

# {prodname} connector for MySQL

## Table of Contents

How the connector works .....	2
Supported MySQL topologies .....	2
Schema history topic .....	3
Schema change topic .....	4
Snapshots .....	9
Operation type of snapshot events .....	20
Topic names .....	21
Transaction metadata .....	21
Data change events .....	23
Change event keys .....	25
Change event values .....	26
<i>create</i> events .....	27
<i>update</i> events .....	33
Primary key updates .....	36
<i>delete</i> events .....	36
Tombstone events .....	38
Data type mappings .....	38
Basic types .....	38
Temporal types .....	41
Decimal types .....	42
Boolean values .....	43
Spatial types .....	44
Setting up MySQL .....	44
Creating a user .....	44
Enabling the binlog .....	46
Enabling GTIDs .....	47
Configuring session timeouts .....	48
Enabling query log events .....	49
Deployment .....	49
MySQL connector configuration example .....	50
Adding connector configuration .....	51
Connector properties .....	51
Monitoring .....	80
Snapshot metrics .....	80
Streaming metrics .....	83
Schema history metrics .....	85

Behavior when things go wrong .....	86
Configuration and startup errors .....	86
MySQL becomes unavailable .....	87
Kafka Connect stops gracefully .....	87
Kafka Connect process crashes .....	87
Kafka becomes unavailable .....	87
MySQL purges binlog files .....	88

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

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

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

For information about the MySQL Database versions that are compatible with this connector, see the [{prodname} release overview](#).

## How the connector works

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



The {prodname} MySQL connector has yet to be tested with MariaDB, but multiple reports from the community indicate successful usage of the connector with this database. Official support for MariaDB is planned for a future {prodname} version.

## Supported MySQL topologies

The {prodname} 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 {prodname} MySQL connector can monitor the server. This is often acceptable, since the binary log can also be used as an incremental [backup](#). In this case, the MySQL connector always connects to and follows this standalone MySQL server instance.

### Primary and replica

The {prodname} MySQL connector can follow one of the primary servers or one of the replicas

(if that replica has its binlog enabled), but the connector sees changes in only the cluster that is visible to that server. Generally, this is not a problem except for the multi-primary topologies.

The connector records its position in the server's binlog, which is different on each server in the cluster. Therefore, the connector must follow just one MySQL server instance. If that server fails, that server must be restarted or recovered before the connector can continue.

## High available clusters

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

## Multi-primary

[Network Database \(NDB\) cluster replication](#) uses one or more MySQL replica nodes that each replicate from multiple primary servers. This is a powerful way to aggregate the replication of multiple MySQL clusters. This topology requires the use of GTIDs.

A {prodname} MySQL connector can use these multi-primary MySQL replicas as sources, and can fail over to different multi-primary MySQL replicas as long as the new replica is caught up to the old replica. That is, the new replica has all transactions that were seen on the first replica. This works even if the connector is using only a subset of databases and/or tables, as the connector can be configured to include or exclude specific GTID sources when attempting to reconnect to a new multi-primary MySQL replica and find the correct position in the binlog.

## Hosted

There is support for the {prodname} MySQL connector to use hosted options such as Amazon RDS and Amazon Aurora.

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

# Schema history topic

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

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

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

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

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

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

See [default names for topics](#) that receive {prodname} event records.

## Schema change topic

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

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

### `ddl`

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

### `databaseName`

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

### `pos`

The position in the binlog where the statements appear.

### `tableChanges`

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



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

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



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

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



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

*Example: Message emitted to the MySQL connector schema change topic*

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

```
{
  "schema": { },
  "payload": {
    "source": { ①
      "version": "2.0.0.Beta1",
      "connector": "mysql",
      "name": "mysql",
      "ts_ms": 1651535750218, ②
      "snapshot": "false",
      "db": "inventory",
      "sequence": null,
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 570,
      "row": 0,
      "thread": null,
      "query": null
    },
    "databaseName": "inventory", ③
    "schemaName": null,
    "ddl": "ALTER TABLE customers ADD middle_name varchar(255) AFTER first_name", ④
    "tableChanges": [ ⑤
      {
        "type": "ALTER", ⑥
        "id": "\"inventory\".\"customers\"", ⑦
        "table": { ⑧
          "defaultCharset": "utf8mb4",
```

```

"primaryKeyColumnNames": [ ⑨
  "id"
],
"columns": [ ⑩
  {
    "name": "id",
    "jdbcType": 4,
    "nativeType": null,
    "typeName": "INT",
    "typeExpression": "INT",
    "charsetName": null,
    "length": null,
    "scale": null,
    "position": 1,
    "optional": false,
    "autoIncremented": true,
    "generated": true
  },
  {
    "name": "first_name",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR",
    "typeExpression": "VARCHAR",
    "charsetName": "utf8mb4",
    "length": 255,
    "scale": null,
    "position": 2,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "middle_name",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR",
    "typeExpression": "VARCHAR",
    "charsetName": "utf8mb4",
    "length": 255,
    "scale": null,
    "position": 3,
    "optional": true,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "last_name",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR",

```

```

        "typeExpression": "VARCHAR",
        "charsetName": "utf8mb4",
        "length": 255,
        "scale": null,
        "position": 4,
        "optional": false,
        "autoIncremented": false,
        "generated": false
    },
    {
        "name": "email",
        "jdbcType": 12,
        "nativeType": null,
        "typeName": "VARCHAR",
        "typeExpression": "VARCHAR",
        "charsetName": "utf8mb4",
        "length": 255,
        "scale": null,
        "position": 5,
        "optional": false,
        "autoIncremented": false,
        "generated": false
    }
],
"attributes": [ ⑪
    {
        "customAttribute": "attributeValue"
    }
]
}
]
}
}

```

Table 1. Descriptions of fields in messages emitted to the schema change topic

Item	Field name	Description
1	<b>source</b>	The <b>source</b> field is structured exactly as standard data change events that the connector writes to table-specific topics. This field is useful to correlate events on different topics.

Item	Field name	Description
2	<code>ts_ms</code>	<p>Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.</p> <p>In the source object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and Debezium.</p>
3	<code>databaseName</code> <code>schemaName</code>	Identifies the database and the schema that contains the change. The value of the <code>databaseName</code> field is used as the message key for the record.
4	<code>ddl</code>	<p>This field contains the DDL that is responsible for the schema change. The <code>ddl</code> field can contain multiple DDL statements. Each statement applies to the database in the <code>databaseName</code> field. Multiple DDL statements appear in the order in which they were applied to the database.</p> <p>Clients can submit multiple DDL statements that apply to multiple databases. If MySQL applies them atomically, the connector takes the DDL statements in order, groups them by database, and creates a schema change event for each group. If MySQL applies them individually, the connector creates a separate schema change event for each statement.</p>
5	<code>tableChanges</code>	An array of one or more items that contain the schema changes generated by a DDL command.
6	<code>type</code>	<p>Describes the kind of change. The value is one of the following:</p> <p><b>CREATE</b> Table created.</p> <p><b>ALTER</b> Table modified.</p> <p><b>DROP</b> Table deleted.</p>



Item	Field name	Description
7	<code>id</code>	Full identifier of the table that was created, altered, or dropped. In the case of a table rename, this identifier is a concatenation of <code>&lt;old&gt;</code> , <code>&lt;new&gt;</code> table names.
8	<code>table</code>	Represents table metadata after the applied change.
9	<code>primaryKeyColumnNames</code>	List of columns that compose the table's primary key.
10	<code>columns</code>	Metadata for each column in the changed table.
11	<code>attributes</code>	Custom attribute metadata for each table change.

See also: [schema history topic](#).

## Snapshots

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

Table 2. Workflow for performing an initial snapshot with a global read lock

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

Step	Action
9	Records the completed snapshot in the connector offsets.

### Connector restarts

If the connector fails, stops, or is rebalanced while performing the *initial snapshot*, then after the connector restarts, it performs a new snapshot. After that *initial snapshot* is completed, the {prodname} MySQL connector restarts from the same position in the binlog so it does not miss any updates.

If the connector stops for long enough, MySQL could purge old binlog files and the connector's position would be lost. If the position is lost, the connector reverts to the *initial snapshot* for its starting position. For more tips on troubleshooting the {prodname} MySQL connector, see [behavior when things go wrong](#).

### Global read locks not allowed

Some environments do not allow global read locks. If the {prodname} MySQL connector detects that global read locks are not permitted, the connector uses table-level locks instead and performs a snapshot with this method. This requires the database user for the {prodname} connector to have **LOCK TABLES** privileges.

Table 3. Workflow for performing an initial snapshot with table-level locks

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

### Ad hoc snapshots

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

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

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

You can re-run a snapshot for a table for which you previously captured a snapshot by initiating a so-called *ad-hoc snapshot*. Ad hoc snapshots require the use of [signaling tables](#). You initiate an ad hoc snapshot by sending a signal request to the {prodname} signaling table.

When you initiate an ad hoc snapshot of an existing table, the connector appends content to the topic that already exists for the table. If a previously existing topic was removed, {prodname} can create a topic automatically if [automatic topic creation](#) is enabled.

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

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

Table 4. Example of an ad hoc **execute-snapshot** signal record

Field	Default	Value
<b>type</b>	<b>incremental</b>	Specifies the type of snapshot that you want to run. Setting the type is optional. Currently, you can request only <b>incremental</b> snapshots.
<b>data-collections</b>	N/A	An array that contains regular expressions matching the fully-qualified names of the table to be snapshotted. The format of the names is the same as for the <b>signal.data.collection</b> configuration option.
<b>additional-condition</b>	Optional.Empty	An optional string, which specifies a condition based on the column(s) of the table(s), to capture a subset of the contents of the table(s).

#### Triggering an ad hoc snapshot

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

Currently, the **execute-snapshot** action type triggers [incremental snapshots](#) only. For more

information, see [Incremental snapshots](#).

## Incremental snapshots

To provide flexibility in managing snapshots, {prodname} includes a supplementary snapshot mechanism, known as *incremental snapshotting*. Incremental snapshots rely on the {prodname} mechanism for [sending signals to a {prodname} connector](#). Incremental snapshots are based on the [DDD-3](#) design document.

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

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

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

### *Incremental snapshot process*

When you run an incremental snapshot, {prodname} sorts each table by primary key and then splits the table into chunks based on the [configured chunk size](#). Working chunk by chunk, it then captures each table row in a chunk. For each row that it captures, the snapshot emits a `READ` event. That event represents the value of the row when the snapshot for the chunk began.

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

### *How {prodname} resolves collisions among records with the same primary key*

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

between the snapshot events and the streamed events are resolved does {prodname} emit an event record to Kafka.

### *Snapshot window*

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

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

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

The connector repeats the process for each snapshot chunk.

### *Triggering an incremental snapshot*

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

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

To specify the tables to include in the snapshot, provide a **data-collections** array that lists the tables or an array of regular expressions used to match tables, for example,

```
{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}
```

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



If the name of a table that you want to include in a snapshot contains a dot (.) in the name of the database, schema, or table, to add the table to the **data-collections**

array, you must escape each part of the name in double quotes.

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

#### Prerequisites

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

#### Procedure

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

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

For example,

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

The values of the **id**, **type**, and **data** parameters in the command correspond to the [fields of the signaling table](#).

The following table describes these parameters:

*Table 5. Descriptions of fields in a SQL command for sending an incremental snapshot signal to the signaling table*

Value	Description
<b>myschema.debezium_signal</b>	Specifies the fully-qualified name of the signaling table on the source database
<b>ad-hoc-1</b>	The <b>id</b> parameter specifies an arbitrary string that is assigned as the <b>id</b> identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. {prodname} does not use this string. Rather, during the snapshot, {prodname} generates its own <b>id</b> string as a watermarking signal.
<b>execute-snapshot</b>	Specifies <b>type</b> parameter specifies the operation that the signal is intended to trigger.

Value	Description
<code>data-collections</code>	<p>A required component of the <code>data</code> field of a signal that specifies an array of table names or regular expressions to match table names to include in the snapshot.</p> <p>The array lists regular expressions which match tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the <code>signal.data.collection</code> configuration property.</p>
<code>incremental</code>	<p>An optional <code>type</code> component of the <code>data</code> field of a signal that specifies the kind of snapshot operation to run.</p> <p>Currently, the only valid option is the default value, <code>incremental</code>.</p> <p>Specifying a <code>type</code> value in the SQL query that you submit to the signaling table is optional.</p> <p>If you do not specify a value, the connector runs an incremental snapshot.</p>
<code>additional-condition</code>	<p>An optional string, which specifies a condition based on the column(s) of the table(s), to capture a subset of the contents of the tables.</p>

#### *Ad hoc incremental snapshots with additional-condition*

- `additional-condition` is used to select a subset of a table's content.
- To give an analogy how `additional-condition` is used:
  - For a snapshot, the SQL query executed behind the scenes is something like:
 

```
SELECT * FROM <tableName> ....
```
  - For a snapshot with a `additional-condition`, the `additional-condition` is appended to the SQL query, something like:
 

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```
  - Send a SQL query to add the ad hoc incremental snapshot request to the signaling table:

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

- Suppose there is a `products` table with columns `id` (primary key), `color` and `brand`.

To snapshot just the content of the `products` table where `color=blue`

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1',
'execute-snapshot', '{"data-collections":
["schema1.products"], "type": "incremental", "additional-condition": "color=blue"}'
);
```



- `additional-condition` can be used to pass condition based on multiple columns. Using the same `products` table, to snapshot content of the `products` table where `color=blue` and `brand=foo`

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1',
'execute-snapshot', '{"data-collections":
["schema1.products"],"type":"incremental", "additional-condition":"color=blue AND
brand=foo"}');
```

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

*Example: Incremental snapshot event message*

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

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

### Stopping an incremental snapshot

Incremental snapshots can also be stopped by sending a signal to the table on the source database. You submit signals to the table as SQL `INSERT` queries. After {prodname} detects the change in the signaling table, it reads the signal, and stops the incremental snapshot operation if it's in progress.

The query that you submit specifies the snapshot operation of `incremental`, and, optionally, the tables of the current running snapshot to be removed.



## Prerequisites

- [Signaling is enabled](#).
  - A signaling data collection exists on the source database and the connector is configured to capture it.
  - The signaling data collection is specified in the `signal.data.collection` property.

## Procedure

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

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

For example,

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

The values of the `id`, `type`, and `data` parameters in the command correspond to the [fields of the signaling table](#).

The following table describes these parameters:

Table 6. Descriptions of fields in a SQL command for sending a stop incremental snapshot signal to the signaling table

Value	Description
<code>myschema.debezium_signal</code>	Specifies the fully-qualified name of the signaling table on the source database
<code>ad-hoc-1</code>	The <code>id</code> parameter specifies an arbitrary string that is assigned as the <code>id</code> identifier for the signal request. Use this string to identify logging messages to entries in the signaling table. {prodname} does not use this string.
<code>stop-snapshot</code>	Specifies <code>type</code> parameter specifies the operation that the signal is intended to trigger.
<code>data-collections</code>	An optional component of the <code>data</code> field of a signal that specifies an array of table names or regular expressions to match table names to remove from the snapshot. The array lists regular expressions which match tables by their fully-qualified names, using the same format as you use to specify the name of the connector's signaling table in the <code>signal.data.collection</code> configuration property. If this component of the <code>data</code> field is omitted, the signal stops the entire incremental snapshot that is in progress.

Value	Description
<code>incremental</code>	<p>A required component of the <code>data</code> field of a signal that specifies the kind of snapshot operation that is to be stopped.</p> <p>Currently, the only valid option is <code>incremental</code>.</p> <p>Specifying a <code>type</code> value in the SQL query that you submit to the signaling table is required.</p> <p>If you do not specify a value, the signal will not stop the incremental snapshot.</p>

## Read-only incremental snapshots

The MySQL connector allows for running incremental snapshots with a read-only connection to the database. To run an incremental snapshot with read-only access, the connector uses the executed global transaction IDs (GTID) set as high and low watermarks. The state of a chunk's window is updated by comparing the GTIDs of binary log (binlog) events or the server's heartbeats against low and high watermarks.

To switch to a read-only implementation, set the value of the `read.only` property to `true`.

### Prerequisites

- [Enable MySQL GTIDs](#).
- If the connector reads from a multi-threaded replica (that is, a replica for which the value of `replica_parallel_workers` is greater than 0) you must set one of the following options:
  - `replica_preserve_commit_order=ON`
  - `slave_preserve_commit_order=ON`

## Ad hoc read-only incremental snapshots

When the MySQL connection is read-only, the [signaling table](#) mechanism can also run a snapshot by sending a message to the Kafka topic that is specified in the `signal.kafka.topic` property.

The key of the Kafka message must match the value of the `database.server.name` connector configuration option.

The value is a JSON object with `type` and `data` fields.

The signal type is `execute-snapshot` and the `data` field must have the following fields:

Table 7. Execute snapshot data fields

Field	Default	Value
<code>type</code>	<code>incremental</code>	<p>The type of the snapshot to be executed. Currently only <code>incremental</code> is supported.</p> <p>See the next section for more details.</p>

Field	Default	Value
<code>data-collections</code>	N/A	An array of comma-separated regular expressions that match fully-qualified names of tables to be snapshot. The format of the names is the same as for <a href="#">signal.data.collection</a> configuration option.
<code>additional-condition</code>	Optional.Empty	An optional string, which specifies a condition based on the column(s) of the table(s), to capture a subset of the contents of the table(s).

An example of the execute-snapshot Kafka message:

```
Key = `test_connector`
```

```
Value = `{"type":"execute-snapshot","data":{"data-collections":["schema1.table1",
"schema1.table2"], "type": "INCREMENTAL"}}`
```

## Ad hoc read-only incremental snapshots with additional-condition

- `additional-condition` is used to select a subset of a table's content.
- To give an analogy how `additional-condition` is used:
  - For a snapshot, the SQL query executed behind the scenes is something like:
 

```
SELECT * FROM <tableName> ....
```
  - For a snapshot with a `additional-condition`, the `additional-condition` is appended to the SQL query, something like:
 

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```
- Suppose there is a `products` table with columns `id` (primary key), `color` and `brand`.

To snapshot just the content of the `products` table where `color=blue`

```
Key = `test_connector`
```

```
Value = `{"type":"execute-snapshot","data":{"data-collections":
["schema1.products"], "type": "INCREMENTAL", "additional-condition":"color=blue"}}`
```

- `additional-condition` can be used to pass condition based on multiple columns. Using the same `products` table, to snapshot content of the `products` table where `color=blue` and `brand=foo`

```
Key = `test_connector`
```

```
Value = `{"type":"execute-snapshot","data":{"data-collections":
["schema1.products"], "type": "INCREMENTAL", "additional-condition":"color=blue AND
```

```
brand=foo"}}"`
```

## Stopping an Ad hoc read-only incremental snapshot

When the MySQL connection is read-only, the [signaling table](#) mechanism can also stop a snapshot by sending a message to the Kafka topic that is specified in the [signal.kafka.topic](#) property.

The key of the Kafka message must match the value of the [database.server.name](#) connector configuration option.

The value is a JSON object with [type](#) and [data](#) fields.

The signal type is [stop-snapshot](#) and the [data](#) field must have the following fields:

Table 8. Execute snapshot data fields

Field	Default	Value
<a href="#">type</a>	<a href="#">incremental</a>	The type of the snapshot to be executed. Currently only <a href="#">incremental</a> is supported. See the next section for more details.
<a href="#">data-collections</a>	N/A	An optional array of comma-separated regular expressions that match fully-qualified names of tables to be snapshotted. The format of the names is the same as for <a href="#">signal.data.collection</a> configuration option.

An example of the stop-snapshot Kafka message:

```
Key = `test_connector`
```

```
Value = `{"type":"stop-snapshot","data": {"data-collections": ["schema1.table1",  
"schema1.table2"], "type": "INCREMENTAL"}}`
```

## Operation type of snapshot events

The MySQL connector emits snapshot events as [READ](#) operations (["op" : "r"](#)). If you prefer that the connector emits snapshot events as [CREATE](#) ([c](#)) events, configure the {prodname} [ReadToInsertEvent](#) single message transform (SMT) to modify the event type.

The following example shows how to configure the SMT:

Example: Using the [ReadToInsertEvent](#) SMT to change the type of snapshot events

```
transforms=snapshotasinsert,...  
transforms.snapshotasinsert.type=io.debezium.connector.mysql.transforms.ReadToInsertEvent
```

# Topic names

By default, the MySQL connector writes change events for all of the **INSERT**, **UPDATE**, and **DELETE** operations that occur in a table to a single Apache Kafka topic that is specific to that table.

The connector uses the following convention to name change event topics:

*serverName.databaseName.tableName*

Suppose that **fulfillment** is the server name, **inventory** is the database name, and the database contains tables named **orders**, **customers**, and **products**. The {prodname} MySQL connector emits events to three Kafka topics, one for each table in the database:

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

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

## **serverName**

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

## **schemaName**

The name of the schema in which the operation occurred.

## **tableName**

The name of the table in which the operation occurred.

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

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

# Transaction metadata

{prodname} can generate events that represent transaction boundaries and that enrich data change event messages.



### *Limits on when {prodname} receives transaction metadata*

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

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

**status**

**BEGIN** or **END**.

**id**

String representation of the unique transaction identifier.

**ts\_ms**

The time of a transaction boundary event (**BEGIN** or **END** event) at the data source. If the data source does not provide {prodname} with the event time, then the field instead represents the time at which {prodname} processes the event.

**event\_count (for END events)**

Total number of events emitted by the transaction.

**data\_collections (for END events)**

An array of pairs of **data\_collection** and **event\_count** elements that indicates the number of events that the connector emits for changes that originate from a data collection.

*Example*

```
{
  "status": "BEGIN",
  "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
  "ts_ms": 1486500577125,
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
  "ts_ms": 1486500577691,
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "s1.a",
      "event_count": 1
    },
    {
      "data_collection": "s2.a",
      "event_count": 1
    }
  ]
}
```

Unless overridden via the **topic.transaction** option, the connector emits transaction events to the

<database.server.name>.transaction topic.

### *Change data event enrichment*

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

#### **id**

String representation of unique transaction identifier.

#### **total\_order**

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

#### **data\_collection\_order**

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

Following is an example of a message:

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
    ...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

For systems which don't have GTID enabled, the transaction identifier is constructed using the combination of binlog filename and binlog position. For example, if the binlog filename and position corresponding to the transaction BEGIN event are mysql-bin.000002 and 1913 respectively then the {prodname} constructed transaction identifier would be **file=mysql-bin.000002,pos=1913**.

## Data change events

The {prodname} 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.

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

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

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

Table 9. Overview of change event basic content

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



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

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](#).



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.

## 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.

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

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

```
{
```

```

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

```

Table 10. Description of change event key

Item	Field name	Description
1	<code>schema</code>	The schema portion of the key specifies a Kafka Connect schema that describes what is in the key's <code>payload</code> portion.
2	<code>mysql-server-1.inventory.customers.Key</code>	<p>Name of the schema that defines the structure of the key's payload. This schema describes the structure of the primary key for the table that was changed. Key schema names have the format <i>connector-name.database-name.table-name.Key</i>. In this example:</p> <ul style="list-style-type: none"> <li>• <code>mysql-server-1</code> is the name of the connector that generated this event.</li> <li>• <code>inventory</code> is the database that contains the table that was changed.</li> <li>• <code>customers</code> is the table that was updated.</li> </ul>
3	<code>optional</code>	Indicates whether the event key must contain a value in its <code>payload</code> 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	<code>fields</code>	Specifies each field that is expected in the <code>payload</code> , including each field's name, type, and whether it is required.
5	<code>payload</code>	Contains the key for the row for which this change event was generated. In this example, the key, contains a single <code>id</code> field whose value is <code>1001</code> .

## 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:

- [create events](#)
- [update events](#)
- [Primary key updates](#)
- [delete events](#)
- [Tombstone events](#)

## **create events**

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

```
{  
  "schema": { ①  
    "type": "struct",  
    "fields": [  
      {  
        "type": "struct",  
        "fields": [  
          {  
            "type": "int32",  
            "optional": false,  
            "field": "id"  
          },  
          {  
            "type": "string",  
            "optional": false,  
            "field": "first_name"  
          },  
          {  
            "type": "string",  
            "optional": false,  
            "field": "last_name"  
          }  
        ]  
      }  
    ]  
  }
```

```

        {
            "type": "string",
            "optional": false,
            "field": "email"
        }
    ],
    "optional": true,
    "name": "mysql-server-1.inventory.customers.Value", ②
    "field": "before"
},
{
    "type": "struct",
    "fields": [
        {
            "type": "int32",
            "optional": false,
            "field": "id"
        },
        {
            "type": "string",
            "optional": false,
            "field": "first_name"
        },
        {
            "type": "string",
            "optional": false,
            "field": "last_name"
        },
        {
            "type": "string",
            "optional": false,
            "field": "email"
        }
    ],
    "optional": true,
    "name": "mysql-server-1.inventory.customers.Value",
    "field": "after"
},
{
    "type": "struct",
    "fields": [
        {
            "type": "string",
            "optional": false,
            "field": "version"
        },
        {
            "type": "string",
            "optional": false,
            "field": "connector"
        }
    ],

```

```

{
  "type": "string",
  "optional": false,
  "field": "name"
},
{
  "type": "int64",
  "optional": false,
  "field": "ts_ms"
},
{
  "type": "boolean",
  "optional": true,
  "default": false,
  "field": "snapshot"
},
{
  "type": "string",
  "optional": false,
  "field": "db"
},
{
  "type": "string",
  "optional": true,
  "field": "table"
},
{
  "type": "int64",
  "optional": false,
  "field": "server_id"
},
{
  "type": "string",
  "optional": true,
  "field": "gtid"
},
{
  "type": "string",
  "optional": false,
  "field": "file"
},
{
  "type": "int64",
  "optional": false,
  "field": "pos"
},
{
  "type": "int32",
  "optional": false,
  "field": "row"
},
},

```

```

        {
            "type": "int64",
            "optional": true,
            "field": "thread"
        },
        {
            "type": "string",
            "optional": true,
            "field": "query"
        }
    ],
    "optional": false,
    "name": "io.debezium.connector.mysql.Source", ③
    "field": "source"
},
{
    "type": "string",
    "optional": false,
    "field": "op"
},
{
    "type": "int64",
    "optional": true,
    "field": "ts_ms"
}
],
"optional": false,
"name": "mysql-server-1.inventory.customers.Envelope" ④
},
"payload": { ⑤
    "op": "c", ⑥
    "ts_ms": 1465491411815, ⑦
    "before": null, ⑧
    "after": { ⑨
        "id": 1004,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
    },
    "source": { ⑩
        "version": "2.0.0.Beta1",
        "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 11. Descriptions of create event value fields

Item	Field name	Description
1	<code>schema</code>	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	<code>name</code>	<p>In the <code>schema</code> section, each <code>name</code> field specifies the schema for a field in the value's payload.</p> <p><code>mysql-server-1.inventory.customers.Value</code> is the schema for the payload's <code>before</code> and <code>after</code> fields. This schema is specific to the <code>customers</code> table.</p> <p>Names of schemas for <code>before</code> and <code>after</code> fields are of the form <code>logicalName.tableName.Value</code>, which ensures that the schema name is unique in the database. This means that when using the <a href="#">Avro converter</a>, the resulting Avro schema for each table in each logical source has its own evolution and history.</p>
3	<code>name</code>	<code>io.debezium.connector.mysql.Source</code> is the schema for the payload's <code>source</code> field. This schema is specific to the MySQL connector. The connector uses it for all events that it generates.
4	<code>name</code>	<code>mysql-server-1.inventory.customers.Envelope</code> is the schema for the overall structure of the payload, where <code>mysql-server-1</code> is the connector name, <code>inventory</code> is the database, and <code>customers</code> is the table.
5	<code>payload</code>	<p>The value's actual data. This is the information that the change event is providing.</p> <p>It may appear that the JSON representations of the events are much larger than the rows they describe. This is because the JSON representation must include the schema and the payload portions of the message. However, by using the <a href="#">Avro converter</a>, you can significantly decrease the size of the messages that the connector streams to Kafka topics.</p>

Item	Field name	Description
6	<code>op</code>	<p>Mandatory string that describes the type of operation that caused the connector to generate the event. In this example, <code>c</code> indicates that the operation created a row. Valid values are:</p> <ul style="list-style-type: none"> <li>• <code>c</code> = create</li> <li>• <code>u</code> = update</li> <li>• <code>d</code> = delete</li> <li>• <code>r</code> = read (applies to only snapshots)</li> </ul>
7	<code>ts_ms</code>	<p>Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.</p> <p>In the <code>source</code> object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and <code>{prodname}</code>.</p>
8	<code>before</code>	<p>An optional field that specifies the state of the row before the event occurred. When the <code>op</code> field is <code>c</code> for create, as it is in this example, the <code>before</code> field is <code>null</code> since this change event is for new content.</p>
9	<code>after</code>	<p>An optional field that specifies the state of the row after the event occurred. In this example, the <code>after</code> field contains the values of the new row's <code>id</code>, <code>first_name</code>, <code>last_name</code>, and <code>email</code> columns.</p>



Item	Field name	Description
10	<code>source</code>	<p>Mandatory field that describes the source metadata for the event. This field contains information that you can use to compare this event with other events, with regard to the origin of the events, the order in which the events occurred, and whether events were part of the same transaction. The source metadata includes:</p> <ul style="list-style-type: none"> <li>• <code>{prodname}</code> version</li> <li>• Connector name</li> <li>• binlog name where the event was recorded</li> <li>• binlog position</li> <li>• Row within the event</li> <li>• If the event was part of a snapshot</li> <li>• Name of the database and table that contain the new row</li> <li>• ID of the MySQL thread that created the event (non-snapshot only)</li> <li>• MySQL server ID (if available)</li> <li>• Timestamp for when the change was made in the database</li> </ul> <p>If the <code>binlog_rows_query_log_events</code> MySQL configuration option is enabled and the connector configuration <code>include.query</code> property is enabled, the <code>source</code> field also provides the <code>query</code> field, which contains the original SQL statement that caused the change event.</p>

## update events

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

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

```

```

    "email": "annek@noanswer.org"
  },
  "source": { ③
    "version": "2.0.0.Beta1",
    "name": "mysql-server-1",
    "connector": "mysql",
    "name": "mysql-server-1",
    "ts_ms": 1465581029100,
    "snapshot": false,
    "db": "inventory",
    "table": "customers",
    "server_id": 223344,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 484,
    "row": 0,
    "thread": 7,
    "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
  },
  "op": "u", ④
  "ts_ms": 1465581029523 ⑤
}
}

```

Table 12. Descriptions of update event value fields

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

Item	Field name	Description
3	<code>source</code>	<p>Mandatory field that describes the source metadata for the event. The <code>source</code> field structure has the same fields as in a <i>create</i> event, but some values are different, for example, the sample <i>update</i> event is from a different position in the binlog. The source metadata includes:</p> <ul style="list-style-type: none"> <li>• <code>{prodname}</code> version</li> <li>• Connector name</li> <li>• binlog name where the event was recorded</li> <li>• binlog position</li> <li>• Row within the event</li> <li>• If the event was part of a snapshot</li> <li>• Name of the database and table that contain the updated row</li> <li>• ID of the MySQL thread that created the event (non-snapshot only)</li> <li>• MySQL server ID (if available)</li> <li>• Timestamp for when the change was made in the database</li> </ul> <p>If the <code>binlog_rows_query_log_events</code> MySQL configuration option is enabled and the connector configuration <code>include.query</code> property is enabled, the <code>source</code> field also provides the <code>query</code> field, which contains the original SQL statement that caused the change event.</p>
4	<code>op</code>	<p>Mandatory string that describes the type of operation. In an <i>update</i> event value, the <code>op</code> field value is <code>u</code>, signifying that this row changed because of an update.</p>
5	<code>ts_ms</code>	<p>Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.</p> <p>In the <code>source</code> object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and <code>{prodname}</code>.</p>



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

# Primary key updates

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

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

## delete events

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

```
{
  "schema": { ... },
  "payload": {
    "before": { ①
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, ②
    "source": { ③
      "version": "2.0.0.Beta1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581902300,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d", ④
    "ts_ms": 1465581902461 ⑤
  }
}
```

Table 13. Descriptions of delete event value fields

Item	Field name	Description
1	<code>before</code>	Optional field that specifies the state of the row before the event occurred. In a <i>delete</i> event value, the <code>before</code> field contains the values that were in the row before it was deleted with the database commit.
2	<code>after</code>	Optional field that specifies the state of the row after the event occurred. In a <i>delete</i> event value, the <code>after</code> field is <code>null</code> , signifying that the row no longer exists.
3	<code>source</code>	<p>Mandatory field that describes the source metadata for the event. In a <i>delete</i> event value, the <code>source</code> field structure is the same as for <i>create</i> and <i>update</i> events for the same table. Many <code>source</code> field values are also the same. In a <i>delete</i> event value, the <code>ts_ms</code> and <code>pos</code> field values, as well as other values, might have changed. But the <code>source</code> field in a <i>delete</i> event value provides the same metadata:</p> <ul style="list-style-type: none"> <li>• <code>{prodname}</code> version</li> <li>• Connector name</li> <li>• binlog name where the event was recorded</li> <li>• binlog position</li> <li>• Row within the event</li> <li>• If the event was part of a snapshot</li> <li>• Name of the database and table that contain the updated row</li> <li>• ID of the MySQL thread that created the event (non-snapshot only)</li> <li>• MySQL server ID (if available)</li> <li>• Timestamp for when the change was made in the database</li> </ul> <p>If the <code>binlog_rows_query_log_events</code> MySQL configuration option is enabled and the connector configuration <code>include.query</code> property is enabled, the <code>source</code> field also provides the <code>query</code> field, which contains the original SQL statement that caused the change event.</p>
4	<code>op</code>	Mandatory string that describes the type of operation. The <code>op</code> field value is <code>d</code> , signifying that this row was deleted.
5	<code>ts_ms</code>	<p>Optional field that displays the time at which the connector processed the event. The time is based on the system clock in the JVM running the Kafka Connect task.</p> <p>In the <code>source</code> object, <code>ts_ms</code> indicates the time that the change was made in the database. By comparing the value for <code>payload.source.ts_ms</code> with the value for <code>payload.ts_ms</code>, you can determine the lag between the source database update and <code>{prodname}</code>.</p>

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

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

## Tombstone events

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

## Data type mappings

The {prodname} 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 {prodname} represents the value in the event.

Columns that store strings are defined in MySQL with a character set and collation. The MySQL connector uses the column's character set when reading the binary representation of the column values in the binlog events.

The connector can map MySQL data types to both *literal* and *semantic* types.

- **Literal type:** how the value is represented using Kafka Connect schema types.
- **Semantic type:** how the Kafka Connect schema captures the meaning of the field (schema name).

If the default data type conversions do not meet your needs, you can [create a custom converter](#) for the connector.

## Basic types

The following table shows how the connector maps basic MySQL data types.

Table 14. Descriptions of basic type mappings

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

MySQL type	Literal type	Semantic type
BIT(>1)	BYTES	<p><code>io.debezium.data.Bits</code></p> <p>The <code>length</code> schema parameter contains an integer that represents the number of bits. The <code>byte[]</code> contains the bits in <i>little-endian</i> form and is sized to contain the specified number of bits. For example, where <code>n</code> is bits:</p> <p><code>numBytes = n/8 + (n%8 == 0 ? 0 : 1)</code></p>
TINYINT	INT16	<i>n/a</i>
SMALLINT[(M)]	INT16	<i>n/a</i>
MEDIUMINT[(M)]	INT32	<i>n/a</i>
INT, INTEGER[(M)]	INT32	<i>n/a</i>
BIGINT[(M)]	INT64	<i>n/a</i>
REAL[(M,D)]	FLOAT32	<i>n/a</i>
FLOAT[(P)]	FLOAT32 or FLOAT64	The precision is used only to determine storage size. A precision <code>P</code> from 0 to 23 results in a 4-byte single-precision <code>FLOAT32</code> column. A precision <code>P</code> from 24 to 53 results in an 8-byte double-precision <code>FLOAT64</code> column.
FLOAT(M,D)	FLOAT64	As of MySQL 8.0.17, the nonstandard <code>FLOAT(M,D)</code> and <code>DOUBLE(M,D)</code> syntax is deprecated, and should expect support for it be removed in a future version of MySQL, set <code>FLOAT64</code> as default.
DOUBLE[(M,D)]	FLOAT64	<i>n/a</i>
CHAR(M)]	STRING	<i>n/a</i>
VARCHAR(M)]	STRING	<i>n/a</i>
BINARY(M)]	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p>
VARBINARY(M)]	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p>
TINYBLOB	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p>
TINYTEXT	STRING	<i>n/a</i>

MySQL type	Literal type	Semantic type
BLOB	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p> <p>Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.</p>
TEXT	STRING	<p><i>n/a</i></p> <p>Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.</p>
MEDIUMBLOB	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p>
MEDIUMTEXT	STRING	<i>n/a</i>
LONGBLOB	BYTES or STRING	<p><i>n/a</i></p> <p>Either the raw bytes (the default), a base64-encoded String, or a hex-encoded String, based on the <code>binary.handling.mode</code> connector configuration property setting.</p> <p>Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.</p>
LONGTEXT	STRING	<p><i>n/a</i></p> <p>Only values with a size of up to 2GB are supported. It is recommended to externalize large column values, using the claim check pattern.</p>
JSON	STRING	<p><code>io.debezium.data.Json</code></p> <p>Contains the string representation of a JSON document, array, or scalar.</p>
ENUM	STRING	<p><code>io.debezium.data.Enum</code></p> <p>The <code>allowed</code> schema parameter contains the comma-separated list of allowed values.</p>
SET	STRING	<p><code>io.debezium.data.EnumSet</code></p> <p>The <code>allowed</code> schema parameter contains the comma-separated list of allowed values.</p>
YEAR[(2 4)]	INT32	<code>io.debezium.time.Year</code>



MySQL type	Literal type	Semantic type
<code>TIMESTAMP(M)</code>	<code>STRING</code>	<code>io.debezium.time.ZonedTimestamp</code> In <a href="#">ISO 8601</a> format with microsecond precision. MySQL allows <code>M</code> to be in the range of <code>0-6</code> .

## Temporal types

Excluding the `TIMESTAMP` data type, MySQL temporal types depend on the value of the `time.precision.mode` connector configuration property. For `TIMESTAMP` columns whose default value is specified as `CURRENT_TIMESTAMP` or `NOW`, the value `1970-01-01 00:00:00` is used as the default value in the Kafka Connect schema.

MySQL allows zero-values for `DATE`, `DATETIME`, and `TIMESTAMP` columns because zero-values are sometimes preferred over null values. The MySQL connector represents zero-values as null values when the column definition allows null values, or as the epoch day when the column does not allow null values.

### *Temporal values without time zones*

The `DATETIME` type represents a local date and time such as "2018-01-13 09:48:27". As you can see, there is no time zone information. Such columns are converted into epoch milliseconds or microseconds based on the column's precision by using UTC. The `TIMESTAMP` type represents a timestamp without time zone information. It is converted by MySQL from the server (or session's) current time zone into UTC when writing and from UTC into the server (or session's) current time zone when reading back the value. For example:

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

Such columns are converted into an equivalent `io.debezium.time.ZonedTimestamp` in UTC based on the server (or session's) current time zone. The time zone will be queried from the server by default. If this fails, it must be specified explicitly by the database `connectionTimeZone` MySQL configuration option. For example, if the database's time zone (either globally or configured for the connector by means of the `connectionTimeZone` option) is "America/Los\_Angeles", the `TIMESTAMP` value "2018-06-20 06:37:03" is represented by a `ZonedTimestamp` with the value "2018-06-20T13:37:03Z".

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

More details about properties related to temporal values are in the documentation for [MySQL connector configuration properties](#).

### **time.precision.mode=adaptive\_time\_microseconds(default)**

The MySQL connector determines the literal type and semantic type based on the column's data type definition so that events represent exactly the values in the database. All time fields are in microseconds. Only positive `TIME` field values in the range of `00:00:00.000000` to `23:59:59.999999` can be captured correctly.

*Table 15. Mappings when `time.precision.mode=adaptive_time_microseconds`*

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

### **time.precision.mode=connect**

The MySQL connector uses defined Kafka Connect logical types. This approach is less precise than the default approach and the events could be less precise if the database column has a *fractional second precision* value of greater than 3. Values in only the range of `00:00:00.000` to `23:59:59.999` can be handled. Set `time.precision.mode=connect` only if you can ensure that the `TIME` values in your tables never exceed the supported ranges. The `connect` setting is expected to be removed in a future version of {prodname}.

Table 16. Mappings when `time.precision.mode=connect`

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

## Decimal types

{prodname} connectors handle decimals according to the setting of the `decimal.handling.mode` [connector configuration property](#).

### **decimal.handling.mode=precise**

Table 17. Mappings when `decimal.handling.mode=precise`

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

### decimal.handling.mode=double

Table 18. Mappings when `decimal.handling.mode=double`

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

### decimal.handling.mode=string

Table 19. Mappings when `decimal.handling.mode=string`

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

## Boolean values

MySQL handles the `BOOLEAN` value internally in a specific way. The `BOOLEAN` column is internally mapped to the `TINYINT(1)` data type. When the table is created during streaming then it uses proper `BOOLEAN` mapping as {prodname} receives the original DDL. During snapshots, {prodname} executes `SHOW CREATE TABLE` to obtain table definitions that return `TINYINT(1)` for both `BOOLEAN` and `TINYINT(1)` columns. {prodname} then has no way to obtain the original type mapping and so maps to `TINYINT(1)`.

To enable you to convert source columns to Boolean data types, {prodname} provides a `TinyIntOneToBooleanConverter` custom converter that you can use in one of the following ways:

- Map all `TINYINT(1)` or `TINYINT(1) UNSIGNED` columns to `BOOLEAN` types.
- Enumerate a subset of columns by using a comma-separated list of regular expressions.  
To use this type of conversion, you must set the `converters` configuration property with the `selector` parameter, as shown in the following example:

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

- NOTE: MySQL8 not showing the length of `tinyint unsigned` type when snapshot executes `SHOW CREATE TABLE`, which means this converter doesn't work. The new option `length.checker` can solve this issue, the default value is `true`. Disable the `length.checker` and specify the columns that need to be converted to `selector` property instead of converting all columns based on type, as shown in the following example:

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

## Spatial types

Currently, the {prodname} MySQL connector supports the following spatial data types.

Table 20. Description of spatial type mappings

MySQL type	Literal type	Semantic type
GEOMETRY, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	STRUCT	<code>io.debezium.data.geometry.Geometry</code> Contains a structure with two fields: <ul style="list-style-type: none"><li>• <code>srid</code> (<code>INT32</code>): spatial reference system ID that defines the type of geometry object stored in the structure</li><li>• <code>wkb</code> (<code>BYTES</code>): binary representation of the geometry object encoded in the Well-Known-Binary (wkb) format. See the <a href="#">Open Geospatial Consortium</a> for more details.</li></ul>

## Setting up MySQL

Some MySQL setup tasks are required before you can install and run a {prodname} connector.

### Creating a user

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

#### Prerequisites

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

#### Procedure

1. Create the MySQL user:

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

2. Grant the required permissions to the user:

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

The table below describes the permissions.



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

3. Finalize the user's permissions:

```
mysql> FLUSH PRIVILEGES;
```

Table 21. Descriptions of user permissions

Keyword	Description
<b>SELECT</b>	Enables the connector to select rows from tables in databases. This is used only when performing a snapshot.
<b>RELOAD</b>	Enables the connector the use of the <b>FLUSH</b> statement to clear or reload internal caches, flush tables, or acquire locks. This is used only when performing a snapshot.
<b>SHOW DATABASES</b>	Enables the connector to see database names by issuing the <b>SHOW DATABASE</b> statement. This is used only when performing a snapshot.
<b>REPLICATION SLAVE</b>	Enables the connector to connect to and read the MySQL server binlog.
<b>REPLICATION CLIENT</b>	Enables the connector the use of the following statements: <ul style="list-style-type: none"><li>• <b>SHOW MASTER STATUS</b></li><li>• <b>SHOW SLAVE STATUS</b></li><li>• <b>SHOW BINARY LOGS</b></li></ul> The connector always requires this.
<b>ON</b>	Identifies the database to which the permissions apply.
<b>TO 'user'</b>	Specifies the user to grant the permissions to.
<b>IDENTIFIED BY 'password'</b>	Specifies the user's MySQL password.

# Enabling the binlog

You must enable binary logging for MySQL replication. The binary logs record transaction updates for replication tools to propagate changes.

## Prerequisites

- A MySQL server.
- Appropriate MySQL user privileges.

## Procedure

1. Check whether the **log-bin** option is already on:

```
// for MySql 5.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
// for MySql 8.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM performance_schema.global_variables WHERE variable_name='log_bin';
```

2. If it is **OFF**, configure your MySQL server configuration file with the following properties, which are described in the table below:

```
server-id      = 223344
log_bin        = mysql-bin
binlog_format  = ROW
binlog_row_image = FULL
expire_logs_days = 10
```

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

```
// for MySql 5.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
// for MySql 8.x
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM performance_schema.global_variables WHERE variable_name='log_bin';
```

Table 22. Descriptions of MySQL binlog configuration properties

Property	Description
<b>server-id</b>	The value for the <b>server-id</b> must be unique for each server and replication client in the MySQL cluster. During MySQL connector set up, {prodname} assigns a unique server ID to the connector.
<b>log_bin</b>	The value of <b>log_bin</b> is the base name of the sequence of binlog files.
<b>binlog_format</b>	The <b>binlog-format</b> must be set to <b>ROW</b> or <b>row</b> .

Property	Description
<code>binlog_row_image</code>	The <code>binlog_row_image</code> must be set to <code>FULL</code> or <code>full</code> .
<code>expire_logs_days</code>	This is the number of days for automatic binlog file removal. The default is <code>0</code> , which means no automatic removal. Set the value to match the needs of your environment. See <a href="#">MySQL purges binlog files</a> .

## Enabling GTIDs

Global transaction identifiers (GTIDs) uniquely identify transactions that occur on a server within a cluster. Though not required for a {prodname} MySQL connector, using GTIDs simplifies replication and enables you to more easily confirm if primary and replica servers are consistent.

GTIDs are available in MySQL 5.6.5 and later. See the [MySQL documentation](#) for more details.

### Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

### Procedure

1. Enable `gtid_mode`:

```
mysql> gtid_mode=ON
```

2. Enable `enforce_gtid_consistency`:

```
mysql> enforce_gtid_consistency=ON
```

3. Confirm the changes:

```
mysql> show global variables like '%GTID%';
```

### Result

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

Table 23. Descriptions of GTID options

Option	Description
<code>gtid_mode</code>	Boolean that specifies whether GTID mode of the MySQL server is enabled or not. <ul style="list-style-type: none"> <li>• <code>ON</code> = enabled</li> <li>• <code>OFF</code> = disabled</li> </ul>
<code>enforce_gtid_consistency</code>	Boolean that specifies whether the server enforces GTID consistency by allowing the execution of statements that can be logged in a transactionally safe manner. Required when using GTIDs. <ul style="list-style-type: none"> <li>• <code>ON</code> = enabled</li> <li>• <code>OFF</code> = disabled</li> </ul>

## Configuring session timeouts

When an initial consistent snapshot is made for large databases, your established connection could timeout while the tables are being read. You can prevent this behavior by configuring `interactive_timeout` and `wait_timeout` in your MySQL configuration file.

### Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

### Procedure

1. Configure `interactive_timeout`:

```
mysql> interactive_timeout=<duration-in-seconds>
```

2. Configure `wait_timeout`:

```
mysql> wait_timeout=<duration-in-seconds>
```

Table 24. Descriptions of MySQL session timeout options

Option	Description
<code>interactive_timeout</code>	The number of seconds the server waits for activity on an interactive connection before closing it. See <a href="#">MySQL's documentation</a> for more details.
<code>wait_timeout</code>	The number of seconds the server waits for activity on a non-interactive connection before closing it. See <a href="#">MySQL's documentation</a> for more details.



# Enabling query log events

You might want to see the original **SQL** statement for each binlog event. Enabling the **binlog\_rows\_query\_log\_events** option in the MySQL configuration file allows you to do this.

This option is available in MySQL 5.6 and later.

## Prerequisites

- A MySQL server.
- Basic knowledge of SQL commands.
- Access to the MySQL configuration file.

## Procedure

- Enable **binlog\_rows\_query\_log\_events**:

```
mysql> binlog_rows_query_log_events=ON
```

**binlog\_rows\_query\_log\_events** is set to a value that enables/disables support for including the original **SQL** statement in the binlog entry.

- **ON** = enabled
- **OFF** = disabled

# Deployment

To deploy a {prodname} MySQL connector, you install the {prodname} MySQL connector archive, configure the connector, and start the connector by adding its configuration to Kafka Connect.

## Prerequisites

- [Apache Zookeeper](#), [Apache Kafka](#), and [Kafka Connect](#) are installed.
- MySQL Server is installed and is [set up to work with the {prodname} connector](#).

## Procedure

1. Download the {prodname} [MySQL connector plug-in](#).
2. Extract the files into your Kafka Connect environment.
3. Add the directory with the JAR files to [Kafka Connect's plugin.path](#).
4. [Configure the connector](#) and [add the configuration to your Kafka Connect cluster](#).
5. Restart your Kafka Connect process to pick up the new JAR files.

If you are working with immutable containers, see [{prodname}'s Container images](#) for Apache Zookeeper, Apache Kafka, MySQL, and Kafka Connect with the MySQL connector already installed and ready to run.

You can also [run {prodname} on Kubernetes and OpenShift](#).

# MySQL connector configuration example

Following is an example of the configuration for a connector instance that captures data from a MySQL server on port 3306 at 192.168.99.100, which we logically name `fullfillment`. Typically, you configure the {prodname} MySQL connector in a JSON file by setting the configuration properties that are available for the connector.

You can choose to produce events for a subset of the schemas and tables in a database. Optionally, you can ignore, mask, or truncate columns that contain sensitive data, that are larger than a specified size, or that you do not need.

```
{
  "name": "inventory-connector", ①
  "config": {
    "connector.class": "io.debezium.connector.mysql.MySqlConnector", ②
    "database.hostname": "192.168.99.100", ③
    "database.port": "3306", ④
    "database.user": "debezium-user", ⑤
    "database.password": "debezium-user-pw", ⑥
    "database.server.id": "184054", ⑦
    "database.server.name": "fullfillment", ⑧
    "database.include.list": "inventory", ⑨
    "database.history.kafka.bootstrap.servers": "kafka:9092", ⑩
    "database.history.kafka.topic": "dbhistory.fullfillment", ⑪
    "include.schema.changes": "true" ⑫
  }
}
```

- ① Connector's name when registered with the Kafka Connect service.
- ② Connector's class name.
- ③ MySQL server address.
- ④ MySQL server port number.
- ⑤ MySQL user with the appropriate privileges.
- ⑥ MySQL user's password.
- ⑦ Unique ID of the connector.
- ⑧ Logical name of the MySQL server or cluster.
- ⑨ List of databases hosted by the specified server.
- ⑩ List of Kafka brokers that the connector uses to write and recover DDL statements to the database history topic.
- ⑪ Name of the database history topic. This topic is for internal use only and should not be used by consumers.
- ⑫ Flag that specifies if the connector should generate events for DDL changes and emit them to the `fullfillment` schema change topic for use by consumers.

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

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 performs the following actions:

- Connects to the MySQL database.
- Reads change-data tables for tables in capture mode.
- Streams change event records to Kafka topics.

## Adding connector configuration

To start running a MySQL connector, configure a connector configuration, and add the configuration to your Kafka Connect cluster.

### *Prerequisites*

- [MySQL is set up to work with a {prodname} connector](#).
- The {prodname} MySQL connector is installed.

### *Procedure*

1. Create a configuration for the MySQL connector.
2. Use the [Kafka Connect REST API](#) to add that connector configuration to your Kafka Connect cluster.

### *Results*

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

## Connector properties


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

- [Required connector configuration properties](#)
- [Advanced connector configuration properties](#)
- [Database history connector configuration properties](#) that control how {prodname} processes events that it reads from the database history topic.
  - [Pass-through database history properties](#)
- [Pass-through database driver properties](#) that control the behavior of the database driver.

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

## Required {prodname} MySQL connector configuration properties

Property	Default	Description
<code>name</code>	No default	Unique name for the connector. Attempting to register again with the same name fails. This property is required by all Kafka Connect connectors.
<code>connector.class</code>	No default	The name of the Java class for the connector. Always specify <code>io.debezium.connector.mysql.MySqlConnector</code> for the MySQL connector.
<code>tasks.max</code>	<code>1</code>	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.
<code>database.hostname</code>	No default	IP address or host name of the MySQL database server.
<code>database.port</code>	<code>3306</code>	Integer port number of the MySQL database server.
<code>database.user</code>	No default	Name of the MySQL user to use when connecting to the MySQL database server.
<code>database.password</code>	No default	Password to use when connecting to the MySQL database server.

Property	Default	Description
<code>database.server.name</code>	No default	<p>Logical name that identifies and provides a namespace for the particular MySQL database server/cluster in which {prodname} is capturing changes. The logical name should be unique across all other connectors, since it is used as a prefix for all Kafka topic names that receive events emitted by this connector. Only alphanumeric characters, hyphens, dots and underscores must be used in the database server logical name.</p> <div>  <p>Do not change the value of this property. If you change the name value, after a restart, instead of continuing to emit events to the original topics, the connector emits subsequent events to topics whose names are based on the new value. The connector is also unable to recover its database history topic.</p> </div>
<code>database.server.id</code>	<i>random</i>	A numeric ID of this database client, which must be unique across all currently-running database processes in the MySQL cluster. This connector joins the MySQL database cluster as another server (with this unique ID) so it can read the binlog.
<code>database.include.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match the names of the databases for which to capture changes. The connector does not capture changes in any database whose name is not in <code>database.include.list</code> . By default, the connector captures changes in all databases. Do not also set the <code>database.exclude.list</code> connector configuration property.

Property	Default	Description
<code>database.exclude.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match the names of databases for which you do not want to capture changes. The connector captures changes in any database whose name is not in the <code>database.exclude.list</code> . Do not also set the <code>database.include.list</code> connector configuration property.
<code>table.include.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers of tables whose changes you want to capture. The connector does not capture changes in any table not included in <code>table.include.list</code> . Each identifier is of the form <i>databaseName.tableName</i> . By default, the connector captures changes in every non-system table in each database whose changes are being captured. Do not also specify the <code>table.exclude.list</code> connector configuration property.
<code>table.exclude.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match fully-qualified table identifiers for tables whose changes you do not want to capture. The connector captures changes in any table not included in <code>table.exclude.list</code> . Each identifier is of the form <i>databaseName.tableName</i> . Do not also specify the <code>table.include.list</code> connector configuration property.
<code>column.exclude.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to exclude from change event record values. Fully-qualified names for columns are of the form <i>databaseName.tableName.columnName</i> .
<code>column.include.list</code>	<i>empty string</i>	An optional, comma-separated list of regular expressions that match the fully-qualified names of columns to include in change event record values. Fully-qualified names for columns are of the form <i>databaseName.tableName.columnName</i> .

Property	Default	Description
<code>column.truncate.to._length_.characters</code>	<i>n/a</i>	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be truncated in the change event record values if the field values are longer than the specified number of characters. You can configure multiple properties with different lengths in a single configuration. The length must be a positive integer. Fully-qualified names for columns are of the form <i>databaseName.tableName.columnName</i> .
<code>column.mask.with._length_.characters</code>	<i>n/a</i>	An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns whose values should be replaced in the change event message values with a field value consisting of the specified number of asterisk (*) characters. You can configure multiple properties with different lengths in a single configuration. Each length must be a positive integer or zero. Fully-qualified names for columns are of the form <i>databaseName.tableName.columnName</i> .

Property	Default	Description
<code>column.mask.hash.hashAlgorithm</code> <code>.with.salt.salt;</code> <code>column.mask.hash.v2.hashAlgorithm.with.salt.salt</code>	<i>n/a</i>	<p>An optional, comma-separated list of regular expressions that match the fully-qualified names of character-based columns. Fully-qualified names for columns are of the form <code>&lt;databaseName&gt;.&lt;tableName&gt;.&lt;columnName&gt;</code>. In the resulting change event record, the values for the specified columns are replaced with pseudonyms.</p> <p>A pseudonym consists of the hashed value that results from applying the specified <i>hashAlgorithm</i> and <i>salt</i>. Based on the hash function that is used, referential integrity is maintained, while column values are replaced with pseudonyms. Supported hash functions are described in the <a href="#">MessageDigest section</a> of the Java Cryptography Architecture Standard Algorithm Name Documentation.</p> <p>In the following example, <code>CzQMA0cB5K</code> is a randomly selected salt.</p> <div style="border: 1px solid #ccc; padding: 10px; margin: 10px 0;"> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> </div> <p>If necessary, the pseudonym is automatically shortened to the length of the column. The connector configuration can include multiple properties that specify different hash algorithms and salts.</p> <p>Depending on the <i>hashAlgorithm</i> used, the <i>salt</i> selected, and the actual data set, the resulting data set might not be completely masked.</p> <p>Hashing strategy version 2 should be used to ensure fidelity if the value is being hashed in different places or systems.</p>



Property	Default	Description
<code>column.propagate.source.type</code>	<i>n/a</i>	<p>An optional, comma-separated list of regular expressions that match the fully-qualified names of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change event records. These schema parameters:</p> <pre> __<i>{prodname}</i>.source.column.type __<i>{prodname}</i>.source.column.length __<i>{prodname}</i>.source.column.scale </pre> <p>are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified names for columns are of one of these forms:</p> <pre> <i>databaseName.tableName.columnName</i>  <i>databaseName.schemaName.tableName.columnName</i> </pre>

Property	Default	Description
<code>datatype.propagate.source.type</code>	<i>n/a</i>	<p>An optional, comma-separated list of regular expressions that match the database-specific data type name of columns whose original type and length should be added as a parameter to the corresponding field schemas in the emitted change event records. These schema parameters:</p> <pre>__debezium.source.column.type</pre> <pre>__debezium.source.column.length</pre> <pre>__debezium.source.column.scale</pre> <p>are used to propagate the original type name and length for variable-width types, respectively. This is useful to properly size corresponding columns in sink databases. Fully-qualified data type names are of one of these forms:</p> <pre>databaseName.tableName.typeName</pre> <pre>databaseName.schemaName.tableName.typeName</pre> <p>See <a href="#">how MySQL connectors map data types</a> for the list of MySQL-specific data type names.</p>

Property	Default	Description
<code>time.precision.mode</code>	<code>adaptive_time_microseconds</code>	<p>Time, date, and timestamps can be represented with different kinds of precision, including:</p> <p><code>adaptive_time_microseconds</code> (the default) captures the date, datetime and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type, with the exception of TIME type fields, which are always captured as microseconds.</p> <p><code>adaptive</code> (deprecated) captures the time and timestamp values exactly as in the database using either millisecond, microsecond, or nanosecond precision values based on the database column's type.</p> <p><code>connect</code> always represents time and timestamp values using Kafka Connect's built-in representations for Time, Date, and Timestamp, which use millisecond precision regardless of the database columns' precision.</p>
<code>decimal.handling.mode</code>	<code>precise</code>	<p>Specifies how the connector should handle values for <code>DECIMAL</code> and <code>NUMERIC</code> columns:</p> <p><code>precise</code> (the default) represents them precisely using <code>java.math.BigDecimal</code> values represented in change events in a binary form.</p> <p><code>double</code> represents them using <code>double</code> values, which may result in a loss of precision but is easier to use.</p> <p><code>string</code> encodes values as formatted strings, which is easy to consume but semantic information about the real type is lost.</p>

Property	Default	Description
<code>bigint.unsigned.handling.mode</code>	<code>long</code>	<p>Specifies how BIGINT UNSIGNED columns should be represented in change events. Possible settings are:</p> <p><code>long</code> represents values by using Java's <code>long</code>, which might not offer the precision but which is easy to use in consumers. <code>long</code> is usually the preferred setting.</p> <p><code>precise</code> uses <code>java.math.BigDecimal</code> to represent values, which are encoded in the change events by using a binary representation and Kafka Connect's <code>org.apache.kafka.connect.data.Decimal</code> type. Use this setting when working with values larger than <math>2^{63}</math>, because these values cannot be conveyed by using <code>long</code>.</p>
<code>include.schema.changes</code>	<code>true</code>	<p>Boolean value that specifies whether the connector should publish changes in the database schema to a Kafka topic with the same name as the database server ID. Each schema change is recorded by using a key that contains the database name and whose value includes the DDL statement(s). This is independent of how the connector internally records database history.</p>
<code>include.schema.comments</code>	<code>false</code>	<p>Boolean value that specifies whether the connector should parse and publish table and column comments on metadata objects. Enabling this option will bring the implications on memory usage. The number and size of logical schema objects is what largely impacts how much memory is consumed by the Debezium connectors, and adding potentially large string data to each of them can potentially be quite expensive.</p>

Property	Default	Description
<code>include.query</code>	<code>false</code>	<p>Boolean value that specifies whether the connector should include the original SQL query that generated the change event.</p> <p>If you set this option to <code>true</code> then you must also configure MySQL with the <code>binlog_rows_query_log_events</code> option set to <code>ON</code>. When <code>include.query</code> is <code>true</code>, the query is not present for events that the snapshot process generates.</p> <p>Setting <code>include.query</code> to <code>true</code> might expose tables or fields that are explicitly excluded or masked by including the original SQL statement in the change event. For this reason, the default setting is <code>false</code>.</p>
<code>event.deserialization.failure.handling.mode</code>	<code>fail</code>	<p>Specifies how the connector should react to exceptions during deserialization of binlog events.</p> <p><code>fail</code> propagates the exception, which indicates the problematic event and its binlog offset, and causes the connector to stop.</p> <p><code>warn</code> logs the problematic event and its binlog offset and then skips the event.</p> <p><code>ignore</code> passes over the problematic event and does not log anything.</p>
<code>inconsistent.schema.handling.mode</code>	<code>fail</code>	<p>Specifies how the connector should react to binlog events that relate to tables that are not present in internal schema representation. That is, the internal representation is not consistent with the database.</p> <p><code>fail</code> throws an exception that indicates the problematic event and its binlog offset, and causes the connector to stop.</p> <p><code>warn</code> logs the problematic event and its binlog offset and skips the event.</p> <p><code>skip</code> passes over the problematic event and does not log anything.</p>

Property	Default	Description
<code>max.batch.size</code>	2048	Positive integer value that specifies the maximum size of each batch of events that should be processed during each iteration of this connector. Defaults to 2048.
<code>max.queue.size</code>	8192	Positive integer value that specifies the maximum number of records that the blocking queue can hold. When {prodname} reads events streamed from the database, it places the events in the blocking queue before it writes them to Kafka. The blocking queue can provide backpressure for reading change events from the database in cases where the connector ingests messages faster than it can write them to Kafka, or when Kafka becomes unavailable. Events that are held in the queue are disregarded when the connector periodically records offsets. Always set the value of <code>max.queue.size</code> to be larger than the value of <code>max.batch.size</code> .
<code>max.queue.size.in.bytes</code>	0	A long integer value that specifies the maximum volume of the blocking queue in bytes. By default, volume limits are not specified for the blocking queue. To specify the number of bytes that the queue can consume, set this property to a positive long value.  If <code>max.queue.size</code> is also set, writing to the queue is blocked when the size of the queue reaches the limit specified by either property. For example, if you set <code>max.queue.size=1000</code> , and <code>max.queue.size.in.bytes=5000</code> , writing to the queue is blocked after the queue contains 1000 records, or after the volume of the records in the queue reaches 5000 bytes.
<code>poll.interval.ms</code>	500	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.
<code>connect.timeout.ms</code>	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
<code>gtid.source.includes</code>	No default	A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources that match one of these include patterns are used. Do not also specify a setting for <code>gtid.source.excludes</code> .
<code>gtid.source.excludes</code>	No default	A comma-separated list of regular expressions that match source UUIDs in the GTID set used to find the binlog position in the MySQL server. Only the GTID ranges that have sources that do not match any of these exclude patterns are used. Do not also specify a value for <code>gtid.source.includes</code> .
<code>tombstones.on.delete</code>	<code>true</code>	<p>Controls whether a <i>delete</i> event is followed by a tombstone event.</p> <p><code>true</code> - a delete operation is represented by a <i>delete</i> event and a subsequent tombstone event.</p> <p><code>false</code> - only a <i>delete</i> event is emitted.</p> <p>After a source record is deleted, emitting a tombstone event (the default behavior) allows Kafka to completely delete all events that pertain to the key of the deleted row in case <a href="#">log compaction</a> is enabled for the topic.</p>

Property	Default	Description
<code>message.key.columns</code>	<i>n/a</i>	<p>A list of expressions that specify the columns that the connector uses to form custom message keys for change event records that it publishes to the Kafka topics for specified tables.</p> <p>By default, {prodname} uses the primary key column of a table as the message key for records that it emits. In place of the default, or to specify a key for tables that lack a primary key, you can configure custom message keys based on one or more columns.</p> <p>To establish a custom message key for a table, list the table, followed by the columns to use as the message key. Each list entry takes the following format:</p> <pre>&lt;fully-qualified_tableName&gt; : &lt;keyColumn&gt;_, &lt;keyColumn&gt;</pre> <p>To base a table key on multiple column names, insert commas between the column names.</p> <p>Each fully-qualified table name is a regular expression in the following format:</p> <pre>&lt;databaseName&gt;.&lt;tableName&gt;</pre> <p>The property can include entries for multiple tables. Use a semicolon to separate table entries in the list.</p> <p>The following example sets the message key for the tables <code>inventory.customers</code> and <code>purchase.orders</code>:</p> <pre>inventory.customers:pk1,pk2;(.*)purchaseorders:pk3,pk4</pre> <p>For the table <code>inventory.customer</code>, the columns <code>pk1</code> and <code>pk2</code> are specified as the message key. For the <code>purchaseorders</code> tables in any database, the columns <code>pk3</code> and <code>pk4</code> server as the message key.</p> <p>There is no limit to the number of columns that you use to create custom message keys. However, it's best to use the minimum number</p>



Property	Default	Description
<code>binary.handling.mode</code>	bytes	<p>Specifies how binary columns, for example, <code>blob</code>, <code>binary</code>, <code>varbinary</code>, should be represented in change events. Possible settings:</p> <p><code>bytes</code> represents binary data as a byte array.</p> <p><code>base64</code> represents binary data as a base64-encoded String.</p> <p><code>hex</code> represents binary data as a hex-encoded (base16) String.</p>
<code>schema.name.adjustment.mode</code>	avro	<p>Specifies how schema names should be adjusted for compatibility with the message converter used by the connector. Possible settings:</p> <ul style="list-style-type: none"> <li>• <code>avro</code> replaces the characters that cannot be used in the Avro type name with underscore.</li> <li>• <code>none</code> does not apply any adjustment.</li> </ul>

## Advanced MySQL connector configuration properties

The following table describes [advanced MySQL connector properties](#). The default values for these properties rarely need to be changed. Therefore, you do not need to specify them in the connector configuration.

Table 25. Descriptions of MySQL connector advanced configuration properties

Property	Default	Description
<code>connect.keep.alive</code>	true	A Boolean value that specifies whether a separate thread should be used to ensure that the connection to the MySQL server/cluster is kept alive.

Property	Default	Description
<code>converters</code>	No default	<p>Enumerates a comma-separated list of the symbolic names of the <code>custom converter</code> instances that the connector can use.</p> <p>For example, <code>boolean</code>.</p> <p>This property is required to enable the connector to use a custom converter.</p> <p>For each converter that you configure for a connector, you must also add a <code>.type</code> property, which specifies the fully-qualified name of the class that implements the converter interface. The <code>.type</code> property uses the following format:</p> <pre>&lt;converterSymbolicName&gt;.type</pre> <p>For example,</p> <pre>boolean.type: io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter</pre> <p>If you want to further control the behavior of a configured converter, you can add one or more configuration parameters to pass values to the converter. To associate these additional configuration parameter with a converter, prefix the parameter name with the symbolic name of the converter.</p> <p>For example, to define a <code>selector</code> parameter that specifies the subset of columns that the <code>boolean</code> converter processes, add the following property:</p> <pre>boolean.selector=db1.table1.*, db1.table2.column1</pre>
<code>table.ignore.builtin</code>	<code>true</code>	<p>A Boolean value that specifies whether built-in system tables should be ignored. This applies regardless of the table include and exclude lists. By default, system tables are excluded from having their changes captured, and no events are generated when changes are made to any system tables.</p>

Property	Default	Description
<code>database.ssl.mode</code>	<code>disabled</code>	<p>Specifies whether to use an encrypted connection. Possible settings are:</p> <p><code>disabled</code> specifies the use of an unencrypted connection.</p> <p><code>preferred</code> establishes an encrypted connection if the server supports secure connections. If the server does not support secure connections, falls back to an unencrypted connection.</p> <p><code>required</code> establishes an encrypted connection or fails if one cannot be made for any reason.</p> <p><code>verify_ca</code> behaves like <code>required</code> but additionally it verifies the server TLS certificate against the configured Certificate Authority (CA) certificates and fails if the server TLS certificate does not match any valid CA certificates.</p> <p><code>verify_identity</code> behaves like <code>verify_ca</code> but additionally verifies that the server certificate matches the host of the remote connection.</p>

Property	Default	Description
<code>binlog.buffer.size</code>	0	<p>The size of a look-ahead buffer used by the binlog reader. The default setting of <code>0</code> disables buffering.</p> <p>Under specific conditions, it is possible that the MySQL binlog contains uncommitted data finished by a <code>ROLLBACK</code> statement. Typical examples are using savepoints or mixing temporary and regular table changes in a single transaction.</p> <p>When a beginning of a transaction is detected then {prodname} tries to roll forward the binlog position and find either <code>COMMIT</code> or <code>ROLLBACK</code> so it can determine whether to stream the changes from the transaction. The size of the binlog buffer defines the maximum number of changes in the transaction that {prodname} can buffer while searching for transaction boundaries. If the size of the transaction is larger than the buffer then {prodname} must rewind and re-read the events that have not fit into the buffer while streaming.</p> <p>NOTE: This feature is incubating. Feedback is encouraged. It is expected that this feature is not completely polished.</p>

Property	Default	Description
<code>snapshot.mode</code>	<code>initial</code>	<p>Specifies the criteria for running a snapshot when the connector starts. Possible settings are:</p> <p><code>initial</code> - the connector runs a snapshot only when no offsets have been recorded for the logical server name.</p> <p><code>initial_only</code> - the connector runs a snapshot only when no offsets have been recorded for the logical server name and then stops; i.e. it will not read change events from the binlog.</p> <p><code>when_needed</code> - the connector runs a snapshot upon startup whenever it deems it necessary. That is, when no offsets are available, or when a previously recorded offset specifies a binlog location or GTID that is not available in the server.</p> <p><code>never</code> - the connector never uses snapshots. Upon first startup with a logical server name, the connector reads from the beginning of the binlog. Configure this behavior with care. It is valid only when the binlog is guaranteed to contain the entire history of the database.</p> <p><code>schema_only</code> - the connector runs a snapshot of the schemas and not the data. This setting is useful when you do not need the topics to contain a consistent snapshot of the data but need them to have only the changes since the connector was started.</p> <p><code>schema_only_recovery</code> - this is a recovery setting for a connector that has already been capturing changes. When you restart the connector, this setting enables recovery of a corrupted or lost database history topic. You might set it periodically to "clean up" a database history topic that has been growing unexpectedly. Database history topics require infinite retention.</p>

Property	Default	Description
<code>snapshot.locking.mode</code>	<code>minimal</code>	<p>Controls whether and how long the connector holds the global MySQL read lock, which prevents any updates to the database, while the connector is performing a snapshot. Possible settings are:</p> <p><code>minimal</code> - the connector holds the global read lock for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global read lock is no longer held and other MySQL clients are updating the database.</p> <p><code>minimal_percona</code> - the connector holds the <a href="#">global backup lock</a> for only the initial portion of the snapshot during which the connector reads the database schemas and other metadata. The remaining work in a snapshot involves selecting all rows from each table. The connector can do this in a consistent fashion by using a REPEATABLE READ transaction. This is the case even when the global backup lock is no longer held and other MySQL clients are updating the database. This mode does not flush tables to disk, is not blocked by long-running reads, and is available only in Percona Server.</p> <p><code>extended</code> - blocks all writes for the duration of the snapshot. Use this setting if there are clients that are submitting operations that MySQL excludes from REPEATABLE READ semantics.</p> <p><code>none</code> - prevents the connector from acquiring any table locks during the snapshot. While this setting is allowed with all snapshot modes, it is safe to use if and <i>only</i> if no schema changes are happening while the snapshot is running. For tables defined with MyISAM engine, the tables would still be locked despite this property being set as MyISAM acquires a table lock. This behavior is unlike InnoDB engine, which acquires row level locks.</p>

Property	Default	Description
<code>snapshot.include.collection.list</code>	All tables specified in <code>table.include.list</code>	An optional, comma-separated list of regular expressions that match the fully-qualified names (<databaseName>.<tableName>) of the tables to include in a snapshot. The specified items must be named in the connector's <code>table.include.list</code> property. This property takes effect only if the connector's <code>snapshot.mode</code> property is set to a value other than <code>never</code> . This property does not affect the behavior of incremental snapshots.


Property	Default	Description
<code>snapshot.select.statement.overrides</code>	No default	<p>Specifies the table rows to include in a snapshot. Use the property if you want a snapshot to include only a subset of the rows in a table. This property affects snapshots only. It does not apply to events that the connector reads from the log.</p> <p>The property contains a comma-separated list of fully-qualified table names in the form <code>&lt;databaseName&gt;.&lt;tableName&gt;</code>. For example,</p> <pre>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</pre> <p>For each table in the list, add a further configuration property that specifies the <code>SELECT</code> statement for the connector to run on the table when it takes a snapshot. The specified <code>SELECT</code> statement determines the subset of table rows to include in the snapshot. Use the following format to specify the name of this <code>SELECT</code> statement property:</p> <pre>snapshot.select.statement.overrides.&lt;databaseName&gt;.&lt;tableName&gt;. For example, snapshot.select.statement.overrides.customers.orders.</pre> <p>Example:</p> <p>From a <code>customers.orders</code> table that includes the soft-delete column, <code>delete_flag</code>, add the following properties if you want a snapshot to include only those records that are not soft-deleted:</p> <div style="border: 1px solid #ccc; padding: 10px; margin: 10px 0;"> <pre>"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre> </div> <p>In the resulting snapshot, the connector includes only the records for which <code>delete_flag = 0</code>.</p>



Property	Default	Description
<code>min.row.count.to.stream.results</code>	1000	<p>During a snapshot, the connector queries each table for which the connector is configured to capture changes. The connector uses each query result to produce a read event that contains data for all rows in that table. This property determines whether the MySQL connector puts results for a table into memory, which is fast but requires large amounts of memory, or streams the results, which can be slower but work for very large tables. The setting of this property specifies the minimum number of rows a table must contain before the connector streams results.</p> <p>To skip all table size checks and always stream all results during a snapshot, set this property to 0.</p>
<code>heartbeat.interval.ms</code>	0	<p>Controls how frequently the connector sends heartbeat messages to a Kafka topic. The default behavior is that the connector does not send heartbeat messages.</p> <p>Heartbeat messages are useful for monitoring whether the connector is receiving change events from the database. Heartbeat messages might help decrease the number of change events that need to be re-sent when a connector restarts. To send heartbeat messages, set this property to a positive integer, which indicates the number of milliseconds between heartbeat messages.</p>
<code>heartbeat.action.query</code>	No default	<p>Specifies a query that the connector executes on the source database when the connector sends a heartbeat message.</p> <p>For example, this can be used to periodically capture the state of the executed GTID set in the source database.</p> <pre>INSERT INTO gtid_history_table (select * from mysql.gtid_executed)</pre>

Property	Default	Description
<code>database.initial.statements</code>	No default	<p>A semicolon separated list of SQL statements to be executed when a JDBC connection, not the connection that is reading the transaction log, to the database is established. To specify a semicolon as a character in a SQL statement and not as a delimiter, use two semicolons, (<code>;;</code>).</p> <p>The connector might establish JDBC connections at its own discretion, so this property is only for configuring session parameters. It is not for executing DML statements.</p>
<code>snapshot.delay.ms</code>	No default	An interval in milliseconds that the connector should wait before performing a snapshot when the connector starts. If you are starting multiple connectors in a cluster, this property is useful for avoiding snapshot interruptions, which might cause re-balancing of connectors.
<code>snapshot.fetch.size</code>	No default	During a snapshot, the connector reads table content in batches of rows. This property specifies the maximum number of rows in a batch.
<code>snapshot.lock.timeout.ms</code>	<code>10000</code>	Positive integer that specifies the maximum amount of time (in milliseconds) to wait to obtain table locks when performing a snapshot. If the connector cannot acquire table locks in this time interval, the snapshot fails. See <a href="#">how MySQL connectors perform database snapshots</a> .
<code>enable.time.adjuster</code>	<code>true</code>	<p>Boolean value that indicates whether the connector converts a 2-digit year specification to 4 digits. Set to <code>false</code> when conversion is fully delegated to the database.</p> <p>MySQL allows users to insert year values with either 2-digits or 4-digits. For 2-digit values, the value gets mapped to a year in the range 1970 - 2069. The default behavior is that the connector does the conversion.</p>

Property	Default	Description
<code>source.struct.version</code>	<code>v2</code>	<p>Schema version for the <code>source</code> block in <code>{prodname}</code> events. <code>{prodname}</code> 0.10 introduced a few breaking changes to the structure of the <code>source</code> block in order to unify the exposed structure across all the connectors.</p> <p>By setting this option to <code>v1</code>, the structure used in earlier versions can be produced. However, this setting is not recommended and is planned for removal in a future <code>{prodname}</code> version.</p>
<code>sanitize.field.names</code>	<code>true</code> if connector configuration sets the <code>key.converter</code> or <code>value.converter</code> property to the Avro converter. <code>false</code> if not.	Indicates whether field names are sanitized to adhere to <a href="#">Avro naming requirements</a> .
<code>skipped.operations</code>	No default	Comma-separated list of operation types to skip during streaming. The following values are possible: <code>c</code> for inserts/create, <code>u</code> for updates, <code>d</code> for deletes. By default, no operations are skipped.
<code>signal.data.collection</code>	No default value	<p>Fully-qualified name of the data collection that is used to send <a href="#">signals</a> to the connector.</p> <p>Use the following format to specify the collection name:</p> <p><code>&lt;databaseName&gt;.&lt;tableName&gt;</code></p>
<code>incremental.snapshot.allow.schema.changes</code>	<code>false</code>	<p>Allow schema changes during an incremental snapshot. When enabled the connector will detect schema change during an incremental snapshot and re-select a current chunk to avoid locking DDLs.</p> <p>Note that changes to a primary key are not supported and can cause incorrect results if performed during an incremental snapshot. Another limitation is that if a schema change affects only columns' default values, then the change won't be detected until the DDL is processed from the binlog stream. This doesn't affect the snapshot events' values, but the schema of snapshot events may have outdated defaults.</p>

Property	Default	Description
<code>incremental.snapshot.chunk.size</code>	<code>1024</code>	The maximum number of rows that the connector fetches and reads into memory during an incremental snapshot chunk. Increasing the chunk size provides greater efficiency, because the snapshot runs fewer snapshot queries of a greater size. However, larger chunk sizes also require more memory to buffer the snapshot data. Adjust the chunk size to a value that provides the best performance in your environment.
<code>read.only</code>	<code>false</code>	Switch to alternative incremental snapshot watermarks implementation to avoid writes to signal data collection
<code>provide.transaction.metadata</code>	<code>false</code>	Determines whether the connector generates events with transaction boundaries and enriches change event envelopes with transaction metadata. Specify <code>true</code> if you want the connector to do this. See <a href="#">Transaction metadata</a> for details.
<code>topic.naming.strategy</code>	<code>io.debezium.schema.DefaultTopicNamingStrategy</code>	The name of the TopicNamingStrategy class that should be used to determine the topic name for data change, schema change, transaction, heartbeat event etc., defaults to <code>DefaultTopicNamingStrategy</code> .
<code>topic.delimiter</code>	<code>.</code>	Specify the delimiter for topic name, defaults to <code>..</code>
<code>topic.prefix</code>	<code>\${database.server.name}</code>	<p>The name of the prefix to be used for all topics, defaults to <code>\${database.server.name}</code>.</p> <div>  <p>Once specify the prefix value to this property, the <code>\${database.server.name}</code> will not play the prefix role of all topics.</p> </div>
<code>topic.cache.size</code>	<code>10000</code>	The size used for holding the topic names in bounded concurrent hash map. This cache will help to determine the topic name corresponding to a given data collection.

Property	Default	Description
<code>topic.heartbeat.prefix</code>	<code>__debezium-heartbeat</code>	<p>Controls the name of the topic to which the connector sends heartbeat messages. The topic name has this pattern:</p> <p><i>topic.heartbeat.prefix.topic.prefix</i></p> <p>For example, if the database server name or topic prefix is <code>fulfillment</code>, the default topic name is <code>__debezium-heartbeat.fulfillment</code>.</p>
<code>topic.transaction</code>	<code>transaction</code>	<p>Controls the name of the topic to which the connector sends transaction metadata messages. The topic name has this pattern:</p> <p><i>topic.prefix.topic.transaction</i></p> <p>For example, if the database server name or topic prefix is <code>fulfillment</code>, the default topic name is <code>fulfillment.transaction</code>.</p>

## {prodname} connector database history configuration properties

{prodname} provides a set of `database.history.*` properties that control how the connector interacts with the schema history topic.

The following table describes the `database.history` properties for configuring the {prodname} connector.

Table 26. Connector database history configuration properties

Property	Default	Description
<code>database.history.kafka.topic</code>		The full name of the Kafka topic where the connector stores the database schema history.
<code>database.history.kafka.bootstrap.servers</code>		A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. This connection is used for retrieving the database schema history previously stored by the connector, and for writing each DDL statement read from the source database. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
<code>database.history.kafka.recovery.poll.interval.ms</code>	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.

Property	Default	Description
<code>database.history.kafka.query.timeout.ms</code>	<code>3000</code>	An integer value that specifies the maximum number of milliseconds the connector should wait while fetching cluster information using Kafka admin client.
<code>database.history.kafka.create.timeout.ms</code>	<code>30000</code>	An integer value that specifies the maximum number of milliseconds the connector should wait while create kafka history topic using Kafka admin client.
<code>database.history.kafka.recovery.attempts</code>	<code>4</code>	The maximum number of times that the connector should try to read persisted history data before the connector recovery fails with an error. The maximum amount of time to wait after receiving no data is <code>recovery.attempts x recovery.poll.interval.ms</code> .
<code>database.history.skip.unparseable.ddl</code>	<code>false</code>	A Boolean value that specifies whether the connector should ignore malformed or unknown database statements or stop processing so a human can fix the issue. The safe default is <code>false</code> . Skipping should be used only with care as it can lead to data loss or mangling when the binlog is being processed.
<code>database.history.store.only.captured.tables.ddl</code>	<code>false</code>	<p>A Boolean value that specifies whether the connector should record all DDL statements</p> <p><code>true</code> records only those DDL statements that are relevant to tables whose changes are being captured by {prodname}. Set to <code>true</code> with care because missing data might become necessary if you change which tables have their changes captured.</p> <p>The safe default is <code>false</code>.</p>

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

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

```
database.history.producer.security.protocol=SSL
database.history.producer.ssl.keystore.location=/var/private/ssl/kafka.server.keystore
```

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

{prodname} strips the prefix from the property name before it passes the property to the Kafka client.

See the Kafka documentation for more details about [Kafka producer configuration properties](#) and [Kafka consumer configuration properties](#).

## {prodname} connector Kafka signals configuration properties

When the MySQL connector is configured as read-only, the alternative for the signaling table is the signals Kafka topic.

{prodname} provides a set of **signal.\*** properties that control how the connector interacts with the Kafka signals topic.

The following table describes the **signal** properties.

*Table 27. Kafka signals configuration properties*

Property	Default	Description
<b>signal.kafka.topic</b>		The name of the Kafka topic that the connector monitors for ad hoc signals.
<b>signal.kafka.bootstrap.servers</b>		A list of host/port pairs that the connector uses for establishing an initial connection to the Kafka cluster. Each pair should point to the same Kafka cluster used by the Kafka Connect process.
<b>signal.kafka.poll.timeout.ms</b>	100	An integer value that specifies the maximum number of milliseconds the connector should wait when polling signals. The default is 100ms.

## **{prodname} connector pass-through signals Kafka consumer client configuration properties**

The {prodname} connector provides for pass-through configuration of the signals Kafka consumer. Pass-through signals properties begin with the prefix `signals.consumer.*`. For example, the connector passes properties such as `signal.consumer.security.protocol=SSL` to the Kafka consumer.

As is the case with the [pass-through properties for database history clients](#), {prodname} strips the prefixes from the properties before it passes them to the Kafka signals consumer.

## **{prodname} connector pass-through database driver configuration properties**

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

As is the case with the [pass-through properties for database history clients](#), {prodname} strips the prefixes from the properties before it passes them to the database driver.

# Monitoring

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

- [Snapshot metrics](#) provide information about connector operation while performing a snapshot.
- [Streaming metrics](#) provide information about connector operation when the connector is reading the binlog.
- [Schema history metrics](#) provide information about the status of the connector's schema history.

[{prodname} monitoring documentation](#) provides details for how to expose these metrics by using JMX.

## Snapshot metrics

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

Snapshot metrics are not exposed unless a snapshot operation is active, or if a snapshot has occurred since the last connector start.

The following table lists the shapshot metrics that are available.

Attributes	Type	Description
<code>LastEvent</code>	<code>string</code>	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 include/exclude list filtering rules configured on the connector.
CapturedTables	string[]	The list of tables that are captured by the connector.
QueueTotalCapacity	int	The length the queue used to pass events between the snapshotter and the main Kafka Connect loop.
QueueRemainingCapacity	int	The free capacity of the queue used to pass events between the snapshotter and the main Kafka Connect loop.
TotalTableCount	int	The total number of tables that are being included in the snapshot.
RemainingTableCount	int	The number of tables that the snapshot has yet to copy.
SnapshotRunning	boolean	Whether the snapshot was started.
SnapshotPaused	boolean	Whether the snapshot was paused.
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. Includes also time when snapshot was paused.

Attributes	Type	Description
SnapshotPausedDurationInSeconds	long	The total number of seconds that the snapshot was paused. If the snapshot was paused several times, the paused time adds up.
RowsScanned	Map<String, Long>	Map containing the number of rows scanned for each table in the snapshot. Tables are incrementally added to the Map during processing. Updates every 10,000 rows scanned and upon completing a table.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if <code>max.queue.size.in.bytes</code> is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

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

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

The {prodname} MySQL connector also provides the `HoldingGlobalLock` custom snapshot metric. This metric is set to a Boolean value that indicates whether the connector currently holds a global

or table write lock.

## Streaming metrics

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

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

The following table lists the streaming metrics that are available.

Attributes	Type	Description
<code>LastEvent</code>	<code>string</code>	The last streaming event that the connector has read.
<code>MillisecondsSinceLastEvent</code>	<code>long</code>	The number of milliseconds since the connector has read and processed the most recent event.
<code>TotalNumberOfEventsSeen</code>	<code>long</code>	The total number of events that this connector has seen since the last start or metrics reset.
<code>TotalNumberOfCreateEventsSeen</code>	<code>long</code>	The total number of create events that this connector has seen since the last start or metrics reset.
<code>TotalNumberOfUpdateEventsSeen</code>	<code>long</code>	The total number of update events that this connector has seen since the last start or metrics reset.
<code>TotalNumberOfDeleteEventsSeen</code>	<code>long</code>	The total number of delete events that this connector has seen since the last start or metrics reset.
<code>NumberOfEventsFiltered</code>	<code>long</code>	The number of events that have been filtered by include/exclude list filtering rules configured on the connector.
<code>CapturedTables</code>	<code>string[]</code>	The list of tables that are captured by the connector.

Attributes	Type	Description
QueueTotalCapacity	int	The length the queue used to pass events between the streamer and the main Kafka Connect loop.
QueueRemainingCapacity	int	The free capacity of the queue used to pass events between the streamer and the main Kafka Connect loop.
Connected	boolean	Flag that denotes whether the connector is currently connected to the database server.
MilliSecondsBehindSource	long	The number of milliseconds between the last change event's timestamp and the connector processing it. The values will incorporate any differences between the clocks on the machines where the database server and the connector are running.
NumberOfCommittedTransactions	long	The number of processed transactions that were committed.
SourceEventPosition	Map<String, String>	The coordinates of the last received event.
LastTransactionId	string	Transaction identifier of the last processed transaction.
MaxQueueSizeInBytes	long	The maximum buffer of the queue in bytes. This metric is available if <code>max.queue.size.in.bytes</code> is set to a positive long value.
CurrentQueueSizeInBytes	long	The current volume, in bytes, of records in the queue.

The {prodname} MySQL connector also provides the following additional streaming metrics:

*Table 28. Descriptions of additional streaming metrics*

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

## Schema history metrics

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

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

Attributes	Type	Description
Status	string	One of <b>STOPPED</b> , <b>RECOVERING</b> (recovering history from the storage), <b>RUNNING</b> describing the state of the database history.

Attributes	Type	Description
<code>RecoveryStartTime</code>	<code>long</code>	The time in epoch seconds at what recovery has started.
<code>ChangesRecovered</code>	<code>long</code>	The number of changes that were read during recovery phase.
<code>ChangesApplied</code>	<code>long</code>	the total number of schema changes applied during recovery and runtime.
<code>MillisecondsSinceLastRecoveredChange</code>	<code>long</code>	The number of milliseconds that elapsed since the last change was recovered from the history store.
<code>MillisecondsSinceLastAppliedChange</code>	<code>long</code>	The number of milliseconds that elapsed since the last change was applied.
<code>LastRecoveredChange</code>	<code>string</code>	The string representation of the last change recovered from the history store.
<code>LastAppliedChange</code>	<code>string</code>	The string representation of the last applied change.

## Behavior when things go wrong

{prodname} 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 {prodname} 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, {prodname}, like Kafka, provides *at least once* delivery of change events.

The rest of this section describes how {prodname} handles various kinds of faults and problems.

## Configuration and startup errors

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

- The connector's configuration is invalid.
- The connector cannot successfully connect to the MySQL server by using the specified connection parameters.
- The connector is attempting to restart at a position in the binlog for which MySQL no longer has

the history available.

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

## MySQL becomes unavailable

If your MySQL server becomes unavailable, the {prodname} MySQL connector fails with an error and the connector stops. When the server is available again, restart the connector.

However, if GTIDs are enabled for a highly available MySQL cluster, you can restart the connector immediately. It will connect to a different MySQL server in the cluster, find the location in the server's binlog that represents the last transaction, and begin reading the new server's binlog from that specific location.

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

## Kafka Connect stops gracefully

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

## Kafka Connect process crashes

If Kafka Connect crashes, the process stops and any {prodname} MySQL connector tasks terminate without their most recently-processed offsets being recorded. In distributed mode, Kafka Connect restarts the connector tasks on other processes. However, the MySQL connector resumes from the last offset recorded by the earlier processes. This means that the replacement tasks might generate some of the same events processed prior to the crash, creating duplicate events.

Each change event message includes source-specific information that you can use to identify duplicate events, for example:

- Event origin
- MySQL server's event time
- The binlog file name and position
- GTIDs (if used)

## Kafka becomes unavailable

The Kafka Connect framework records {prodname} change events in Kafka by using the Kafka producer API. If the Kafka brokers become unavailable, the {prodname} MySQL connector pauses until the connection is reestablished and the connector resumes where it left off.

## MySQL purges binlog files

If the {prodname} MySQL connector stops for too long, the MySQL server purges older binlog files and the connector's last position may be lost. When the connector is restarted, the MySQL server no longer has the starting point and the connector performs another initial snapshot. If the snapshot is disabled, the connector fails with an error.

See [snapshots](#) for details about how MySQL connectors perform initial snapshots.