
   "event_serial_no": null
  },
  "databaseName": "testDB", 1
  "schemaName": "dbo",
  "ddl": null, 2
  "tableChanges": [ 3
   {
    "type": "CREATE", 4
    "id": "\"testDB\".\"dbo\".\"customers\"", 5
    "table": { 6
     "defaultCharsetName": null,
     "primaryKeyColumnNames": [ 7
```

```
      "id"
    ],
    "columns": [ 8
      {
        "name": "id",
        "jdbcType": 4,
        "nativeType": null,
        "typeName": "int identity",
        "typeExpression": "int identity",
        "charsetName": null,
        "length": 10,
        "scale": 0,
        "position": 1,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "first_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 2,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "last_name",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 3,
        "optional": false,
        "autoIncremented": false,
        "generated": false
      },
      {
        "name": "email",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "varchar",
        "typeExpression": "varchar",
        "charsetName": null,
        "length": 255,
        "scale": null,
        "position": 4,
        "optional": false,
              "autoIncremented": false,
              "generated": false
            }
          ]
        }
      }
    ]
  }
}
```

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

Item	Field name(s)	Description
1	**databaseName** **schemaName**	Identifies the database and the schema that contain the change.
2	**ddl**	Always **null** for the SQL Server connector. For other connectors, this field contains the DDL responsible for the schema change. This DDL is not available to SQL Server connectors.
3	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.
4	**type**	Describes the kind of change. The value is one of the following: • **CREATE** - table created • **ALTER** - table modified • **DROP** - table deleted
5	**id**	Full identifier of the table that was created, altered, or dropped.
6	**table**	Represents table metadata after the applied change.
7	**primaryKeyColumnNames**	List of columns that compose the table’s primary key.
8	**columns**	Metadata for each column in the changed table.

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

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

5.3.5. Change data events

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

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

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

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

Table 5.2. Overview of change event basic content

Item	Field name	Description

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

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

WARNING

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

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

5.3.5.1. Change Event Keys

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

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

Example table

```
CREATE TABLE customers (
  id INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE
);
```

Example change event key

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

```
{
    "schema": { 1
        "type": "struct",
        "fields": [ 2
            {
                "type": "int32",
                "optional": false,
                "field": "id"
            }
        ],
        "optional": false, 3
        "name": "server1.dbo.customers.Key" 4
    },
    "payload": { 5
        "id": 1004
    }
}
```

Table 5.3. Description of change event key

Item	Field name	Description
1	**schema**	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's **payload** portion.
2	**fields**	Specifies each field that is expected in the **payload**, including each field's name, type, and whether it is required. In this example, there is one required field named **id** of type **int32**.
3	**optional**	Indicates whether the event key must contain a value in its **payload** field. In this example, a value in the key's payload is required. A value in the key's payload field is optional when a table does not have a primary key.

Item	Field name	Description
4	**server1.dbo .customers.Key**	Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format *connector-name.database-schema-name.table-name*.**Key**. In this example: ● **server1** is the name of the connector that generated this event. ● **dbo** is the database schema for the table that was changed. ● **customers** is the table that was updated.
5	**payload**	Contains the key for the row for which this change event was generated. In this example, the key, contains a single **id** field whose value is **1004**.

5.3.5.2. Change event values

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

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

```
CREATE TABLE customers (
  id INTEGER IDENTITY(1001,1) NOT NULL PRIMARY KEY,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE
);
```

The value portion of a change event for a change to this table is described for each event type.

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

5.3.5.2.1. *create* events

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

```
{
  "schema": { (1)
    "type": "struct",
    "fields": [
      {
```

```
      "type": "struct",
      "fields": [
        {
          "type": "int32",
          "optional": false,
          "field": "id"
        },
        {
          "type": "string",
          "optional": false,
          "field": "first_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "server1.dbo.customers.Value", (2)
      "field": "before"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "int32",
          "optional": false,
          "field": "id"
        },
        {
          "type": "string",
          "optional": false,
          "field": "first_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "last_name"
        },
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "server1.dbo.customers.Value",
      "field": "after"
    },
    {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": false,
            "field": "schema"
          },
          {
            "type": "string",
            "optional": false,
            "field": "table"
          },
          {
            "type": "string",
            "optional": true,
            "field": "change_lsn"
          },
          {
            "type": "string",
            "optional": true,
            "field": "commit_lsn"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "event_serial_no"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.sqlserver.Source", (3)
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "server1.dbo.customers.Envelope" (4)
  },
  "payload": { (5)
    "before": null, (6)
    "after": { (7)
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "source": { (8)
      "version": "1.2.4.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
      "event_serial_no": "1"
    },
    "op": "c", (9)
    "ts_ms": 1559729471739 (10)
  }
}
```

Table 5.4. Descriptions of *create* event value fields

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

Item	Field name	Description
2	**name**	In the **schema** section, each **name** field specifies the schema for a field in the value's payload. **server1.dbo.customers.Value** is the schema for the payload's**before** and **after** fields. This schema is specific to the**customers** table. Names of schemas for **before** and **after** fields are of the form ***logicalName.database-schemaName.tableName*.Value**, which ensures that the schema name is unique in the database. This means that when using the Avro converter, the resulting Avro schema for each table in each logical source has its own evolution and history.
3	**name**	**io.debezium.connector.sqlserver.Source** is the schema for the payload's **source** field. This schema is specific to the SQL Server connector. The connector uses it for all events that it generates.
4	**name**	**server1.dbo.customers.Envelope** is the schema for the overall structure of the payload, where **server1** is the connector name, **dbo** is the database schema name, and **customers** is the table.
5	**payload**	The value's actual data. This is the information that the change event is providing. It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the Avro converter, you can significantly decrease the size of the messages that the connector streams to Kafka topics.
6	**before**	An optional field that specifies the state of the row before the event occurred. When the **op** field is **c** for create, as it is in this example, the **before** field is **null** since this change event is for new content.
7	**after**	An optional field that specifies the state of the row after the event occurred. In this example, the **after** field contains the values of the new row's **id**, **first_name**, **last_name**, and **email** columns.

Item	Field name	Description
8	**source**	Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes: • Debezium version • Connector type and name • Database and schema names • Timestamp for when the change was made in the database • If the event was part of a snapshot • Name of the table that contains the new row • Server log offsets
9	**op**	Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, **c** indicates that the operation created a row. Valid values are: • **c** = create • **u** = update • **d** = delete • **r** = read (applies to only snapshots)
10	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

5.3.5.2.2. *update* events

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

```
{
  "schema": { ... },
  "payload": {
    "before": { (1)
```

```
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "after": { 2
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "source": { 3
      "version": "1.2.4.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
      "event_serial_no": "2"
    },
    "op": "u", 4
    "ts_ms": 1559729998706 5
  }
}
```

Table 5.5. Descriptions of *update* event value fields

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

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. The **source** field structure has the same fields as in a*create* event, but some values are different, for example, the sample *update* event has a different offset. The source metadata includes: ● Debezium version ● Connector type and name ● Database and schema names ● Timestamp for when the change was made in the database ● If the event was part of a snapshot ● Name of the table that contains the new row ● Server log offsets The **event_serial_no** field differentiates events that have the same commit and change LSN. Typical situations for when this field has a value other than **1**: ● *update* events have the value set to **2** because the update generates two events in the CDC change table of SQL Server (see the source documentation for details). The first event contains the old values and the second contains contains new values. The connector uses values in the first event to create the second event. The connector drops the first event. ● When a primary key is updated SQL Server emits two evemts. A *delete* event for the removal of the record with the old primary key value and a *create* event for the addition of the record with the new primary key. Both operations share the same commit and change LSN and their event numbers are **1** and **2**, respectively.
4	**op**	Mandatory string that describes the type of operation. In an *update* event value, the **op** field value is **u**, signifying that this row changed because of an update.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

NOTE

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

5.3.5.2.3. *delete* events

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

```
{
 "schema": { ... },
 },
 "payload": {
  "before": { 1
   "id": 1005,
   "first_name": "john",
   "last_name": "doe",
   "email": "noreply@example.org"
  },
  "after": null, 2
  "source": { 3
   "version": "1.2.4.Final",
   "connector": "sqlserver",
   "name": "server1",
   "ts_ms": 1559730445243,
   "snapshot": false,
   "db": "testDB",
   "schema": "dbo",
   "table": "customers",
   "change_lsn": "00000027:00000db0:0005",
   "commit_lsn": "00000027:00000db0:0007",
   "event_serial_no": "1"
  },
  "op": "d", 4
  "ts_ms": 1559730450205 5
 }
}
```

Table 5.6. Descriptions of *delete* event value fields

Item	Field name	Description
1	**before**	Optional field that specifies the state of the row before the event occurred. In a *delete* event value, the **before** field contains the values that were in the row before it was deleted with the database commit.
2	**after**	Optional field that specifies the state of the row after the event occurred. In a *delete* event value, the **after** field is **null**, signifying that the row no longer exists.

Item	Field name	Description
3	**source**	Mandatory field that describes the source metadata for the event. In a *delete* event value, the **source** field structure is the same as for *create* and *update* events for the same table. Many **source** field values are also the same. In a *delete* event value, the **ts_ms** and **pos** field values, as well as other values, might have changed. But the **source** field in a *delete* event value provides the same metadata: • Debezium version • Connector type and name • Database and schema names • Timestamp for when the change was made in the database • If the event was part of a snapshot • Name of the table that contains the new row • Server log offsets
4	**op**	Mandatory string that describes the type of operation. The **op** field value is **d**, signifying that this row was deleted.
5	**ts_ms**	Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task. In the **source** object, **ts_ms** indicates the time that the change was made in the database. By comparing the value for **payload.source.ts_ms** with the value for **payload.ts_ms**, you can determine the lag between the source database update and Debezium.

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

Tombstone events

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

5.3.6. Transaction Metadata

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

5.3.6.1. 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**.

5.3.6.2. 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"
  }
}
```

5.3.7. 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.

5.3.7.1. 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

5.3.7.2. 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.

5.3.7.3. 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
```

5.3.8. 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)
BIT	**BOOLEAN**	n/a
TINYINT	**INT16**	n/a
SMALLINT	**INT16**	n/a
INT	**INT32**	n/a
BIGINT	**INT64**	n/a
REAL	**FLOAT32**	n/a
FLOAT[(N)]	**FLOAT64**	n/a
CHAR[(N)]	**STRING**	n/a
VARCHAR[(N)]	**STRING**	n/a
TEXT	**STRING**	n/a
NCHAR[(N)]	**STRING**	n/a
NVARCHAR[(N)]	**STRING**	n/a

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

Other data type mappings are described in the following sections.

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

5.3.8.1. 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)
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.

SQL Server data type	Literal type (schema type)	Semantic type (schema name)
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)
DATE	**INT32**	**org.apache.kafka.connect.data.Date** Represents the number of days since the epoch.
TIME([P])	**INT64**	**org.apache.kafka.connect.data.Time** Represents the number of milliseconds since midnight, and does not include timezone information. SQL Server allows **P** to be in the range 0-7 to store up to tenth of 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.

5.3.8.1.1. 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.

5.3.8.2. Decimal values

SQL Server data type	Literal type (schema type)	Semantic type (schema name)

SQL Server data type	Literal type (schema type)	Semantic type (schema name)
NUMERIC[(P[,S])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** The **scale** schema parameter contains an integer 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. DEPLOYMENT

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

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

1. Use Red Hat AMQ Streams to set up Apache Kafka and Kafka Connect on OpenShift. AMQ Streams offers operators and images that bring Kafka to OpenShift.

2. Download the Debezium SQL Server connector.

3. Extract the connector files into your Kafka Connect environment.

4. Add the connector plug-in's parent directory to your Kafka Connect **plugin.path**, for example:

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

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

5. Restart your Kafka Connect process to ensure that the new JAR files are picked up.

You also need to set up SQL Server to run a Debezium connector.

Additional resources

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

- Installing Debezium on OpenShift
- Installing Debezium on RHEL

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
```

Table 5.7. Descriptions of connector configuration settings

Item	Description
1	The name of our connector when we register it with a Kafka Connect service.
2	The name of this SQL Server connector class.
3	The address of the SQL Server instance.
4	The port number of the SQL Server instance.
5	The name of the SQL Server user.
6	The password for the SQL Server user.
7	The name of the database to capture changes from.
8	The logical name of the SQL Server instance/cluster, which forms a namespace and is used in all the names of the Kafka topics to which the connector writes, the Kafka Connect schema names, and the namespaces of the corresponding Avro schema when the Avro converter is used.
9	A list of all tables whose changes Debezium should capture.
10	The list of Kafka brokers that this connector will use to write and recover DDL statements to the database history topic.
11	The name of the database history topic where the connector will write and recover DDL statements. This topic is for internal use only and should not be used by consumers.

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. Adding connector configuration

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

Prerequisites

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

Procedure

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

   ```
   tree ./my-plugins/
   ./my-plugins/
   ├── debezium-connector-sqlserver
   │   ├── ...
   ```

2. Create and publish a custom image for running your Debezium connector:

   a. Create a new **Dockerfile** by using **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** 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-rhel7:1.5.0
      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 container image. For example, if you saved the **Dockerfile** that you created in the previous step as **debezium-container-for-sqlserver**, and if the **Dockerfile** is in the current directory, then you would run the following command:
      podman build -t debezium-container-for-sqlserver:latest .

   c. Push your custom image to your container registry, for example:
      podman push debezium-container-for-sqlserver:latest

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

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

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

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

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

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

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

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

   a. Display the Kafka Connect log output:

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

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

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

Results

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

5.4.3. 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.

5.4.3.1. Snapshot Metrics

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

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.

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

Attributes	Type	Description
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.

5.4.3.2. Streaming Metrics

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

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

Attributes	Type	Description
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

5.4.3.3. Schema History Metrics

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

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

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

5.4.4. 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.

Property	Default	Description
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**.

Property	Default	Description
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.
column.mask .hash.*hashAlgorithm* .with.salt.*salt*	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in the change event message values with a field value consisting of the hashed value using the algorithm ***hashAlgorithm*** and salt ***salt***. Based on the used hash function referential integrity is kept while data is pseudonymized. Supported hash functions are described in the {link-java7-standard-names}[MessageDigest section] of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash is automatically shortened to the length of the column. Multiple properties with different lengths can be used in a single configuration, although in each the length must be a positive integer or zero. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. Example: `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = dbo.orders.customerName, dbo.shipment.customerName` where **CzQMA0cB5K** is a randomly selected salt. Note: Depending on the ***hashAlgorithm*** used, the ***salt*** selected and the actual data set, the resulting masked data set may not be completely anonymized.

Property	Default	Description
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision, including: **adaptive** (the default) captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column’s type; or **connect** always represents time and timestamp values using Kafka Connect’s built-in representations for Time, Date, and Timestamp, which uses millisecond precision regardless of the database columns' precision. See temporal values.
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded with a key that contains the database name and a value that is a JSON structure that describes the schema update. This is independent of how the connector internally records database history. The default is **true**.
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*.

Property	Default	Description
column.mask.with.*length*.chars	*n/a*	An optional comma-separated list of regular expressions that match the fully-qualified names of character-based columns 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.

Property	Default	Description
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.

Property	Default	Description
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).
source.timestamp.mode	*commit*	String representing the criteria of the attached timestamp within the source record (ts_ms). **commit** will set the source timestamp to the instant where the record was committed in the database (default and current behavior). **processing** will set the source timestamp to the instant where the record was processed by Debezium. This option could be used when either we want to set the top level **ts_ms** value here or when we want to skip the query to extract the timestamp of that LSN.

Property	Default	Description
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.

Property	Default	Description
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.

Property	Default	Description
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".

Property	Default	Description
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. DEBEZIUM CONNECTOR FOR DB2

IMPORTANT

The Debezium Db2 connector 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.

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

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

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

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

The connector uses the abstract syntax notation (ASN) libraries that come as a standard part of Db2 LUW (Db2 for Linux, UNIX and Windows) and which you can add to Db2 zOS. To use ASN and hence this connector, you must have a license for the IBM InfoSphere Data Replication (IIDR) product. However, IIDR does not need to be installed.

The Db2 connector has been tested with Db2/Linux {linux-version}. It is expected that the connector would also work on Windows, AIX and zOS.

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

- Section 6.8, “Managing Debezium Db2 connectors”
- Section 6.9, “Updating schemas for Db2 tables in capture mode for Debezium connectors”

6.1. OVERVIEW OF DEBEZIUM DB2 CONNECTOR

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

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

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

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

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

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

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

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

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

6.2. HOW DEBEZIUM DB2 CONNECTORS WORK

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

Details are in the following topics:

- Section 6.2.1, “How Debezium Db2 connectors perform database snapshots”
- Section 6.2.2, “How Debezium Db2 connectors read change-data tables”
- Section 6.2.3, “Default names of Kafka topics that receive Debezium Db2 change event records”
- Section 6.2.4, “About the Debezium Db2 connector schema change topic”
- Section 6.2.5, “Debezium Db2 connector-generated events that represent transaction boundaries”

6.2.1. How Debezium Db2 connectors perform database snapshots

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

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

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

6.2.2. How Debezium Db2 connectors read change-data tables

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

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

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

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

By default, the Db2 connector writes change events for all insert, update, and delete operations on a single table to a single Kafka topic. The name of the Kafka topic has the following format:

databaseName.schemaName.tableName

databaseName
: The logical name of the connector as specified with the **database.server.name** connector configuration property.

schemaName
: The name of the schema in which the operation occurred.

tableName
: The name of the table in which the operation occurred.

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

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

To configure a Db2 connector to emit change events to differently-named Kafka topics, see the documentation for the topic routing transformation.

6.2.4. About the Debezium Db2 connector schema change topic

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

WARNING

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

Debezium emits a message to the schema change topic when:

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

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

```
{
  "schema": {
  ...
  },
  "payload": {
    "source": {
      "version": "1.2.4.Final",
      "connector": "db2",
      "name": "db2",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "testdb",
      "schema": "DB2INST1",
      "table": "CUSTOMERS",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "databaseName": "TESTDB", (1)
    "schemaName": "DB2INST1",
    "ddl": null, (2)
    "tableChanges": [ (3)
      {
        "type": "CREATE", (4)
        "id": "\"DB2INST1\".\"CUSTOMERS\"", (5)
        "table": { (6)
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ (7)
            "ID"
      ],
      "columns": [ (8)
        {
          "name": "ID",
          "jdbcType": 4,
          "nativeType": null,
          "typeName": "int identity",
          "typeExpression": "int identity",
          "charsetName": null,
          "length": 10,
          "scale": 0,
          "position": 1,
          "optional": false,
          "autoIncremented": false,
          "generated": false
        },
        {
          "name": "FIRST_NAME",
          "jdbcType": 12,
          "nativeType": null,
          "typeName": "varchar",
          "typeExpression": "varchar",
          "charsetName": null,
          "length": 255,
          "scale": null,
          "position": 2,
          "optional": false,
          "autoIncremented": false,
          "generated": false
        },
        {
          "name": "LAST_NAME",
          "jdbcType": 12,
          "nativeType": null,
          "typeName": "varchar",
          "typeExpression": "varchar",
          "charsetName": null,
          "length": 255,
          "scale": null,
          "position": 3,
          "optional": false,
          "autoIncremented": false,
          "generated": false
        },
        {
          "name": "EMAIL",
          "jdbcType": 12,
          "nativeType": null,
          "typeName": "varchar",
          "typeExpression": "varchar",
          "charsetName": null,
          "length": 255,
          "scale": null,
          "position": 4,
          "optional": false,
          "autoIncremented": false,
              "generated": false
            }
          ]
        }
      }
    ]
  }
}
```

Table 6.1. Descriptions of fields in messages emitted to the schema change topic

Item	Field name(s)	Description
1	**databaseName** **schemaName**	Identifies the database and the schema that contain the change.
2	**ddl**	Always **null** for the Db2 connector. For other connectors, this field contains the DDL responsible for the schema change. This DDL is not available to Db2 connectors.
3	**tableChanges**	An array of one or more items that contain the schema changes generated by a DDL command.
4	**type**	Describes the kind of change. The value is one of the following: ● **CREATE** - table created ● **ALTER** - table modified ● **DROP** - table deleted
5	**id**	Full identifier of the table that was created, altered, or dropped.
6	**table**	Represents table metadata after the applied change.
7	**primaryKeyColumnNames**	List of columns that compose the table's primary key.
8	**columns**	Metadata for each column in the changed table.

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

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

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

Debezium can generate events that represent transaction boundaries and that enrich change data event messages. For every transaction **BEGIN** and **END**, Debezium generates an event that contains the following fields:

- **status** - **BEGIN** or **END**
- **id** - string representation of unique transaction identifier
- **event_count** (for **END** events) - total number of events emitted by the transaction
- **data_collections** (for **END** events) - an array of pairs of **data_collection** and **event_count** that provides the number of events emitted by changes originating from the given data collection

Example

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

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

The connector emits transaction events to the ***database.server.name*****.transaction** topic.

Data change event enrichment

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

- **id** - string representation of unique transaction identifier
- **total_order** - absolute position of the event among all events generated by the transaction
- **data_collection_order** - the per-data collection position of the event among all events that were emitted by the transaction

Following is an example of a message:

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

6.3. DESCRIPTIONS OF DEBEZIUM DB2 CONNECTOR DATA CHANGE EVENTS

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

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

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

```
{
 "schema": { (1)
   ...
  },
 "payload": { (2)
```

```
    ...
  },
  "schema": { (3)
    ...
  },
  "payload": { (4)
    ...
  },
}
```

Table 6.2. Overview of change event basic content

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

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

WARNING

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

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

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

Details are in the following topics:

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

6.3.1. About keys in Debezium db2 change events

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

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

Example table

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

Example change event key

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

```
{
    "schema": { (1)
        "type": "struct",
        "fields": [ (2)
```

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

Table 6.3. Description of change event key

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

6.3.2. About values in Debezium Db2 change events

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

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

Example table

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

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

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

create events

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

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

```
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "ID"
      },
      {
        "type": "string",
        "optional": false,
        "field": "FIRST_NAME"
      },
      {
        "type": "string",
        "optional": false,
        "field": "LAST_NAME"
      },
      {
        "type": "string",
        "optional": false,
        "field": "EMAIL"
      }
    ],
    "optional": true,
    "name": "mydatabase.MYSCHEMA.CUSTOMERS.Value",
    "field": "after"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "version"
      },
      {
        "type": "string",
        "optional": false,
        "field": "connector"
      },
      {
        "type": "string",
        "optional": false,
        "field": "name"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "ts_ms"
      },
      {
        "type": "boolean",
        "optional": true,
        "default": false,
        "field": "snapshot"
      },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": false,
            "field": "schema"
          },
          {
            "type": "string",
            "optional": false,
            "field": "table"
          },
          {
            "type": "string",
            "optional": true,
            "field": "change_lsn"
          },
          {
            "type": "string",
            "optional": true,
            "field": "commit_lsn"
          },
        ],
        "optional": false,
        "name": "io.debezium.connector.db2.Source",  3
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mydatabase.MYSCHEMA.CUSTOMERS.Envelope"  4
  },
  "payload": {  5
    "before": null,  6
    "after": {  7
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "source": {  8
      "version": "1.2.4.Final",
      "connector": "db2",
        "name": "myconnector",
        "ts_ms": 1559729468470,
        "snapshot": false,
        "db": "mydatabase",
        "schema": "MYSCHEMA",
        "table": "CUSTOMERS",
        "change_lsn": "00000027:00000758:0003",
        "commit_lsn": "00000027:00000758:0005",
      },
      "op": "c", (9)
      "ts_ms": 1559729471739 (10)
    }
  }
```

Table 6.4. Descriptions of *create* event value fields

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

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

update events

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

```
{
  "schema": { ... },
  "payload": {
    "before": {  ①
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "after": {  ②
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "noreply@example.org"
    },
    "source": {  ③
      "version": "1.2.4.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
    },
    "op": "u",  ④
    "ts_ms": 1559729998706  ⑤
  }
}
```

Table 6.5. Descriptions of *update* event value fields

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

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

NOTE

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

delete events

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

```
{
  "schema": { ... },
  },
  "payload": {
    "before": {  (1)
```

```
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "noreply@example.org"
    },
    "after": null, 2
    "source": { 3
      "version": "1.2.4.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559730445243,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000db0:0005",
      "commit_lsn": "00000027:00000db0:0007"
    },
    "op": "d", 4
    "ts_ms": 1559730450205 5
  }
}
```

Table 6.6. Descriptions of *delete* event value fields

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

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

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

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

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

6.4. HOW DEBEZIUM DB2 CONNECTORS MAP DATA TYPES

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

The Db2 connector represents changes to rows with events that are structured like the table in which the row exists. The event contains a field for each column value. How that value is represented in the event depends on the Db2 data type of the column. This section describes these mappings.

Details are in the following sections:

- Basic types
- Temporal types
- Timestamp types
- Decimal types

Basic types

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

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

Table 6.7. Mappings for Db2 basic data types

Db2 data type	Literal type (schema type)	Semantic type (schema name)
BOOLEAN	**BOOLEAN**	n/a
BIGINT	**INT64**	n/a
BINARY	**BYTES**	n/a
BLOB	**BYTES**	n/a
CHAR[(N)]	**STRING**	n/a
CLOB	**STRING**	n/a
DATE	**INT32**	**io.debezium.time.Date** String representation of a timestamp without timezone information
DECFLOAT	**BYTES**	**org.apache.kafka.connect.data.Decimal**
DECIMAL	**BYTES**	**org.apache.kafka.connect.data.Decimal**
DBCLOB	**STRING**	n/a
DOUBLE	**FLOAT64**	n/a
INTEGER	**INT32**	n/a
REAL	**FLOAT32**	n/a
SMALLINT	**INT16**	n/a
TIME	**INT32**	**io.debezium.time.Time**+ String representation of a time without timezone information

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

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

Temporal types

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

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

time.precision.mode=adaptive

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

Table 6.8. Mappings when **time.precision.mode** is **adaptive**

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

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

time.precision.mode=connect

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

Table 6.9. Mappings when**time.precision.mode** is**connect**

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

Timestamp types

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

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

Table 6.10. Decimal types

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

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

6.5. SETTING UP DB2 TO RUN A DEBEZIUM CONNECTOR

A database administrator must put tables into capture mode before you can run a Debezium Db2 connector to capture changes that are committed to a Db2 database. To put tables into capture mode, Debezium provides a set of user-defined functions (UDFs) for your convenience. The procedure here shows how to install and run these management UDFs. Alternatively, you can run Db2 control commands to put tables into capture mode.

This procedure assumes that you are logged in as the **db2instl** user, which is the default instance and user name when using the Db2 docker container image.

Prerequisites

- On the machine on which Db2 is running, the content in **debezium-connector-db2/src/test/docker/db2-cdc-docker** is available in the **$HOME/asncdctools/src** directory.

Procedure

Procedure

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

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

   ```
   ./bldrtn asncdc
   ```

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

   ```
   db2 start db DB_NAME
   ```

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

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

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

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

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

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

   b. Restart the database:

      ```
      db2 restart db DB_NAME
      ```

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

   ```
   db2 connect to DB_NAME
   ```

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

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

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

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

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

8. Create the ASN control tables:

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

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

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

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

10. Start the ASN agent:

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

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

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

12. Reinitialize the ASN service:

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

Additional resource

Reference table for Debezium Db2 management UDFs

6.6. DEPLOYING DEBEZIUM DB2 CONNECTORS

To deploy a Debezium Db2 connector, install the Debezium Db2 connector archive, configure the connector, and start the connector by adding its configuration to Kafka Connect. Details are in the following topics:

- Section 6.6.1, “Steps for installing Debezium Db2 connectors”
- Section 6.6.2, “Debezium db2 connector configuration example”
- Section 6.6.3, “Adding Debezium Db2 connector configuration to Kafka Connect”
- Section 6.6.4, “Description of Debezium Db2 connector configuration properties”

6.6.1. Steps for installing Debezium Db2 connectors

To install the Db2 connector, follow the procedures in Installing Debezium on OpenShift. The main steps are:

1. Set up Db2 to run a Debezium connector . This enables Db2 replication to expose change-data for tables that are in capture mode.
2. Use Red Hat AMQ Streams to set up Apache Kafka and Kafka Connect on OpenShift. AMQ Streams offers operators and images that bring Kafka to OpenShift.
3. Download the Debezium Db2 connector.

4. Extract the files into your Kafka Connect environment.

5. Add the plug-in's parent directory to your Kafka Connect **plugin.path**, for example:

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

   The above example assumes that you extracted the Debezium Db2 connector to the **/kafka/connect/Debezium-connector-db2** path.

6. Restart your Kafka Connect process to ensure that the new JAR files are picked up.

6.6.2. Debezium db2 connector configuration example

Following is an example of the configuration for a Db2 connector that connects to a Db2 server on port 50000 at 192.168.99.100, whose logical name is **fullfillment**. Typically, you configure a Debezium Db2 connector in a **.yaml** file using the configuration properties available for the connector.

You can 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.

```
apiVersion: kafka.strimzi.io/v1beta1
  kind: KafkaConnector
  metadata:
    name: inventory-connector  (1)
    labels: strimzi.io/cluster: my-connect-cluster
  spec:
    class: io.debezium.connector.db2.Db2Connector
    tasksMax: 1  (2)
    config:  (3)
      database.hostname: 192.168.99.100  (4)
      database.port: 50000
      database.user: db2inst1
      database.password: Password!
      database.dbname: mydatabase
      database.server.name: fullfillment  (5)
      database.whitelist: public.inventory  (6)
```

Table 6.11. Descriptions of connector configuration settings

Item	Description
1	The name of the connector.
2	Only one task should operate at any one time.
3	The connector's configuration.
4	The database host, which is the address of the Db2 instance.
5	The logical name of the Db2 instance/cluster, which forms a namespace and is used in the names of the Kafka topics to which the connector writes, the names of Kafka Connect schemas, and the namespaces of the corresponding Avro schema when the Avro Connector is used.

Item	Description
6	Changes in only the **public.inventory** database are captured.

See the complete list of connector properties that you can specify in these configurations.

You can send this configuration with a **POST** command to a running Kafka Connect service. The service records the configuration and starts one connector task that connects to the Db2 database, reads change-data tables for tables in capture mode, and streams change event records to Kafka topics.

6.6.3. Adding Debezium Db2 connector configuration to Kafka Connect

You can use a provided Debezium container to deploy a Debezium Db2 connector. In this procedure, you build a custom Kafka Connect container image for Debezium, configure the Debezium connector as needed, and then add your connector configuration to your Kafka Connect environment.

Prerequisites

- Podman or Docker is installed and you have sufficient rights to create and manage containers.
- You installed the Debezium Db2 connector archive.

Procedure

1. Extract the Debezium Db2 connector archive to create a directory structure for the connector plug-in, for example:

```
tree ./my-plugins/
./my-plugins/
├── debezium-connector-db2
│   ├── ...
```

2. Create and publish a custom image for running your Debezium connector:

   a. Create a new **Dockerfile** by using **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** 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-rhel7:1.5.0
   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 container image. For example, if you saved the **Dockerfile** that you created in the previous step as **debezium-container-for-db2**, and if the **Dockerfile** is in the current directory, then you would run the following command:
   podman build -t debezium-container-for-db2:latest .

   c. Push your custom image to your container registry, for example:
   podman push debezium-container-for-db2:latest

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

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: my-connect-cluster
spec:
  #...
  image: debezium-container-for-db2
```

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

3. Create a **KafkaConnector** custom resource that defines your Debezium Db2 connector instance. See the connector configuration example.

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

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

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

   a. Display the Kafka Connect log output:

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

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

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

Results

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

6.6.4. Description of Debezium Db2 connector configuration properties

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

- Required configuration properties

- Advanced configuration properties
- Pass-through configuration properties

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

Table 6.12. Required connector configuration properties

Property	Default	Description
name		Unique name for the connector. Attempting to register again with the same name will fail. This property is required by all Kafka Connect connectors.
connector.class		The name of the Java class for the connector. Always use a value of **io.debezium.connector.db2.Db2Connector** for the Db2 connector.
tasks.max	**1**	The maximum number of tasks that should be created for this connector. The Db2 connector always uses a single task and therefore does not use this value, so the default is always acceptable.
database.hostname		IP address or hostname of the Db2 database server.
database.port	**50000**	Integer port number of the Db2 database server.
database.user		Name of the Db2 database user for connecting to the Db2 database server.
database.password		Password to use when connecting to the Db2 database server.
database.dbname		The name of the Db2 database from which to stream the changes
database.server.name		Logical name that identifies and provides a namespace for the particular Db2 database server that hosts the database for which Debezium is capturing changes. Only alphanumeric characters and underscores should be used in the database server logical name. The logical name should be unique across all other connectors, since it is used as a topic name prefix for all Kafka topics that receive records from this connector.

Property	Default	Description
database.history .kafka.topic		The full name of the Kafka topic where the connector stores the database schema history.
database.history .kafka.bootstrap.servers		A list of host/port pairs that the connector uses to establish an initial connection to the Kafka cluster. This connection is used for retrieving database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Debezium Kafka Connect process.
table.whitelist		An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you want the connector to capture. Any table not included in the include list does not have its changes captured. Each identifier is of the form *schemaName*.*tableName*. By default, the connector captures changes in every non-system table. Do not also set the **table.blacklist** property.
table.blacklist		An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you do not want the connector to capture. The connector captures changes in each non-system table that is not included in the exclude list. Each identifier is of the form *schemaName*.*tableName*. Do not also set the **table.whitelist** property.
column.blacklist	*empty string*	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to exclude from change event values. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*. Primary key columns are always included in the event's key, even if they are excluded from the value.

Property	Default	Description
column.mask .hash.*hashAlgorithm* .with.salt.*salt*	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be pseudonyms in change event values. A pseudonym is a field value that consists of the hashed value obtained by applying the ***hashAlgorithm*** algorithm and the ***salt*** salt that you specify in the property name. Based on the hash algorithm applied, referential integrity is kept while data is masked. Supported hash algorithms are described in the {link-java7-standard-names}[MessageDigest section] of the Java Cryptography Architecture Standard Algorithm Name Documentation. The hash value is automatically shortened to the length of the column. You can specify multiple instances of this property with different algorthims and salts. Fully-qualified names for columns are of the form *schemaName*.*tableName*.*columnName*. For example: **column.mask.hash.SHA-256.with.salt.CzQMA0cB5K =** + **inventory.orders.customerName, inventory.shipment.customerName** where **CzQMA0cB5K** is a randomly selected salt. Depending on the ***hashAlgorithm*** used, the ***salt*** selected, and the actual data set, the field value may not be completely masked.
time.precision.mode	**adaptive**	Time, date, and timestamps can be represented with different kinds of precision: **adaptive** captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column’s type. **connect** always represents time and timestamp values by using Kafka Connect’s built-in representations for **Time**, **Date**, and **Timestamp**, which uses millisecond precision regardless of the database columns' precision. See temporal values.

Property	Default	Description
tombstones.on.delete	**true**	Controls whether a tombstone event should be generated after a *delete* event. **true** - delete operations are represented by a *delete* event and a subsequent tombstone event. **false** - only a *delete* event is sent. After a *delete* operation, emitting a tombstone event enables Kafka to delete all change event records that have the same key as the deleted row.
include.schema.changes	**true**	Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded with a key that contains the database name and a value that is a JSON structure that describes the schema update. This is independent of how the connector internally records database history.
column.truncate.to.*length*.chars	*n/a*	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form *schemaName.tableName.columnName*. In change event records, values in these columns are truncated if they are longer than the number of characters specified by *length* in the property name. You can specify multiple properties with different lengths in a single configuration. Length must be a positive integer, for example, **column.truncate.to.20.chars**.

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

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

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

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

Table 6.13. Advanced connector configuration properties

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

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

Property	Default	Description
poll.interval.ms	**1000**	Positive integer value that specifies the number of milliseconds the connector should wait for new change events to appear before it starts processing a batch of events. Defaults to 1000 milliseconds, or 1 second.
max.queue.size	**8192**	Positive integer value for the maximum size of the blocking queue. The connector places change events that it reads from the database log into the blocking queue before writing them to Kafka. This queue can provide backpressure for reading change-data tables when, for example, writing records to Kafka is slower than it should be or Kafka is not available. Events that appear in the queue are not included in the offsets that are periodically recorded by the connector. The **max.queue.size** value should always be larger than the value of the **max.batch.size** connector configuration property.
max.batch.size	**2048**	Positive integer value that specifies the maximum size of each batch of events that the connector processes.
heartbeat.interval.ms	**0**	Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages. Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages. Heartbeat messages are useful when there are many updates in a database that is being tracked but only a tiny number of updates are in tables that are in capture mode. In this situation, the connector reads from the database transaction log as usual but rarely emits change records to Kafka. This means that the connector has few opportunities to send the latest offset to Kafka. Sending heartbeat messages enables the connector to send the latest offset to Kafka.

Property	Default	Description
heartbeat.topics.prefix	**__debezium-heartbeat**	Specifies the prefix for the name of the topic to which the connector sends heartbeat messages. The format for this topic name is **<heartbeat.topics.prefix>.<server.name>**.
snapshot.delay.ms		An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
snapshot.fetch.size	**2000**	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
snapshot.lock.timeout.ms	**10000**	Positive integer value that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this interval, the snapshot fails. How the connector performs snapshots provides details. Other possible settings are: **0** - The connector immediately fails when it cannot obtain a lock. **-1** - The connector waits infinitely.

Property	Default	Description
snapshot.select .statement.overrides		Controls which table rows are included in snapshots. This property affects snapshots only. It does not affect events that the connector reads from the log. Specify a comma-separated list of fully-qualified table names in the form *schemaName.tableName*. For each table that you specify, also specify another configuration property: **snapshot.select.statement.overrides.*SCHEMA_NAME.TABLE_NAME***. For example: **snapshot.select.statement.overrides.customers.orders**. Set this property to a **SELECT** statement that obtains only the rows that you want in the snapshot. When the connector performs a snapshot, it executes this **SELECT** statement to retrieve data from that table. A possible use case for setting these properties is large, append-only tables. You can specify a **SELECT** statement that sets a specific point for where to start a snapshot, or where to resume a snapshot if a previous snapshot was interrupted.
sanitize.field.names	**true** if connector configuration sets the **key.converter** or **value.converter** property to the Avro converter. **false** if not.	Indicates whether field names are sanitized to adhere to Avro naming requirements.
provide.transaction .metadata	**false**	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify **true** if you want the connector to do this. See Transaction metadata for details.

Pass-through connector configuration properties

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

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

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

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

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

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

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

6.7. MONITORING DEBEZIUM DB2 CONNECTOR PERFORMANCE

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

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

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

Snapshot metrics

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

Attributes	Type	Description
LastEvent	**string**	The last snapshot event that the connector has read.
MilliSecondsSinceLastEvent	**long**	The number of milliseconds since the connector has read and processed the most recent event.

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

Attributes	Type	Description
RowsScanned	**Map<String, Long>**	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.

Streaming metrics

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

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

Attributes	Type	Description
Connected	**boolean**	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	**long**	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incoporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	**long**	The number of processed transactions that were committed.
SourceEventPosition	**Map<String, String>**	The coordinates of the last received event.
LastTransactionId	**string**	Transaction identifier of the last processed transaction.

Schema history metrics

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

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

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

6.8. MANAGING DEBEZIUM DB2 CONNECTORS

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

Table 6.14. Descriptions of Debezium management UDFs

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

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

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

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

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

WARNING

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

There are generally two procedures for updating table schemas:

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

Each approach has advantages and disadvantages.

6.9.1. Performing offline schema updates for Debezium Db2 connectors

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

Prerequisites

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

Procedure

1. Suspend the application that updates the database.

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

6.9.2. Performing online schema updates for Debezium Db2 connectors

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

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

Prerequisites

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

Procedure when adding a column to the end of a table

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

Procedure when adding a column to the middle of a table

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

CHAPTER 7. MONITORING DEBEZIUM

You can use the JMX metrics provided by Zookeeper and Kafka to monitor Debezium. To use these metrics, you must enable them when you start the Zookeeper, Kafka, and Kafka Connect services. Enabling JMX involves setting the correct environment variables.

NOTE

If you are running multiple services on the same machine, be sure to use distinct JMX ports for each service.

7.1. MONITORING DEBEZIUM ON RHEL

7.1.1. 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**.

7.1.2. 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**

7.1.3. 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**

7.2. 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 8. DEBEZIUM LOGGING

Debezium has extensive logging built into its connectors, and you can change the logging configuration to control which of these log statements appear in the logs and where those logs are sent. Debezium (as well as Kafka, Kafka Connect, and Zookeeper) use the Log4j logging framework for Java.

By default, the connectors produce a fair amount of useful information when they start up, but then produce very few logs when the connector is keeping up with the source databases. This is often sufficient when the connector is operating normally, but may not be enough when the connector is behaving unexpectedly. In such cases, you can change the logging level so that the connector generates much more verbose log messages describing what the connector is doing and what it is not doing.

8.1. 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.

8.2. 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
...
```

Table 8.1. Descriptions of log settings

Item	Description
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.

8.3. 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.

8.3.1. 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.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
...
```

Table 8.2. Descriptions of log settings

Item	Description
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
...
```

8.3.2. 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**).

8.4. 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 9. CONFIGURING DEBEZIUM CONNECTORS FOR YOUR APPLICATION

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

- **Topic router** SMT re-routes data change event records to topics that you specify.
- **Content-based router SMT** evaluates data change event record content and re-routes event records to particular topics according to content.
- **Filter** SMT uses an expression that you specify to evaluate data change event records. The connector streams only those events that evaluate to true.
- **Event flattening** SMT flattens the complex structure of a data change event record into the simplified format that might be required by some Kafka consumers.
- **Avro serialization** for PostgreSQL, MongoDB, or SQL Server connectors makes it easier for change event record consumers to adapt to a changing record schema.
- **Outbox event router** SMT provides support for the outbox pattern.
- **CloudEvents converter** enables a Debezium connector to emit change event records that conform to the CloudEvents specification.

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

9.1.1. 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.

9.1.2. 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.

9.1.3. 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
...
```

9.1.4. Options for configuring topic routing transformation

Option	Default	Description
topic.regex		Specifies a regular expression that the transformation applies to each change event record to determine if it should be routed to a particular topic.
topic.replacement		Specifies a regular expression that represents the destination topic name. The transformation routes each matching record to the topic identified by this expression. This expression can refer to groups captured by the regular expression that you specify for **topic.regex**. To refer to a group, specify **$1**, **$2**, and so on.
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.

Option	Default	Description
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.

9.2. ROUTING CHANGE EVENT RECORDS TO TOPICS ACCORDING TO EVENT CONTENT

By default, Debezium streams all of the change events that it reads from a table to a single static topic. However, there might be situations in which you might want to reroute selected events to other topics, based on the event content. The process of routing messages based on their content is described in the Content-based routing messaging pattern. To apply this pattern in Debezium, you use the content-based routing single message transform (SMT) to write expressions that are evaluated for each event. Depending how an event is evaluated, the SMT either routes the event message to the original destination topic, or reroutes it to the topic that you specify in the expression.

IMPORTANT

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

While it is possible to use Java to create a custom SMT to encode routing logic, using a custom-coded SMT has its drawbacks. For example:

- It is necessary to compile the transformation up front and deploy it to Kafka Connect.
- Every change needs code recompilation and redeployment, leading to inflexible operations.

The content-based routing SMT supports scripting languages that integrate with JSR 223 (Scripting for the Java™ Platform).

Debezium does not come with any implementations of the JSR 223 API. To use an expression language with Debezium, you must download the JSR 223 script engine implementation for the language, and add to your Debezium connector plug-in directories, along any other JAR files used by the language implementation. For example, for Groovy 3, you can download its JSR 223 implementation from https://groovy-lang.org/. The JSR 223 implementation for GraalVM JavaScript is available at https://github.com/graalvm/graaljs.

9.2.1. Setting up the Debezium content-based-routing SMT

For security reasons, the content-based routing SMT is not included with the Debezium connector archives. Instead, it is provided in a separate artifact, **debezium-scripting-1.2.4.Final.tar.gz**. To use the content-based routing SMT with a Debezium connector plug-in, you must explicitly add the SMT artifact to your Kafka Connect environment.

IMPORTANT

After the routing SMT is present in a Kafka Connect instance, any user who is allowed to add a connector to the instance can run scripting expressions. To ensure that scripting expressions can be run only by authorized users, be sure to secure the Kafka Connect instance and its configuration interface before you add the routing SMT.

Procedure

1. Download the Debezium scripting SMT archive (**debezium-scripting-1.2.4.Final.tar.gz**) from https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=red.hat.integration&downloadType=distributions.
2. Extract the contents of the archive into the Debezium plug-in directories of your Kafka Connect environment.
3. Obtain a JSR-223 script engine implementation and add its contents to the Debezium plug-in directories of your Kafka Connect environment.
4. Restart the Kafka Connect process to pick up the new JAR files.

9.2.2. Example: Debezium basic content-based routing configuration

To configure a Debezium connector to route change event records based on the event content, you configure the **ContentBasedRouter** SMT in the Kafka Connect configuration for the connector.

Configuration of the content-based routing SMT requires you to specify a regular expression that defines the filtering criteria. In the configuration, you create a regular expression that defines routing criteria. The expression defines a pattern for evaluating event records. It also specifies the name of a destination topic where events that match the pattern are routed. The pattern that you specify might designate an event type, such as a table insert, update, or delete operation. You might also define a pattern that matches a value in a specific column or row.

For example, to reroute all update (**u**) records to an **updates** topic, you might add the following configuration to your connector configuration:

```
...
transforms=route
transforms.route.type=io.debezium.transforms.ContentBasedRouter
transforms.route.language=jsr223.groovy
transforms.route.topic.expression=value.op == 'u' ? 'updates' : null
...
```

The preceding example specifies the use of the **Groovy** expression language.

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

9.2.3. Variables for use in Debezium content-based routing expressions

Debezium binds certain variables into the evaluation context for the SMT. When you create expressions to specify conditions to control the routing destination, the SMT can look up and interpret the values of these variables to evaluate conditions in an expression.

The following table lists the variables that Debezium binds into the evaluation context for the content-based routing SMT:

Table 9.1. Content-based routing expression variables

Name	Description	Type
key	A key of the message.	**org.apache.kafka.connect.data .Struct**
value	A value of the message.	**org.apache.kafka.connect.data .Struct**
keySchema	Schema of the message key.	**org.apache.kafka.connect.data .Schema**
valueSchema	Schema of the message value.	**org.apache.kafka.connect.data .Schema**
topic	Name of the target topic.	String
headers	A Java map of message headers. The key field is the header name. The **headers** variable exposes the following properties: • **value** (of type **Object**) • **schema** (of type **org.apache.kafka .connect.data.Schema**)	**java.util.Map<String, io.debezium .transforms.scripting .RecordHeader>**

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

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

9.2.4. Configuration of content-based routing conditions for other scripting languages

The way that you express content-based routing conditions depends on the scripting language that you use. For example, as shown in this basic Debezium content-based routing SMT example , when you use **Groovy** as the expression language, the following expression reroutes all update (**u**) records to the **updates** topic, while routing other records to the default topic:

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

■

Other languages use different methods to express the same condition.

TIP

The Debezium MongoDB connector emits the **after** and **patch** fields as serialized JSON documents rather than as structures. To use the ContentBasedRouting SMT with the MongoDB connector, you must first unwind the fields by applying the **ExtractNewDocumentState** SMT.

You could also take the approach of using a JSON parser within the expression. For example, if you use Groovy as the expression language, add the **groovy-json** artifact to the classpath, and then add an expression such as **(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'**.

Javascript

When you use JavaScript as the expression language, you can call the **Struct#get()** method to specify the content-based routing condition, as in the following example:

```
value.get('op') == 'u' ? 'updates' : null
```

Javascript with Graal.js

When you create coentent-based routing conditions by using JavaScript with Graal.js, you use an approach that is similar to the one use with Groovy. For example:

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

9.2.5. Options for configuring the content-based routing transformation

Property	Default	Description
topic.regex		An optional regular expression that evaluates the name of the destination topic for an event to determine whether to apply the condition logic. If the name of the destination topic matches the value in **topic.regex**, the transformation applies the condition logic before it passes the event to the topic. If the name of the topic does not match the value in **topic.regex**, the SMT passes the event to the topic unmodified.
language		The language in which the expression is written. Must begin with **jsr223.**, for example, **jsr223.groovy**, or **jsr223.graal.js**. Debezium supports bootstrapping through the JSR 223 API ("Scripting for the Java ™ Platform") only.

Property	Default	Description
topic.expression		The expression to be evaluated for every message. Must evaluate to a **String** value where a result of non-null reroutes the message to a new topic, and a **null** value routes the message to the default topic.
null.handling.mode	**keep**	Specifies how the transformation handles **null** (tombstone) messages. You can specify one of the following options: **keep** (Default) Pass the messages through. **drop** Remove the messages completely. **evaluate** Apply the condition logic to the messages.

9.3. FILTERING DEBEZIUM CHANGE EVENT RECORDS

By default, Debezium delivers every data change event that it receives to the Kafka broker. However, in many cases, you might be interested in only a subset of the events emitted by the producer. To enable you to process only the records that are relevant to you, Debezium provides the *filter* simple message transform (SMT).

IMPORTANT

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

While it is possible to use Java to create a custom SMT to encode filtering logic, using a custom-coded SMT has its drawbacks. For example:

- It is necessary to compile the transformation up front and deploy it to Kafka Connect.
- Every change needs code recompilation and redeployment, leading to inflexible operations.

The filter SMT supports scripting languages that integrate with JSR 223 (Scripting for the Java™ Platform).

Debezium does not come with any implementations of the JSR 223 API. To use an expression language with Debezium, you must download the JSR 223 script engine implementation for the language, and add to your Debezium connector plug-in directories, along any other JAR files used by the language implementation. For example, for Groovy 3, you can download its JSR 223 implementation from https://groovy-lang.org/. The JSR223 implementation for GraalVM JavaScript is available at https://github.com/graalvm/graaljs.

9.3.1. Setting up the Debezium filter SMT

For security reasons, the filter SMT is not included with the Debezium connector archives. Instead, it is provided in a separate artifact, **debezium-scripting-1.2.4.Final.tar.gz**. To use the filter SMT with a Debezium connector plug-in, you must explicitly add the SMT artifact to your Kafka Connect environment.

IMPORTANT

After the filter SMT is present in a Kafka Connect instance, any user who is allowed to add a connector to the instance can run scripting expressions. To ensure that scripting expressions can be run only by authorized users, be sure to secure the Kafka Connect instance and its configuration interface before you add the filter SMT.

Procedure

1. Download the Debezium scripting SMT archive (**debezium-scripting-1.2.4.Final.tar.gz**) from https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?product=red.hat.integration&downloadType=distributions.
2. Extract the contents of the archive into the Debezium plug-in directories of your Kafka Connect environment.
3. Obtain a JSR-223 script engine implementation and add its contents to the Debezium plug-in directories of your Kafka Connect environment.
4. Restart the Kafka Connect process to pick up the new JAR files.

9.3.2. Example: Debezium basic filter SMT configuration

You configure the filter transformation in the Debezium connector's Kafka Connect configuration. In the configuration, you specify the events that you are interested in by defining filter conditions that are based on business rules. As the filter SMT processes the event stream, it evaluates each event against the configured filter conditions. Only events that meet the criteria of the filter conditions are passed to the broker.

To configure a Debezium connector to filter change event records, configure the **Filter** SMT in the Kafka Connect configuration for the Debezium connector. Configuration of the filter SMT requires you to specify a regular expression that defines the filtering criteria.

For example, you might add the following configuration in your connector configuration.

```
...
transforms=filter
transforms.filter.type=io.debezium.transforms.Filter
transforms.filter.language=jsr223.groovy
transforms.filter.condition=value.op == 'u' && value.before.id == 2
...
```

The preceding example specifies the use of the **Groovy** expression language. The regular expression **value.op == 'u' && value.before.id == 2** removes all messages, except those that represent update (**u**) records with **id** values that are equal to **2**.

9.3.3. Variables for use in filter expressions

Debezium binds certain variables into the evaluation context for the filter SMT. When you create expressions to specify filter conditions, you can use the variables that Debezium binds into the evaluation context. By binding variables, Debezium enables the SMT to look up and interpret their values as it evaluates the conditions in an expression.

The following table lists the variables that Debezium binds into the evaluation context for the filter SMT:

Table 9.2. Filter expression variables

Name	Description	Type
key	A key of the message.	**org.apache.kafka.connect.data .Struct**
value	A value of the message.	**org.apache.kafka.connect.data .Struct**
keySchema	Schema of the message key.	**org.apache.kafka.connect.data .Schema**
valueSchema	Schema of the message value.	**org.apache.kafka.connect.data .Schema**
topic	Name of the target topic.	String
headers	A Java map of message headers. The key field is the header name. The **headers** variable exposes the following properties: • **value** (of type **Object**) • **schema** (of type **org.apache.kafka .connect.data.Schema**)	**java.util.Map<String, io.debezium .transforms.scripting .RecordHeader>**

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

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

9.3.4. Filter condition configuration for other scripting languages

The way that you express filtering conditions depends on the scripting language that you use.

For example, as shown in this basic filter SMT example, when you use **Groovy** as the expression language, the following expression removes all messages, except for update records that have **id** values set to **2**:

```
value.op == 'u' && value.before.id == 2
```

Other languages use different methods to express the same condition.

TIP

The Debezium MongoDB connector emits the **after** and **patch** fields as serialized JSON documents rather than as structures. To use the filter SMT with the MongoDB connector, you must first unwind the fields by applying the **ExtractNewDocumentState** SMT.

You could also take the approach of using a JSON parser within the expression. For example, if you use Groovy as the expression language, add the **groovy-json** artifact to the classpath, and then add an expression such as **(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'**.

Javascript

If you use JavaScript as the expression language, you can call the **Struct#get()** method to specify the filtering condition, as in the following example:

```
value.get('op') == 'u' && value.get('before').get('id') == 2
```

Javascript with Graal.js

If you use JavaScript with Graal.js to define filtering conditions, you use an approach that is similar to the one that you use with Groovy. For example:

```
value.op == 'u' && value.before.id == 2
```

9.3.5. Options for configuring filter transformation

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

Table 9.3. filter SMT configuration options

Property	Default	Description
topic.regex		An optional regular expression that evaluates the name of the destination topic for an event to determine whether to apply filtering logic. If the name of the destination topic matches the value in **topic.regex**, the transformation applies the filter logic before it passes the event to the topic. If the name of the topic does not match the value in **topic.regex**, the SMT passes the event to the topic unmodified.

Property	Default	Description
language		The language in which the expression is written. Must begin with **jsr223.**, for example, **jsr223.groovy**, or **jsr223.graal.js**. Debezium supports bootstrapping through the JSR 223 API ("Scripting for the Java ™ Platform") only.
condition		The expression to be evaluated for every message. Must evaluate to a Boolean value where a result of **true** keeps the message, and a result of **false** removes it.
null.handling.mode	**keep**	Specifies how the transformation handles **null** (tombstone) messages. You can specify one of the following options: **keep** (Default) Pass the messages through. **drop** Remove the messages completely. **evaluate** Apply the filter condition to the messages.

9.4. 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 9.4.1, “Description of Debezium change event structure”
- Section 9.4.2, “Behavior of Debezium **ExtractNewRecordState** transformation”
- Section 9.4.3, “Configuration of **ExtractNewRecordState** transformation”
- Section 9.4.4, “Example of adding metadata to the Kafka record”
- Section 9.4.5, “Options for configuring **ExtractNewRecordState** transformation”

9.4.1. 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.

9.4.2. 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 **ExtractNewRecordState** 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 **ExtractNewRecordState** SMT to keep the tombstone record in the stream.

9.4.3. 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.

9.4.4. 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**.

9.4.5. Options for configuring ExtractNewRecordState transformation

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

Option	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.

Option	Default	Description
route.by.field		To use row data to determine the topic to route the record to, set this option to an **after** field attribute. The SMT routes the record to the topic whose name matches the value of the specified **after** field attribute. For a**DELETE** operation, set this option to a **before** field attribute. For example, configuration of **route.by.field=destination** routes records to the topic whose name is the value of **after.destination**. The default behavior is that a Debezium connector sends each change event record to a topic whose name is formed from the name of the database and the name of the table in which the change was made. If you are configuring the **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.

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

9.5. CONFIGURING DEBEZIUM CONNECTORS TO USE 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 .

9.5.1. About the 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.

9.5.2. Overview of deploying a Debezium connector that uses Avro serialization

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.GetOrCreateIdStrategy
value.converter=io.apicurio.registry.utils.converter.AvroConverter
value.converter.apicurio.registry.url=http://apicurio:8080/api
value.converter.apicurio.registry.global-
id=io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
```

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

9.5.3. Deploying connectors that use Avro in Debezium containers

In your environment, you might want to use a provided Debezium container to deploy Debezium connectors that use Avro serialization. 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 Docker installed and sufficient rights to create and manage containers.
- You downloaded the Debezium connector plug-in(s) that you want to deploy with Avro serialization.

Procedure

1. Deploy an instance of Service Registry. See Getting Started with Service Registry, Installing Service Registry from the OpenShift OperatorHub, which provides instructions for:
   - Installing AMQ Streams
   - Setting up AMQ Streams storage
   - Installing Service Registry
2. Extract the Debezium connector 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-rhel7:1.5.0** 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-rhel7:1.5.0
      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:

      - 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.GetOrCreateIdStrategy
    value.converter: io.apicurio.registry.utils.converter.AvroConverter
    value.converter.apicurio.registry.url: http://apicurio:8080/api
    value.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
```

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

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

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

   a. Display the Kafka Connect log output:

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

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

```
...
2020-02-21 17:57:30,801 INFO Starting snapshot for jdbc:mysql://mysql:3306/?
```

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

Taking the snapshot involves a number of steps:

```
...
2020-02-21 17:57:30,822 INFO Step 0: disabling autocommit, enabling repeatable read
transactions, and setting lock wait timeout to 10
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,836 INFO Step 1: flush and obtain global read lock to prevent
writes to database (io.debezium.connector.mysql.SnapshotReader) [debezium-
mysqlconnector-dbserver1-snapshot]
2020-02-21 17:57:30,839 INFO Step 2: start transaction with consistent snapshot
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,840 INFO Step 3: read binlog position of MySQL 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]
...
```

9.5.4. About Avro name requirements

As stated in the Avro documentation, names must adhere to the following rules:

- Start with **[A-Za-z_]**
- Subsequently contains only **[A-Za-z0-9_]** characters

Debezium uses the column's name as the basis for the corresponding Avro field. This can lead to problems during serialization if the column name does not also adhere to the Avro naming rules. Each Debezium connector provides a configuration property, **sanitize.field.names** that you can set to **true** if you have columns that do not adhere to Avro rules for names. Setting **sanitize.field.names** to **true** allows serialization of non-conformant fields without having to actually modify your schema.

9.6. CONFIGURING DEBEZIUM CONNECTORS TO USE THE OUTBOX PATTERN

The outbox pattern is a way to safely and reliably exchange data between multiple (micro) services. An outbox pattern implementation avoids inconsistencies between a service's internal state (as typically persisted in its database) and state in events consumed by services that need the same data.

To implement the outbox pattern in a Debezium application, configure a Debezium connector to:

- Capture changes in an outbox table
- Apply the Debezium outbox event router single message transformation (SMT)

A Debezium connector that is configured to apply the outbox SMT should capture changes in only an outbox table. A connector can capture changes in more than one outbox table only if each outbox table has the same structure.

IMPORTANT

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

See Reliable Microservices Data Exchange With the Outbox Pattern to learn about why the outbox pattern is useful and how it works.

NOTE

The outbox event router SMT does **not** support the MongoDB connector.

The following topics provide details:

- Section 9.6.1, "Example of a Debezium outbox message"
- Section 9.6.2, "Outbox table structure expected by Debezium outbox event router SMT"

- Section 9.6.3, “Basic Debezium outbox event router SMT configuration”
- Section 9.6.4, “Using Avro as the payload format in Debezium outbox messages”
- Section 9.6.5, “Emitting additional fields in Debezium outbox messages”
- Section 9.6.6, “Options for configuring outbox event router transformation”

9.6.1. Example of a Debezium outbox message

To learn about how to configure the Debezium outbox event router SMT, consider the following example of a Debezium outbox message:

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\",
\"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\":
\"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
\"customerId\": 123}"
}
```

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

```
# Kafka Message key: "406c07f3-26f0-4eea-a50c-109940064b8f"
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484
{
  "before": null,
  "after": {
    "id": "406c07f3-26f0-4eea-a50c-109940064b8f",
    "aggregateid": "1",
    "aggregatetype": "Order",
    "payload": "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\":
\"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\",
\"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
\"customerId\": 123}",
    "timestamp": 1556890294344,
    "type": "OrderCreated"
  },
  "source": {
    "version": "1.2.4.Final",
    "connector": "postgresql",
    "name": "dbserver1-bare",
    "db": "orderdb",
    "ts_usec": 1556890294448870,
    "txId": 584,
    "lsn": 24064704,
    "schema": "inventory",
    "table": "outboxevent",
    "snapshot": false,
    "last_snapshot_record": null,
    "xmin": null
  },
  "op": "c",
  "ts_ms": 1556890294484
}
```

This example of a Debezium outbox message is based on the default outbox event router configuration, which assumes an outbox table structure and event routing based on aggregates. To customize behavior, the outbox event router SMT provides numerous configuration options.

9.6.2. Outbox table structure expected by Debezium outbox event router SMT

To apply the default outbox event router SMT configuration, your outbox table is assumed to have the following columns:

```
Column        |          Type          | Modifiers
--------------+------------------------+-----------
id            | uuid                   | not null
aggregatetype | character varying(255) | not null
aggregateid   | character varying(255) | not null
type          | character varying(255) | not null
payload       | jsonb                  |
```

Table 9.4. Descriptions of expected outbox table columns

Column	Effect
id	Contains the unique ID of the event. In an outbox message, this value is a header. You can use this ID, for example, to remove duplicate messages. To obtain the unique ID of the event from a different outbox table column, set the **table.field.event.id** SMT option in the connector configuration.
aggregatetype	Contains a value that the SMT appends to the name of the topic to which the connector emits an outbox message. The default behavior is that this value replaces the default **${routedByValue}** variable in the **route.topic.replacement** SMT option. For example, in a default configuration, the **route.by.field** SMT option is set to **aggregatetype** and the **route.topic.replacement** SMT option is set to **outbox.event.${routedByValue}**. Suppose that your application adds two records to the outbox table. In the first record, the value in the **aggregatetype** column is **customers**. In the second record, the value in the **aggregatetype** column is **orders**. The connector emits the first record to the **outbox.event.customers** topic. The connector emits the second record to the **outbox.event.orders** topic. To obtain this value from a different outbox table column, set the **route.by.field** SMT option in the connector configuration.

aggregateid	Contains the event key, which provides an ID for the payload. The SMT uses this value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions. To obtain the event key from a different outbox table column, set the **table.field.event.key** SMT option in the connector configuration.
type	A user-defined value that helps categorize or organize events.
payload	The representation of the event itself. The default structure is JSON. The content in this field becomes one of these: • Part of the outbox message **payload**. • If other metadata, including **eventType** is delivered as headers, the payload becomes the message itself without encapsulation in an envelope. To obtain the event payload from a different outbox table column, set the **table.field.event.payload** SMT option in the connector configuration.

9.6.3. Basic Debezium outbox event router SMT configuration

To configure a Debezium connector to support the outbox pattern, configure the **outbox.EventRouter** SMT. For example, the basic configuration in a **.properties** file looks like this:

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

9.6.4. Using Avro as the payload format in Debezium outbox messages

The outbox event router SMT supports arbitrary payload formats. The **payload** column value in an outbox table is passed on transparently. An alternative to working with JSON is to use Avro. This can be beneficial for message format governance and for ensuring that outbox event schemas evolve in a backwards-compatible way.

How a source application produces Avro formatted content for outbox message payloads is out of the scope of this documentation. One possibility is to leverage the **KafkaAvroSerializer** class to serialize **GenericRecord** instances. To ensure that the Kafka message value is the exact Avro binary data, apply the following configuration to the connector:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.ByteBufferConverter
```

By default, the **payload** column value (the Avro data) is the only message value. Configuration of **ByteBufferConverter** as the value converter propagates the **payload** column value as-is into the Kafka message value.

9.6.5. Emitting additional fields in Debezium outbox messages

Your outbox table might contain columns whose values you want to add to the emitted outbox messages. For example, consider an outbox table that has a value of **purchase-order** in the **aggregatetype** column and another column, **eventType**, whose possible values are **order-created** and **order-shipped**. To emit the **eventType** column value in the outbox message header, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=type:header:eventType
```

To emit the **eventType** column value in the outbox message envelope, configure the SMT like this:

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=type:envelope:eventType
```

9.6.6. Options for configuring outbox event router transformation

The following table describes the options that you can specify for the outbox event router SMT. In the table, the **Group** column indicates a configuration option classification for Kafka.

Table 9.5. Descriptions of outbox event router SMT configuration options

Option	Default	Group	Description
table.field.event.id	**id**	Table	Specifies the outbox table column that contains the unique event ID.
table.field .event.key	**aggregateid**	Table	Specifies the outbox table column that contains the event key. When this column contains a value, the SMT uses that value as the key in the emitted outbox message. This is important for maintaining correct order in Kafka partitions.
table.field.event .timestamp		Table	By default, the timestamp in the emitted outbox message is the Debezium event timestamp. To use a different timestamp in outbox messages, set this option to an outbox table column that contains the timestamp that you want to be in emitted outbox messages.
table.field.event .payload	**payload**	Table	Specifies the outbox table column that contains the event payload.
table.field.event .payload.id	**aggregateid**	Table	Specifies the outbox table column that contains the payload ID.

Option	Default	Group	Description
table.fields .additional .placement		Table, Envelope	Specifies one or more outbox table columns that you want to add to outbox message headers or envelopes. Specify a comma-separated list of pairs. In each pair, specify the name of a column and whether you want the value to be in the header or the envelope. Separate the values in the pair with a colon, for example: **id:header,my-field:envelope** To specify an alias for the column, specify a trio with the alias as the third value, for example: **id:header,my-field:envelope:my-alias** The second value is the placement and it must always be **header** or **envelope**. Configuration examples are in emitting additional fields in Debezium outbox messages.
table.field .event.schema .version		Table, Schema	When set, this value is used as the schema version as described in the Kafka Connect Schema Javadoc.
route.by.field	**aggregatetype**	Router	Specifies the name of a column in the outbox table. The default behavior is that the value in this column becomes a part of the name of the topic to which the connector emits the outbox messages. An example is in the description of the expected outbox table.

Option	Default	Group	Description
route.topic.regex	**(?<routedByValue>.*)**	Router	Specifies a regular expression that the outbox SMT applies in the RegexRouter to outbox table records. This regular expression is part of the setting of the **route.topic.replacement** SMT option. The default behavior is that the SMT replaces the default **${routedByValue}** variable in the setting of the **route.topic.replacement** SMT option with the setting of the **route.by.field** outbox SMT option.
route.topic .replacement	**outbox.event. ${routedByValue}**	Router	Specifies the name of the topic to which the connector emits outbox messages. The default topic name is **outbox.event.** followed by the **aggregatetype** column value in the outbox table record. For example, if the **aggregatetype** value is **customers**, the topic name is **outbox.event.customers**. To change the topic name, you can: ● Set the **route.by.field** option to a different column. ● Set the **route.topic.regex** option to a different regular expression.
route .tombstone.on .empty.payload	**false**	Router	Indicates whether an empty or **null** payload causes the connector to emit a tombstone event.

Option	Default	Group	Description
debezium.op .invalid.behavior	**warn**	Debezium	Determines the behavior of the SMT when there is an **UPDATE** operation on the outbox table. Possible settings are: • **warn** - The SMT logs a warning and continues to the next outbox table record. • **error** - The SMT logs an error and continues to the next outbox table record. • **fatal** - The SMT logs an error and the connector stops processing. All changes in an outbox table are expected to be **INSERT** operations. That is, an outbox table functions as a queue; updates to records in an outbox table are not allowed. The SMT automatically filters out **DELETE** operations on an outbox table.

9.7. EMITTING CHANGE EVENT RECORDS IN CLOUDEVENTS FORMAT

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

IMPORTANT

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

The CloudEvents specification defines:

- A set of standardized event attributes
- Rules for defining custom attributes
- Encoding rules for mapping event formats to serialized representations such as JSON or Avro

- Protocol bindings for transport layers such as Apache Kafka, HTTP or AMQP

To configure a Debezium connector to emit change event records that conform to the CloudEvents specification, Debezium provides the **io.debezium.converters.CloudEventsConverter**, which is a Kafka Connect message converter.

Currently, only structured mapping mode is supported. The CloudEvents change event envelope can be JSON or Avro and each envelope type supports JSON or Avro as the **data** format. It is expected that a future Debezium release will support binary mapping mode. For information about using Avro, see:

- Avro serialization
- Apicurio Registry

9.7.1. Example change event records in CloudEvents format

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

```
{
  "id" : "name:test_server;lsn:29274832;txId:565", 1
  "source" : "/debezium/postgresql/test_server", 2
  "specversion" : "1.0", 3
  "type" : "io.debezium.postgresql.datachangeevent", 4
  "time" : "2020-01-13T13:55:39.738Z", 5
  "datacontenttype" : "application/json", 6
  "iodebeziumop" : "r", 7
  "iodebeziumversion" : "1.2.4.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"
    }
  }
}
```

Table 9.6. Descriptions of fields in CloudEvents event with JSON format envelope

Item	Description
1	Unique ID that the connector generates for the change event based on the change event's content.
2	The source of the event, which is the logical name of the database as specified by the **database.server.name** property in the connector's configuration.
3	The CloudEvents specification version.
4	Connector type that generated the change event. The format of this field is **io.debezium.*CONNECTOR_TYPE*.datachangeevent**. The value of ***CONNECTOR_TYPE*** is **mongodb**, **mysql**, **postgresql**, or **sqlserver**.
5	Time of the change in the source database.
6	Describes the content type of the **data** attribute, which is JSON in this example. The only alternative is Avro.
7	An operation identifier. Possible values are **r** for read, **c** for create, **u** for update, or **d** for delete.
8	All **source** attributes that are known from Debezium change events are mapped to CloudEvents extension attributes by using the **iodebezium** prefix for the attribute name.
9	When enabled in the connector, each **transaction** attribute that is known from Debezium change events is mapped to a CloudEvents extension attribute by using the **iodebeziumtx** prefix for the attribute name.
10	The actual data change itself. Depending on the operation and the connector, the data might contain **before**, **after** and/or **patch** fields.

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

```
{
  "id" : "name:test_server;lsn:33227720;txId:578",
  "source" : "/debezium/postgresql/test_server",
  "specversion" : "1.0",
  "type" : "io.debezium.postgresql.datachangeevent",
  "time" : "2020-01-13T14:04:18.597Z",
  "datacontenttype" : "application/avro" (1)
  "dataschema" : "http://my-registry/schemas/ids/1", (2)
  "iodebeziumop" : "r",
  "iodebeziumversion" : "1.2.4.Final",
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578924258597",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "578",
  "iodebeziumlsn" : "33227720",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "578",
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : "AAAAAAEAAgICAg==" (3)
}
```

Table 9.7. Descriptions of fields in CloudEvents event with Avro data

Item	Description
1	Indicates that the **data** attribute contains Avro binary data.
2	URI of the schema to which the Avro data adheres.
3	The **data** attribute contains base64-encoded Avro binary data.

It is also possible to use Avro for the envelope as well as the **data** attribute.

9.7.2. Example of configuring CloudEventsConverter

Configure **io.debezium.converters.CloudEventsConverter** in your Debezium connector configuration. The following example shows how to configure **CloudEventsConverter** to emit change event records that have the following characteristics:

- Use JSON as the envelope.
- Use the schema registry at **http://my-registry/schemas/ids/1** to serialize the **data** attribute as binary Avro data.

```
...
"value.converter": "io.debezium.converters.CloudEventsConverter",
"value.converter.serializer.type" : "json",
"value.converter.data.serializer.type" : "avro",
"value.converter.avro.schema.registry.url": "http://my-registry/schemas/ids/1"
...
```

Specification of **serializer.type** is optional, because **json** is the default.

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**, **JsonConverter**, or **AvroConverter**.

9.7.3. CloudEventsConverter 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. The value can be **json** or **avro**.
data.serializer.type	**json**	The encoding type to use for the **data** attribute. The value can be **json** or **avro**.
json. ...	N/A	Any configuration properties to be passed through to the underlying converter when using JSON. The **json.** prefix is removed.
avro. ...	N/A	Any configuration properties to be passed through to the underlying converter when using Avro. The **avro.** prefix is removed. For example, for Avro **data**, you would specify the **avro.schema.registry.url** property.

Revised on 2020-10-20 00:34:17 UTC