# Red Hat Integration 2020.q1

## Debezium ユーザーガイド

Debezium 1.0 での使用

Last Updated: 2023-02-28

# Red Hat Integration 2020.q1 Debezium ユーザーガイド

Debezium 1.0 での使用

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

## 法律上の通知



## 概要

本ガイドでは、Debezium で提供されるコネクターを使用する方法を説明します。

# 目次

# 第1章 MYSQL の DEBEZIUM コネクター

MySQL には、データベースにコミットされた順序ですべての操作を記録するバイナリーログ (binlog) があります。これには、テーブルスキーマとテーブル内のデータの変更が含まれます。MySQL はレプリケーションとリカバリーに binlog を使用します。

MySQL コネクターは binlog を読み取り、行レベルの **INSERT**、**UPDATE**、**DELETE** 操作の変更イベントを生成し、Kafka トピックに変更イベントを記録します。クライアントアプリケーションはこれらの Kafka トピックを読み取ります。

MySQL は通常、指定された期間後に binlogs をパージするように設定されているため、MySQL コネクターは各データベースの 整合性スナップショット を実行し、初期 **整合性スナップショット** を実行します。MySQL コネクターは、スナップショットが作成された時点から binlog を読み取ります。

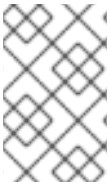
## 1.1. MYSQL コネクターの仕組みの概要

Debezium MySQL コネクターはテーブルの構造を追跡し、スナップショットを実行し、binlog イベントを Debezium 変更イベントに変換し、これらのイベントが Kafka に記録されます。

### 1.1.1. MySQL コネクターによるデータベーススキーマの使用方法

データベースクライアントがデータベースをクエリーすると、データベースの現在のスキーマが使用されます。データベーススキーマが頻繁に変更されると、Debezium **MySQL コネクター** は **INSERT**、**UPDATE**、および **DELETE** 操作ごとにスキーマがどのように表示されるかを認識します。

MySQL には、各テーブルのスキーマのインメモリー表現を解析し、更新する binlog の **行レベルの変更** と **DDL ステートメント** の両方が含まれます。これは、正確な変更イベントを生成する各操作時にテーブル構造を理解するために使用されます。



**注記**

コネクターは、すべての DDL ステートメントと、別のデータベース履歴の位置を記録し、コネクターの再起動時(**クラッシュまたは正常なシャットダウンの後**)が、その特定の時点からの binlog の読み取りを続行します。

**ヒント**

トピックの命名規則の詳細 は、MySQL コネクターおよび Kafka トピック を参照してください。

**関連情報**

- ここで説明する **データベース履歴トピック** を使用しない場合は、スキーマ変更トピック を参照してください。

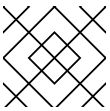
### 1.1.2. MySQL コネクターによるデータベーススナップショットの実行方法

Debezium MySQL コネクターが最初に起動すると、データベースの最初の **整合性スナップショット** が実行されます。以下のフローは、このスナップショットの完了方法を説明します。

**注記**

これは、snapshot.mode プロパティーの **初期** として設定されるデフォルトの **スナップショットモード** です。他のスナップショットモードについては、MySQL コネクター設定プロパティー を確認してください。

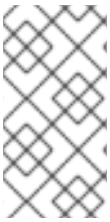
**コネクター**

| Step | アクション |
|---|---|
| 1 | 他のデータベースクライアントによる **書き込み** をブロックする **グローバル読み取りロック** を取得します。<br><br>  <br>**注記**<br>スナップショット自体は、コネクターの binlog の位置およびテーブルスキーマの読み取りを干渉する可能性のある DDL の適用を防ぐことはありません。グローバル読み取りロックは、後のステップでリリースする前に binlog の位置が読み取られている間に保持されます。 |
| 2 | 繰り返し可能な読み取りセマンティクス でトランザクションを開始し、トランザクション内の後続の読み取りがすべて **整合性スナップショット** に対して実行されるようにします。 |
| 3 | 現在の binlog の位置を読み取ります。 |
| 4 | コネクターの設定によって許可されるデータベースおよびテーブルのスキーマを読み取ります。 |
| 5 | **グローバル読み取りロック** を解放します。これにより、他のデータベースクライアントがデータベースに書き込みできるようになりました。 |
| 6 | DDL の変更をスキーマ変更トピックに書き込みます。これには、必要な **DROP...** および **CREATE...** DDL ステートメントがすべて含まれます。<br><br>  <br>**注記**<br>これは、該当する場合に発生します。 |
| 7 | データベーステーブルをスキャンし、各行に関連するテーブル固有の Kafka トピックで **CREATE** イベントを生成します。 |
| 8 | トランザクションをコミットします。 |
| 9 | コネクターオフセットの完了済みスナップショットを記録します。 |

#### 1.1.2.1. コネクターが失敗するとどうなりますか？

**最初のスナップ** ショットの実行中にコネクターが失敗、停止、またはリバランスされると、コネクターは再起動後に新しいスナップショットを作成します。この意図的な **スナップショット** が完了すると、Debezium MySQL コネクターは binlog の同じ位置から再起動するため、更新が見逃されることはあり

ません。



**注記**

コネクターが長時間停止した場合、MySQL が古い binlog ファイルをパージし、コネクターの位置が失われる可能性があります。位置が失われた場合、コネクターは **最初のスナップショット** を開始位置に戻します。Debezium MySQL コネクターのトラブルシューティングに関する詳細は、MySQL connector common issues を参照してください。

#### 1.1.2.2. グローバル読み取りロックが許可されていない場合はどうすればよいですか？

一部の環境では、**グローバル読み取りロック** が許可されません。Debezium MySQL コネクターがグローバル読み取りロックが許可されないことを検出すると、代わりにテーブルレベルロックを使用して、この方法でスナップショットを実行します。

**重要**

ユーザーが **LOCK_TABLES** 権限を持っている必要があります。

**コネクター**

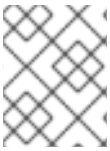
| Step | アクション |
|---|---|
| 1 | 繰り返し可能な読み取りセマンティクス でトランザクションを開始し、トランザクション内の後続の読み取りがすべて **整合性スナップショット** に対して実行されるようにします。 |
| 2 | データベースとテーブルの名前を読み取り、選別します。 |
| 3 | 現在の binlog の位置を読み取ります。 |
| 4 | コネクターの設定によって許可されるデータベースおよびテーブルのスキーマを読み取ります。 |
| 5 | DDL の変更をスキーマ変更トピックに書き込みます。これには、必要な **DROP...** および **CREATE...** DDL ステートメントがすべて含まれます。<br>  <br>**注記**<br>これは、該当する場合に発生します。 |
| 6 | データベーステーブルをスキャンし、各行に関連するテーブル固有の Kafka トピックで **CREATE** イベントを生成します。 |
| 7 | トランザクションをコミットします。 |
| 8 | テーブルレベルロックを解除します。 |
| 9 | コネクターオフセットの完了済みスナップショットを記録します。 |

### 1.1.3. MySQL コネクターがスキーマ変更トピックを処理する方法

Debezium **MySQL コネクターを設定すると、MySQL** サーバーのデータベースに適用されるすべての DDL ステートメントが含まれるスキーマ変更イベントを生成できます。コネクターは、これらのイベントをすべて < **serverName** > という名前の Kafka トピックに書き込みます。 **serverName** は **database.server.name** 設定プロパティーに指定されたコネクターの名前になります。

**重要**

**スキーマ変更イベント** を使用する場合は、スキーマ変更トピックを使用し、データベース履歴トピックを使用し **ません**。



**注記**

スキーマの変更が正しい順序で保持されるように、Kafka の **num.partitions** 設定が **1** に設定されていることを確認してください。

#### 1.1.3.1. スキーマ変更トピック構造

スキーマ変更トピックに書き込まれる各メッセージには、DDL ステートメントの適用時に使用される接続されたデータベースの名前が含まれるメッセージキーが含まれます。

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

スキーマ変更イベントメッセージの値には、DDL ステートメント、ステートメントが適用されるデータベース、およびステートメントが現れる binlog の位置が含まれる構造が含まれます。

```
{
  "schema": {
    "type": "struct",
    "name": "io.debezium.connector.mysql.SchemaChangeValue",
    "optional": false,
    "fields": [
      {
        "field": "databaseName",
        "type": "string",
        "optional": false
      },
```

```
{
  "field": "ddl",
  "type": "string",
  "optional": false
},
{
  "field": "source",
  "type": "struct",
  "name": "io.debezium.connector.mysql.Source",
  "optional": false,
  "fields": [
    {
      "type": "string",
      "optional": true,
      "field": "version"
    },
    {
      "type": "string",
      "optional": false,
      "field": "name"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "server_id"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "ts_sec"
    },
    {
      "type": "string",
      "optional": true,
      "field": "gtid"
    },
    {
      "type": "string",
      "optional": false,
      "field": "file"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "pos"
    },
    {
      "type": "int32",
      "optional": false,
      "field": "row"
    },
    {
      "type": "boolean",
      "optional": true,
      "default": false,
      "field": "snapshot"
```

```
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          },
          {
            "type": "string",
            "optional": true,
            "field": "query"
          }
        ]
      }
    ]
  },
  "payload": {
    "databaseName": "inventory",
    "ddl": "CREATE TABLE products ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255) NOT NULL, description VARCHAR(512), weight FLOAT ); ALTER TABLE
products AUTO_INCREMENT = 101;",
    "source" : {
      "version": "0.10.0.Beta4",
      "name": "mysql-server-1",
      "server_id": 0,
      "ts_sec": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "snapshot": true,
      "thread": null,
      "db": null,
      "table": null,
      "query": null
    }
  }
}
```


#### 1.1.3.1.1. スキーマ変更トピックに関する重要なヒント

**ddl** フィールドには複数の DDL ステートメントが含まれる場合があります。すべてのステートメントは **databaseName** フィールドのデータベースに適用され、データベースに適用されるのと同じ順序で表示されます。**source** フィールドは、テーブル固有のトピックに書き込まれた標準のデータ変更イベントとして設定されます。このフィールドは、異なるトピックでイベントを関連付けるのに役立ちます。

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

**クライアントが DDL ステートメントを複数のデータベース に送信する場合**

- MySQL がこれらをアトミックに適用する場合、コネクターは DDL ステートメントを順番に取得し、データベース別にグループ化して、各グループにスキーマ変更イベントを作成します。
- MySQL がこれらを個別に適用すると、コネクターは各ステートメントに対して個別のスキーマ変更イベントを作成します。

**関連情報**

- ここで説明した **スキーマ変更トピック** を使用しない場合は、データベース履歴トピック を確認してください。

### 1.1.4. MySQL コネクターイベント

Debezium MySQL コネクターによって生成されたすべてのデータ変更イベントには、キーと値が含まれます。変更イベントキーと変更イベント値には、それぞれ **スキーマ** と **ペイロードが含まれ、スキーマはペイロード** の構造を記述し、ペイロードにはデータが含まれます。

**警告**

MySQL コネクターは、すべての Kafka Connect スキーマ名が Avro スキーマ名の形式 に準拠するようにします。これは、文字やアンダースコアではない文字がアンダースコアに置き換えられます。これにより、論理サーバー名、データベース名、およびテーブル名コンテナー（これらのアンダースコアに置き換えられたその他の文字）時にスキーマ名で予期しない競合が発生する可能性があります。

#### 1.1.4.1. 変更イベントキー

指定のテーブルでは、変更イベントのキーには、イベントの作成時に **プライマリーRY KEY** （または一意の制約）の各列のフィールドが含まれる構造があります。サンプルのテーブルと、そのテーブルのスキーマとペイロードがどのように表示されるかを見てみましょう。

**テーブルの例**

```
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;
```

**変更イベントキーの例**

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

1. **スキーマ** は、**ペイロード** の内容を記述します。
2. **mysql-server- 1.inventory.customers.Key は、mysql-server-**1 がコネクター名、**inventory** はデータベース、**customers** がテーブルである構造を定義するスキーマの名前です。
3. **ペイロード** はオプションではないことを示します。
4. **ペイロード** で想定されるフィールドのタイプを指定します。
5. ペイロード自体。この場合、単一の **id** フィールドのみが含まれます。

このキーは、**id** プライマリーキー列の値が **1001** である **mysql-server-1** のコネクターから外れる **inventory.customers** テーブルの行を記述します。

#### 1.1.4.2. 変更イベント値

変更イベント値には、schema および payload セクションが含まれます。エンベロープ構造を持つ変更イベント値には 3 つのタイプがあります。この構造のフィールドについては以下に説明され、各変更イベント値の例でマークが付けられます。

- 「変更イベント値の作成」
- 「変更イベント値の更新」
- 「変更イベント値の削除」

| 項目 | フィールド名 | 説明 |
|---|---|---|
| 1 | **name** | **mysql-server- 1.inventory.customers.Key は、 mysql-server-**1 がコネクター名、**inventory** はデータベース、**customers** がテーブルである構造を定義するスキーマの名前です。 |
| 2 | **op** | 操作のタイプを記述する **必須** 文字列。<br><br>**値**<br>● **c** = create<br>● **u** = update<br>● **d** = delete<br>● **r** = read (**最初のスナップショット** のみ) |
| 3 | **before** | イベント発生前の行の状態を指定する任意のフィールド。 |
| 4 | **after** | イベント発生後の行の状態を指定する任意のフィールド。 |
| 5 | **source** | 以下を含むイベントのソースメタデータを記述する **必須** フィールド。<br>● Debezium のバージョン<br>● コネクター名<br>● イベントが記録された binlog 名<br>● binlog の位置<br>● イベント内の行<br>● イベントがスナップショットの一部である場合<br>● 影響を受けるデータベースおよびテーブルの名前<br>● イベントを作成する MySQL スレッドの ID （スナップショット以外）<br>● MySQL サーバー ID （利用可能な場合）<br>● timestamp<br><br>  <br>**注記**<br>binlog_rows_query_log_events オプションが有効で、コネクターの **include.query** オプションが有効になっている場合、イベントを生成した元の SQL ステートメントが含まれる **クエリー** フィールドが表示されます。 |

| 項目 | フィールド名 | 説明 |
| --- | --- | --- |
| 6 | **ts_ms** | コネクターがイベントを処理した時間を表示する任意のフィールド。<br><br>**注記**<br><br>この時間は、Kafka Connect タスクを実行している JVM のシステムクロックを基にします。 |

サンプルのテーブルと、そのテーブルのスキーマとペイロードがどのように表示されるかを見てみましょう。

**テーブルの例**

```
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;
```

#### 1.1.4.2.1. 変更イベント値の作成

以下の例は、**customers** テーブルの **作成** イベントを示しています。

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

```
    ],
    "optional": true,
    "name": "mysql-server-1.inventory.customers.Value",
    "field": "before"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "id"
      },
      {
        "type": "string",
        "optional": false,
        "field": "first_name"
      },
      {
        "type": "string",
        "optional": false,
        "field": "last_name"
      },
      {
        "type": "string",
        "optional": false,
        "field": "email"
      }
    ],
    "optional": true,
    "name": "mysql-server-1.inventory.customers.Value",
    "field": "after"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "version"
      },
      {
        "type": "string",
        "optional": false,
        "field": "connector"
      },
      {
        "type": "string",
        "optional": false,
        "field": "name"
      },
      {
        "type": "int64",
        "optional": false,
        "field": "ts_ms"
      },
```

```
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "string",
          "optional": false,
          "field": "db"
        },
        {
          "type": "string",
          "optional": true,
          "field": "table"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "server_id"
        },
        {
          "type": "string",
          "optional": true,
          "field": "gtid"
        },
        {
          "type": "string",
          "optional": false,
          "field": "file"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "pos"
        },
        {
          "type": "int32",
          "optional": false,
          "field": "row"
        },
        {
          "type": "int64",
          "optional": true,
          "field": "thread"
        },
        {
          "type": "string",
          "optional": true,
          "field": "query"
        }
      ],
      "optional": false,
      "name": "io.product.connector.mysql.Source",
      "field": "source"
    },
```

```
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mysql-server-1.inventory.customers.Envelope"
  },
  "payload": { 2
    "op": "c",
    "ts_ms": 1465491411815,
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.0.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
    }
  }
}
```

1 このイベントの **値** の **schema** 部分は、エンベロープのスキーマ、ソース構造のスキーマ(MySQL コネクターに固有ですべてのイベントで再利用)、**before** および **after** フィールドのテーブル固有のスキーマを表示します。

2 このイベントの **値** の **payload** 部分には、イベント内の情報、行が作成されたことを記述している ( **op=c**であるため)、**after** フィールドの値には新しい挿入された行の **id**、**first_name**、**last_name**、および **email** 列の値が含まれていることを表しています。

#### 1.1.4.2.2. 変更イベント値の更新

**customers** テーブルの **更新** 変更イベントの値には、**作成** イベントと同じスキーマがあります。ペイロードは同じように設定されますが、異なる値を保持します。以下に例を示します（書式を調整して読みやすくしてあります）。

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { 2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 3
      "version": "1.0.3.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u", 4
    "ts_ms": 1465581029523 5
  }
}
```

これを **挿入** イベントの値と比較すると、**payload** セクションにいくつかの違いがあります。

1. **before** フィールドは、データベースのコミット前の行と値の状態を表しています。
2. **after** フィールドは、更新された行の状態を表し、**first_name** の値は **Anne Marie** になっています。before と **after** の構造を比較 **する** と、コミットが原因で、この行で実際に何が変更されたかを判断できます。
3. **source** フィールド構造には以前と同じフィールドがありますが、値は異なります（このイベントは binlog の異なる位置にあります）。**source** 構造には、この変更の MySQL レコードに関する情報が表示されます（トレーサビリティーを提供）。また、このトピックや他のトピックの他のイベントと比較し、他のイベントと同じ MySQL コミットの前、後、または一部でこのイベントが発生したかどうかを確認するための情報もあります。

4 **op** フィールドの値は **u** になっており、更新によってこの行が変更されたことを示しています。

5 **ts_ms** フィールドは、Debezium がこのイベントを処理したときのタイムスタンプを表示します。

**注記**

行のプライマリーキーまたは一意の鍵の列を更新すると、行のキーの値が変更され、Debezium は 3 つのイベントを出力します。つまり、行の古いキーを持つ **DELETE** イベントと tombstone イベント、行の新しいキーを持つ **INSERT** イベントです。

#### 1.1.4.2.3. 変更イベント値の削除

**customers** テーブルの **削除** 変更イベントの値には、**作成** および **更新** イベントと同じスキーマがあります。ペイロードは同じように設定されますが、異なる値を保持します。以下に例を示します（書式を調整して読みやすくしてあります）。

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, 2
    "source": { 3
      "version": "1.0.3.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d", 4
    "ts_ms": 1465581902461 5
  }
}
```

**作成** と **更新** イベントの **ペイロード** 部分をペイロードと比較すると、いくつかの違いがあります。

1 **before** フィールドは、データベースのコミットで削除した行の状態を表しています。

2 **after** フィールドは **null** で、行が存在しないことを示します。

3 **source** フィールド構造には以前と同じ値が多数ありますが、**ts_sec** および **pos** フィールドは変

更されました（他のシナリオでは ファイルが変更された可能性があります）。

4 **op** フィールドの値は **d** になっており、この行が削除されたことを示しています。

5 **ts_ms** は、Debezium がこのイベントを処理したときのタイムスタンプを示します。

このイベントは、この行の削除を処理するのに必要な情報をコンシューマーに提供します。コンシューマーによっては、削除を適切に処理するために古い値が必要になることがあるため、古い値が含まれます。

MySQL コネクターのイベントは、Kafka ログコンパクション と動作するように設計されています。これにより、すべてのキーの最新のメッセージが保持される限り、古いメッセージを削除できます。これにより、トピックに完全なデータセットが含まれ、キーベースの状態のリロードに使用できるようになりました。

行が **削除** された場合でも、Kafka は同じキーを持つ以前のメッセージをすべて削除できるため、上記の 削除 イベント値はログコンパクションで動作します。メッセージの値が **null** の場合、Kafka は同じキーを持つすべてのメッセージを削除できることを認識します。これを可能にするために、Debezium の MySQL コネクターは、**null** 値以外で同じキーを持つ特別な廃棄イベントを持つ **削除** イベントに従います。

### 1.1.5. MySQL コネクターによるデータ型のマッピング方法

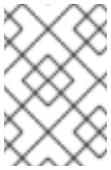
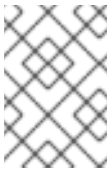
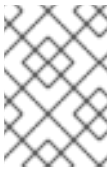
Debezium MySQL コネクターは、行が存在するテーブルのように構造化されたイベントで行への変更を表します。イベントには、各列の値のフィールドが含まれます。その列の MySQL データ型によって、その値がイベントでどのように表されるかが決まります。

文字列を格納する列は、文字セットと照合順序を使用して MySQL に定義されます。MySQL コネクターは、binlog イベントの列値のバイナリー表現を読み取るときに、列の文字セットを使用します。以下の表は、MySQL データ型を **リテラル** 型と **セマンティック型** の両方にマップする方法を示しています。

- **リテラル型** : Kafka Connect スキーマ型を使用して値がどのように表されるか。
- **セマンティック型** : Kafka Connect スキーマがどのようにフィールド（スキーマ名）の意味をキャプチャーするか。

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **BOOLEAN, BOOL** | **BOOLEAN** | **該当なし** |
| **BIT(1)** | **BOOLEAN** | **該当なし** |

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **BIT(>1)** | **BYTES** | io.debezium.data.Bits<br><br>**注記**<br><br>**length** スキーマパラメーターには、ビット数を表す整数が含まれます。**byte[]** にはビットが **リトルエンディアン** 形式で含まれ、指定数のビットが含まれるようにサイズが指定されます。<br><br>**例(n がビット)**<br><br>numBytes = n/8 + (n%8== 0 ? 0 : 1) |
| **TINYINT** | **INT16** | **該当なし** |
| **SMALLINT[(M)]** | **INT16** | **該当なし** |
| **MEDIUMINT[(M)]** | **INT32** | **該当なし** |
| **INT, INTEGER[(M)]** | **INT32** | **該当なし** |
| **BIGINT[(M)]** | **INT64** | **該当なし** |
| **REAL[(M,D)]** | **FLOAT32** | **該当なし** |
| **FLOAT[(M,D)]** | **FLOAT64** | **該当なし** |
| **DOUBLE[(M,D)]** | **FLOAT64** | **該当なし** |
| **CHAR(M)]** | **STRING** | **該当なし** |
| **VARCHAR(M)]** | **STRING** | **該当なし** |
| **BINARY(M)]** | **BYTES** | **該当なし** |
| **VARBINARY(M)]** | **BYTES** | **該当なし** |
| **TINYBLOB** | **BYTES** | **該当なし** |
| **TINYTEXT** | **STRING** | **該当なし** |
| **BLOB** | **BYTES** | **該当なし** |

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **TEXT** | **STRING** | **該当なし** |
| **MEDIUMBLOB** | **BYTES** | **該当なし** |
| **MEDIUMTEXT** | **STRING** | **該当なし** |
| **LONGBLOB** | **BYTES** | **該当なし** |
| **LONGTEXT** | **STRING** | **該当なし** |
| **JSON** | **STRING** | io.debezium.data.Json<br>  <br>**注記**<br>**JSON** ドキュメント、配列、またはスケーラーの文字列表現が含まれます。 |
| **ENUM** | **STRING** | io.debezium.data.Enum<br>  <br>**注記**<br>**許可されるスキーマパラメーターには、許可さ** れる値のコンマ区切りリストが含まれます。 |
| **SET** | **STRING** | io.debezium.data.EnumSet<br>  <br>**注記**<br>**許可されるスキーマパラメーターには、許可さ** れる値のコンマ区切りリストが含まれます。 |
| **YEAR[(2\|4)]** | **INT32** | io.debezium.time.Year |
| **TIMESTAMP[(M)]** | **STRING** | io.debezium.time.ZonedTimestamp<br>  <br>**注記**<br>マイクロ秒の精度を持つ ISO 8601 形式。<br>MySQL では、**M** を **0-6** の範囲にすることができます。 |

#### 1.1.5.1. 時間値

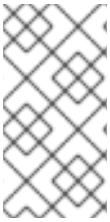
**TIMESTAMP** データ型を除き、MySQL の時間型は **time.precision.mode** 設定プロパティーの値によって異なります。

### ヒント

詳細は MySQL コネクター設定プロパティー を参照してください。

タイムゾーンのない時間値は、UTC からミリ秒またはマイクロ秒(**DATETIME**)または設定されたデータベースタイムゾーン(**TIMESTAMP**)に変換されます。

- 値が **2018-06-20 06:37:03** の **DATETIME** は、**1529476623000** になります。
- 値が **2018-06-20 06:37:03** の **TIMESTAMP** は **2018-06-20T13:37:03Z** になります。

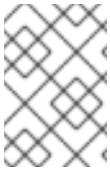


#### 注記

MySQL では、**DATE** DATE、**DATETIME**、および **TIMESTAMP** 列にゼロの値を使用できます。これは、null 値よりも優先されることがあります。ただし、MySQL コネクターは、列定義が null を許可する場合、または列が null を許可しない場合はエポック日として、それらを null 値として表します。

**time.precision.mode=adaptive_time_microseconds(default)**

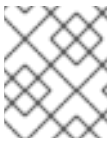
MySQL コネクターは、イベントがデータベースの値を正確に表すように、列のデータ型定義に基づいてリテラル型とセマンティック型を決定します。すべての時間フィールドはマイクロ秒です。

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **DATE** | **INT32** | io.debezium.time.Date<br>  <br>**注記**<br>エポックからの日数を表します。 |
| **TIME[(M)]** | **INT64** | io.debezium.time.MicroTime<br>  <br>**注記**<br>時間の値をマイクロ秒単位で表し、タイムゾーン情報は含まれません。MySQL では、**M** を **0-6** の範囲にすることができます。 |
| **DATETIME, DATETIME(0), DATETIME(1), DATETIME(2), DATETIME(3)** | **INT64** | io.debezium.time.Timestamp<br>  <br>**注記**<br>エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **DATETIME(4), DATETIME(5), DATETIME(6)** | **INT64** | io.debezium.time.MicroTimestamp<br>  <br>**注記**<br>エポックからの経過時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |

**time.precision.mode=connect**

MySQL コネクターは事前定義された Kafka Connect の論理型を使用します。この方法はデフォルトの方法よりも精度が低く、データベース列に **3** を超える **少数秒の精度**値がある場合は、イベントの精度が低くなる可能性があります。

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **DATE** | **INT32** | org.apache.kafka.connect.data.Date<br>  <br>**注記**<br>エポックからの日数を表します。 |
| **TIME[(M)]** | **INT64** | org.apache.kafka.connect.data.Time<br>  <br>**注記**<br>午前 0 時以降の時間値をマイクロ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME[(M)]** | **INT64** | org.apache.kafka.connect.data.Timestamp<br>  <br>**注記**<br>エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |

#### 1.1.5.2. 10 進数値

10 進数は **decimal.handling.mode** プロパティーで処理されます。

**ヒント**

詳細は MySQL コネクター設定プロパティー を参照してください。

**decimal.handling.mode=precise**

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **NUMERIC[(M[,D])]** | **BYTES** | org.apache.kafka.connect.data.Decimal<br>  <br>**注記**<br>**scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。 |
| **DECIMAL[(M[,D])]** | **BYTES** | org.apache.kafka.connect.data.Decimal<br>  <br>**注記**<br>**scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。 |

**decimal.handling.mode=double**

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **NUMERIC[(M[,D])]** | **FLOAT64** | **該当なし** |
| **DECIMAL[(M[,D])]** | **FLOAT64** | **該当なし** |

**decimal.handling.mode=string**

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **NUMERIC[(M[,D])]** | **STRING** | **該当なし** |
| **DECIMAL[(M[,D])]** | **STRING** | **該当なし** |

#### 1.1.5.3. 空間データ型

現在、Debezium MySQL コネクターは以下の空間データ型をサポートしています。

| MySQL 型 | リテラル型 | セマンティック型 |
|---|---|---|
| **GEOMETRY, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION** | **STRUCT** | io.debezium.data.geometry.Geometry<br><br>**注記**<br><br>2 つのフィールドを持つ構造が含まれます。<br><br>• **srid (INT32**: 構造に保存されたジオメトリーオブジェクトの型を定義する、空間参照システム ID)。<br>• **wkb (BYTES)**: Well-Known-Binary (wkb)形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。詳細は、Open Geospatial Consortium を参照してください。 |

### 1.1.6. MySQL コネクターおよび Kafka トピック

Debezium MySQL コネクターは、すべての **INSERT**、**UPDATE**、**DELETE** 操作のイベントを 1 つのテーブルから単一の Kafka トピックに書き込みます。Kafka トピックの命名規則は次のとおりです。

format

```
serverName.databaseName.tableName
```

**例1.1 例を示します。**

**fulfillment** がサーバー名、**inventory** には、**顧客**、および **製品** の 3 つ のテーブルが含まれるデータベースがあるとします。**Debezium MySQL コネクターは、データベースのテーブルごとに 1 つずつ、3 つの Kafka トピックでイベントを生成します。**

**fulfillment.inventory.orders**

**fulfillment.inventory.customers**

**fulfillment.inventory.products**

### 1.1.7. MySQL がサポートするトポロジー

**Debezium MySQL コネクターは以下の MySQL トポロジーをサポートします。**

スタンドアロン

**単一の MySQL サーバーを使用する場合は、Debezium MySQL コネクターがサーバーを監視できるように、binlog を有効 (および任意で GTID を有効) にする必要があります。バイナリーログも増分 バックアップ として使用できるため、これは多くの場合で許容されます。この場合、**

MySQL コネクターは常にこのスタンドアロン MySQL サーバーインスタンスに接続し、それに従います。

マスターおよびスレーブ

Debezium MySQL コネクターはマスターまたはスレーブの 1 つ（スレーブのbinlog が有効になっている場合）に従うことができますが、コネクターはそのサーバーに表示されるクラスターの変更のみを確認します。通常、これはマルチマスタートポロジー以外の問題ではありません。

コネクターは、サーバーの binlog の位置を記録します。この位置は、クラスターの各サーバーごとに異なります。そのため、コネクターは 1 つの MySQL サーバーインスタンスのみに従う必要があります。そのサーバーに障害が発生した場合は、コネクターを続行する前に再起動または復元する必要があります。

高可用性クラスター

MySQL にはさまざまな 高可用性 ソリューションがあり、問題や障害からほぼ簡単に回復できます。ほとんどの HA MySQL クラスターは GTID を使用するため、スレーブはいずれかのマスター上のすべての変更を追跡できます。

multi-master

マルチマスター MySQL トポロジー は、複数のマスターからそれぞれを複製する 1 つ以上の MySQL スレーブ を使用します。これは、複数の MySQL クラスターのレプリケーションを集約する強力な方法であり、GTID を使用する必要があります。

Debezium MySQL コネクターはこれらのマルチマスター MySQL スレーブをソースとして使用し、新しいスレーブが古いスレーブにキャッチされている限り、異なるマルチマスター MySQL スレーブにフェイルオーバーできます（たとえば、新しいスレーブには最初のスレーブで最後に確認されたすべてのトランザクションがあります）。これは、新しいマルチマスター MySQL スレーブへの再接続を試み、binlog で正しい場所を見つけようとする際に、特定の GTID ソースを追加または除外するようにコネクターを設定することができるため、コネクターがデータベースやテーブルのサブセットのみを使用している場合でも機能します。

ホステッド

Debezium MySQL コネクターが Amazon RDS や Amazon Aurora などのホステッドオプションを使用するためのサポートがあります。

重要

これらのホストオプションは グローバル読み取り ロックを許可しないため、テーブルレベルのロックを使用して 整合性スナップショット を作成します。

## 1.2. MYSQL サーバーの設定

### 1.2.1. Debezium の MySQL ユーザーの作成

Debezium MySQL コネクターが監視するすべてのデータベースに対して、適切なパーミッションで MySQL ユーザーを定義する必要があります。

**前提条件**

- MySQL サーバーが必要です。
- 基本的な SQL コマンドを知っている必要があります。

**手順**

1. MySQL ユーザーを作成します。

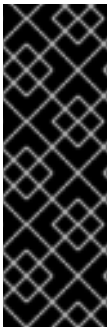
```
mysql> CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';
```

2. 必要なパーミッションをユーザーに付与します。

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

**ヒント**

各 パーミッションの注記については、説明 されているパーミッションを参照してください。



**重要**

グローバル読み取り ロックを許可しない Amazon RDS や Amazon Aurora などのホストオプションを使用する場合は、テーブルレベルのロックを使用して 整合性スナップショット を作成します。この場合は、作成するユーザーに LOCK_TABLES パーミッションも付与する必要があります。詳細は MySQL コネクターの仕組みの概要 を参照してください。

3. **ユーザーのパーミッションの最終処理を行います。**

```
mysql> FLUSH PRIVILEGES;
```

#### 1.2.1.1. パーミッションの説明

| permission/item | 説明 |
| --- | --- |
| **SELECT** | コネクターがデータベースのテーブルから行を選択できるようにする<br><br>  <br>**注記**<br>これは、スナップショットを実行する場合にのみ使用されます。 |
| **RELOAD** | 内部キャッシュのクリアまたはリロード、テーブルのフラッシュ、またはロックの取得を行う **FLUSH** ステートメントをコネクターが使用できるようにします。<br><br>  <br>**注記**<br>これは、スナップショットを実行する場合にのみ使用されます。 |
| **SHOW DATABASES** | **SHOW DATABASE** ステートメントを実行して、コネクターがデータベース名を確認できるようにします。<br><br>  <br>**注記**<br>これは、スナップショットを実行する場合にのみ使用されます。 |
| **REPLICATION-SLAVE** | コネクターが MySQL サーバーの binlog に接続し、読み取りできるようにします。 |
| **REPLICATION CLIENT** | コネクターが以下のステートメントを使用できるようにします。<br>• **SHOW MASTER STATUS**<br>• **SHOW SLAVE STATUS**<br>• **SHOW BINARY LOGS**<br><br>  <br>**重要**<br>これは常にコネクターに必要です。 |

| permission/item | 説明 |
| --- | --- |
| **ON** | パーミッションが適用される **データベース** を識別します。 |
| **TO 'user'** | パーミッションが付与される **ユーザー** を指定します。 |
| **IDENTIFIED BY 'password'** | ユーザーの **パスワード** を指定します。 |

### 1.2.2. Debezium の MySQL binlog の有効化

**MySQL レプリケーションのバイナリーロギングを有効にする必要があります。バイナリーログは、変更を伝播するためにレプリケーションツールのトランザクション更新を記録します。**

**前提条件**

- **MySQL サーバーが必要です。**
- **適切な MySQL ユーザーの権限が必要です。**

**手順**

1. **log-bin オプションがすでにオンかどうかを確認します。**

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

2. **OFF の場合は、以下のように MySQL サーバー設定ファイルを設定します。**

**ヒント**

**各プロパティーの注記については、ブログ 設定 プロパティー を参照してください。**

```
server-id        = 223344 (1)
log_bin          = mysql-bin (2)
binlog_format    = ROW (3)
binlog_row_image = FULL (4)
expire_logs_days = 10 (5)
```

3. **もう一度 binlog ステータスをチェックして、変更を確認します。**

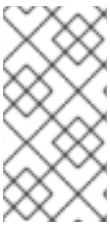
```
mysql> SELECT variable_value as "BINARY LOGGING STATUS (log-bin) ::"
FROM information_schema.global_variables WHERE variable_name='log_bin';
```

#### 1.2.2.1. binlog 設定プロパティー

| 数値 | プロパティー | 説明 |
|---|---|---|
| 1 | **server-id** | **server-id** の値は、MySQL クラスター内の各サーバーおよびレプリケーションクライアントに対して一意である必要があります。MySQL コネクターの設定時に、コネクターに一意のサーバー ID を割り当てます。 |
| 2 | **log_bin** | **log_bin** の値は、binlog ファイルのシーケンスのベース名です。 |
| 3 | **binlog_format** | **binlog-format** は **ROW** または **row** に設定する必要があります。 |
| 4 | **binlog_row_image** | **binlog_row_image** は **FULL** または **full** に設定する必要があります。 |
| 5 | **expire_logs_days** | これは、binlog ファイルが自動的に削除される日数です。デフォルトは **0** で、自動的に削除されません。<br>**ヒント**<br>実際の環境に見合った値を設定します。 |

### 1.2.3. Debezium の MySQL グローバルトランザクション識別子の有効化

**グローバルトランザクション識別子 (GTID) は、クラスター内のサーバーで発生するトランザクションを一意に識別します。Debezium MySQL コネクターには必要ありませんが、GTID を使用するとレプリケーションが簡素化され、マスターサーバーとスレーブサーバーの一貫性が保たれるかどうかをより簡単に確認することができます。**



**注記**

**GTID は MySQL 5.6.5 以降でのみ利用できます。詳細は MySQL のドキュメント を参照してください。**

**前提条件**

- **MySQL サーバーが必要です。**
- **基本的な SQL コマンドを知っている必要があります。**
- **MySQL 設定ファイルにアクセスできる必要があります。**

**手順**

1. **gtid_mode を有効にします。**

```
mysql> gtid_mode=ON
```

2. **enforce_gtid_consistency を有効にします。**

```
mysql> enforce_gtid_consistency=ON
```

3. **変更を確認します。**

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

**response**

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

#### 1.2.3.1. オプションの説明

| permission/item | 説明 |
| --- | --- |

| permission/item | 説明 |
| --- | --- |
| **gtid_mode** | MySQL サーバーの GTID モードが有効かどうかを指定するブール値。<br>● **ON** = 有効化<br>● **OFF** = 無効化 |
| **enforce_gtid_consistency** | トランザクション的に安全な方法でログインできるステートメントの実行を許可することにより、GTID の一貫性を有効にするかどうかをサーバーに指示するブール値。GTID を使用する際に必要です。<br>● **ON** = 有効化<br>● **OFF** = 無効化 |

### 1.2.4. Debezium のセッションタイムアウトの設定

**大規模なデータベースに対して最初の整合性スナップショットが作成されると、テーブルの読み込み時に、確立された接続がタイムアウトする可能性があります。MySQL 設定ファイルで interactive_timeout と wait_timeout を設定すると、この動作の発生を防ぐことができます。**

**前提条件**

- **MySQL サーバーが必要です。**
- **基本的な SQL コマンドを知っている必要があります。**
- **MySQL 設定ファイルにアクセスできる必要があります。**

**手順**

1. **interactive_timeout を設定します。**

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

2. **wait_timeout を設定します。**

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

#### 1.2.4.1. オプションの説明

| permission/item | 説明 |
| --- | --- |
| **interactive_timeout** | サーバーが対話的な接続を閉じる前にアクティビティーの発生を待つ時間 (秒単位)。<br><br>**ヒント**<br><br>詳細は MySQL のドキュメント を参照してください。 |
| **wait_timeout** | サーバーが非対話的な接続を閉じる前にアクティビティーを待つ秒数。<br><br>**ヒント**<br><br>詳細は MySQL のドキュメント を参照してください。 |

### 1.2.5. Debezium のクエリーログイベントの有効化

**各 binlog イベントの元の SQL ステートメントを確認したい場合があります。MySQL 設定ファイルで binlog_rows_query_log_events オプションを有効にすると、これを行うことができます。**

**注記**

**このオプションは、MySQL 5.6 以降でのみ利用できます。**

**前提条件**

- **MySQL サーバーが必要です。**
- **基本的な SQL コマンドを知っている必要があります。**
- **MySQL 設定ファイルにアクセスできる必要があります。**

**手順**

1. **binlog_rows_query_log_events を有効にします。**

```
mysql> binlog_rows_query_log_events=ON
```

#### 1.2.5.1. オプションの説明

| permission/item | 説明 |
| --- | --- |
| **binlog_rows_query_log_events`** | binlog エントリーに元の **SQL** ステートメントを含めるサポートを有効または無効にするブール値。<br>• **ON** = 有効化<br>• **OFF** = 無効化 |

## 1.3. DEPLOYING THE MYSQL CONNECTOR

### 1.3.1. MySQL コネクターのインストール

**Debezium MySQL コネクターのインストールは、JAR をダウンロードして Kafka Connect 環境に抽出し、プラグインの親ディレクトリーが Kafka Connect 環境に指定されていることを確認する必要がある単純なプロセスです。**

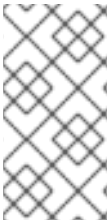
**前提条件**

- **Zookeeper、Kafka、および Kafka Connect がインストールされている。**
- **MySQL Server がインストールされ、設定されていること。**

**手順**

1. **Debezium MySQL コネクター をダウンロードします。**
2. **ファイルを Kafka Connect 環境に展開します。**
3. **プラグインの親ディレクトリーを Kafka Connect プラグインパスに追加します。**

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



注記

上記の例では、Debezium MySQL コネクターを /kafka/connect/Debezium-connector-mysql パスに展開したことを前提としています。

4. Kafka Connect プロセスを再起動します。これにより、新しい JAR が確実に選択されるようになります。

### 1.3.2. MySQL コネクターの設定

通常、コネクターに使用できる設定プロパティーを使用して、.yaml ファイルに Debezium MySQL コネクターを設定します。

**前提条件**

- コネクターの インストールプロセス を完了している必要があります。

**手順**

1. .yaml ファイルにコネクターの 名前 を設定します。
2. Debezium MySQL コネクターに必要な設定プロパティーを設定します。

**ヒント**

設定プロパティーの完全リストは、MySQL コネクター設定プロパティー を参照してください。

**MySQL コネクターの設定例**

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: inventory-connector 1
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
```

```
  tasksMax: 1 (2)
  config: (3)
    database.hostname: mysql (4)
    database.port: 3306
    database.user: debezium
    database.password: dbz
    database.server.id: 184054 (5)
    database.server.name: dbserver1 (6)
    database.whitelist: inventory (7)
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 (8)
    database.history.kafka.topic: schema-changes.inventory (9)
```

(1) (1)

コネクターの名前。

(2) (2)

1 度に 1 つのタスクのみが動作する必要があります。MySQL コネクターは MySQL サーバーの binlog を読み取るため、単一のコネクタータスクを使用することで、順序とイベントの処理が適切に行われるようになります。Kafka Connect サービスはコネクターを使用して作業を行う 1 つ以上のタスクを開始し、実行中のタスクを自動的に Kafka Connect サービスのクラスター全体に分散します。いずれかのサービスが停止またはクラッシュすると、これらのタスクは稼働中のサービスに再分散されます。

(3) (3)

コネクターの設定。

(4) (4)

データベースホスト。これは、MySQL サーバーを実行しているコンテナーの名前です (mysql)。

(5) (5)

一意なサーバー ID および名前。サーバー名は、MySQL サーバーまたはサーバーのクラスターの論理識別子です。

(6)

この名前は、すべての Kafka トピックの接頭辞として使用されます。

(7)

inventory データベースの変更のみが検出されます。

8

コネクターは、このブローカー (イベントの送信先となるブローカーと同じ) とトピック名を使用して、データベーススキーマの履歴を Kafka に保存します。

9

再起動時に、コネクターは、コネクターが読み取りを開始すべき時点で binlog に存在したデータベースのスキーマを復元します。

### 1.3.3. MySQL コネクター設定プロパティー

Debezium MySQL コネクターを実行するには、ここに記載の設定プロパティーが 必要 です。また、詳細の MySQL コネクタープロパティー もデフォルト値を変更する必要がなく、コネクター設定で指定する必要はありません。

ヒント

Debezium MySQL コネクターは、Kafka プロデューサーおよびコンシューマーの作成時に パススルー 設定をサポートします。パススルー プロパティーの詳細は、Kafka のドキュメント を 参照してください。

| プロパティー | デフォルト | 説明 |
|---|---|---|
| **name** | | コネクターの一意名。同じ名前で再登録を試みると失敗します。(このプロパティーはすべての Kafka Connect コネクターに必要です) |
| **connector.class** | | コネクターの Java クラスの名前。MySQL コネクターには、常に **io.debezium.connector.mysql.MySqlConnector** の値を使用します。 |
| **tasks.max** | **1** | このコネクターのために作成する必要のあるタスクの最大数。MySQL コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。 |
| **database.hostname** | | MySQL データベースサーバーの IP アドレスまたはホスト名。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **database.port** | **3306** | MySQL データベースサーバーのポート番号 (整数)。 |
| **database.user** | | MySQL データベースサーバーへの接続時に使用する MySQL データベースの名前。 |
| **database.password** | | MySQL データベースサーバーへの接続時に使用するパスワード。 |
| **database.server.name** | | 監視対象の特定の MySQL データベースサーバー/クラスターの namespace を識別および提供する論理名。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターから生成されるすべての Kafka トピック名の接頭辞として使用されるためです。英数字とアンダースコアのみを使用する必要があります。 |
| **database.server.id** | random | このデータベースクライアントの数値 ID。MySQL クラスターで現在稼働しているすべてのデータベースプロセスで一意である必要があります。このコネクターは、MySQL データベースクラスターを (この一意の ID を持つ) 別のサーバーとして結合するため、binlog を読み取ることができます。デフォルトでは、5400 から 6400 までの乱数が生成されますが、明示的な値を設定することを推奨します。 |
| **database.history.kafka.topic** | | コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。 |
| **database.history .kafka.bootstrap.servers** | | Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。これは、Kafka Connect プロセスによって使用される同じ Kafka クラスターを示す必要があります。 |
| **database.whitelist** | **空の文字列** | 監視するデータベース名と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないデータベース名は監視から除外されます。デフォルトでは、すべてのデータベースが監視されます。**database.blacklist** と併用できません。 |
| **database.blacklist** | **空の文字列** | 監視から除外されるデータベース名と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないデータベース名が監視されます。**database.whitelist** と併用できません。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **table.whitelist** | **空の文字列** | 監視するテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないテーブルはすべて監視から除外されます。各識別子の形式は databaseName.tableName です。デフォルトでは、コネクターは各監視対象データベースのシステム以外のテーブルをすべて監視します。**table.blacklist** と併用できません。 |
| **table.blacklist** | **空の文字列** | 監視から除外されるテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないテーブルはすべて監視されます。各識別子の形式は databaseName.tableName です。**table.whitelist** と併用できません。 |
| **column.blacklist** | **空の文字列** | 変更イベントメッセージの値から除外される必要がある列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **column.truncate.to.length.chars** | **該当なし** | フィールド値が指定された文字数より長い場合に、変更イベントメッセージ値で値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数である必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **column.mask.with.length.chars** | **該当なし** | 文字ベースの列の完全修飾名にマッチする正規表現のコンマ区切りリスト (オプション) で、変更イベントメッセージの値を、指定された数のアスタリスク (*) 文字で設定されるフィールド値に置き換える必要があります。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数またはゼロである必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **column.propagate.source.type** | **該当なし** | 出力された変更メッセージの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター( **__Debezium.source.column.type**、**__Debezium.source.column.length**、および **_Debezium.source.column.scale** )は、それぞれ元の型名と長さ（可変幅型）を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **time.precision.mode** | **adaptive_time_microseconds** | 時間、日付、およびタイムスタンプは、以下を含む異なる精度の種類で表すことができます。**adaptive_time_microseconds**（デフォルト）は、データベース列の型を基にして、ミリ秒、マイクロ秒、またはナノ秒の精度値のいずれかを使用して、データベースの値と全く同じように日付、日時、およびタイムスタンプ値をキャプチャーします。または connect は、常にマイクロ秒としてキャプチャーされる TIME タイプフィールドを除き、ミリ秒、マイクロ秒、または **connect** は、Kafka Connect の Time の組み込み表現を使用して時間とタイムスタンプ値を常に表現します。 date、および Timestamp。データベース列の精度に関わらず、ミリ秒の精度を使用します。 |
| **decimal.handling.mode** | **precise** | コネクターによる **DECIMAL** 列および **NUMERIC** 列の値の処理方法を指定します。**precise**（デフォルト）はバイナリー形式で変更イベントで表される **java.math.BigDecimal** 値を使用して正確に表します。または **double** は **double** 値を使用して表します。精度が失われる可能性はありますが、非常に簡単に使用できます。**string** オプションは値をフォーマットされた文字列としてエンコードします。これは簡単に使用できますが、実際の型に関するセマンティック情報は失われます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **bigint.unsigned.handling.mode** | **long** | BIGINT UNSIGNED 列を変更イベントで表す方法を指定します。**precise** は **java.math.BigDecimal** を使用して値を表します。値は、バイナリー表現と Kafka Connect の **org.apache.kafka.connect.data.Decimal** タイプを使用して変更イベントでエンコードされます。**long**（デフォルト）は Java の **long** を使用して値を表します。精度を提供しない場合もありますが、コンシューマーでの使用がはるかに簡単になります。通常は **long** が望ましい設定です。2^63 を超える値を使用する場合のみ、それらの値は **long** を使用して伝えことができないため、**正確な** 設定を使用する必要があります。 |
| **include.schema.changes** | **true** | コネクターがデータベーススキーマの変更を、データベースサーバー ID と同じ名前の Kafka トピックに公開するかどうかを指定するブール値。各スキーマの変更はデータベース名が含まれるキーを使用して記録され、その値には DDL ステートメントが含まれます。これは、コネクターがデータベース履歴を内部で記録する方法には依存しません。デフォルトは **true** です。 |
| **include.query** | **false** | 変更イベントを生成した元の SQL クエリーがコネクターに含まれる必要があるかどうかを指定するブール値。<br>注記：このオプションでは、MySQL を binlog_rows_query_log_events オプションを ON に設定する必要があります。スナップショットプロセスから生成されたイベントに対するクエリーは表示されません。<br>警告：このオプションを有効にすると、変更イベントに元の SQL ステートメントを含めることで、明示的にブラックリストに指定またはマスクされたテーブルまたはフィールドが公開される可能性があります。このため、このオプションはデフォルトで false に設定されます。 |
| **event.processing.failure.handling.mode** | **fail** | binlog イベントのデシリアライズ中にコネクターが例外に反応する方法を指定します。**fail** は例外（問題のあるイベントとその binlog オフセットを示す）を伝播し、コネクターを停止させます。<br>**warn** は問題のあるイベントをスキップし、問題のあるイベントとその binlog オフセットがログに記録されます。<br>**skip** は問題のあるイベントがスキップされます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **inconsistent.schema.handling.mode** | **fail** | 内部スキーマ表現に存在しないテーブルに関連する binlog イベント（つまり、内部表現がデータベースと一貫性がない）に関連する binlog イベントにコネクターがどのように反応するかを指定します。つまり、問題のあるイベントとその binlog オフセットを示す例外が発生します（問題のあるイベントとその binlog オフセットを示す）。<br>**warn** により、問題のあるイベントがスキップされ、問題のあるイベントとその binlog オフセットが記録されます。<br>**skip** は問題のあるイベントがスキップされます。 |
| **max.queue.size** | **8192** | データベースログから読み取られた変更イベントが Kafka に書き込まれる前に配置される、ブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに binlog リーダーにバックプレシャーを提供できます。キューに発生するイベントは、このコネクターによって定期的に記録されるオフセットには含まれません。デフォルトは 8192 で、常に **max.batch.size** プロパティーに指定された最大バッチサイズよりも大きくする必要があります。 |
| **max.batch.size** | **2048** | このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。デフォルトは 2048 です。 |
| **poll.interval.ms** | **1000** | 各反復処理の実行中に新しい変更イベントが表示されるまでコネクターが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。 |
| **connect.timeout.ms** | **30000** | コネクターが MySQL データベースサーバーへの接続を試行した後、タイムアウトするまでの最大の待機期間をミリ秒単位で指定する正の整数値。デフォルトは 30 秒です。 |
| **gtid.source.includes** | | MySQL サーバーで binlog の位置を見つけるために使用される GTID セットのソース UUID に一致する、正規表現のコンマ区切りリスト。これらの include パターンのいずれかに一致するソースを持つ GTID 範囲のみが使用されます。**gtid.source.excludes** と併用できません。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **gtid.source.excludes** | | MySQL サーバーで binlog の位置を見つけるために使用される GTID セットのソース UUID に一致する、正規表現のコンマ区切りリスト。これらの除外パターンのいずれにも一致するソースを持つ GTID 範囲のみが使用されます。**gtid.source.includes** と併用できません。 |
| **tombstones.on.delete** | **true** | 削除イベント後に廃棄 (tombstone) イベントを生成するかどうかを制御します。<br>**true** の場合、削除操作は 削除 イベントと後続の廃棄 (tombstone)イベントで表されます。**false** の場合、削除 イベントのみが送信されます。<br>廃棄 (tombstone) イベントを生成すると (デフォルトの動作)、Kafka はソースレコードが削除されると、指定のキーに関連するすべてのイベントを完全に削除できます。 |
| **message.key.columns** | **空の文字列** | プライマリーキーをマップする完全修飾テーブルおよび列と一致する正規表現のセミコロン区切りリスト。<br>各項目 (正規表現) は、カスタムキーを表す完全修飾 **<fully-qualified table>:<a comma-separated list of columns>** と一致する必要があります。<br>特定のコネクターに応じて、完全修飾テーブルは **DB_NAME.TABLE_NAME** または **SCHEMA_NAME.TABLE_NAME** として定義できます。 |

#### 1.3.3.1. 高度な MySQL コネクタープロパティー

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **connect.keep.alive** | **true** | MySQL サーバー/クラスターへの接続を確実に維持するために別のスレッドを使用するかどうかを指定するブール値。 |
| **table.ignore.builtin** | **true** | 組み込みシステムテーブルを無視するかどうかを指定するブール値。これは、テーブルのホワイトリストまたはブラックリストに関係なく適用されます。デフォルトでは、システムテーブルは監視から除外され、どのシステムテーブルにも変更が加えられてもイベントが生成されません。 |
| **database.history.kafka.recovery.poll.interval.ms** | **100** | 永続化されたデータのポーリングが行われている間にコネクターが起動/回復を待つ最大時間 (ミリ秒単位) を指定する整数値。デフォルトは 100 ミリ秒です。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **database.history.kafka.recovery.attempts** | **4** | エラーでコネクターのリカバリーが失敗する前に、コネクターが永続化された履歴データの読み取りを試行する最大回数。データが受信されなかった場合に最大待機する時間は、**recovery.attempts** x **recovery.poll.interval.ms** です。 |
| **database.history.skip.unparseable.ddl** | **false** | コネクターが不正なデータベースステートメントや不明なデータベースステートメントを無視するか、処理を停止し、オペレーターが問題を修正するかどうかを指定するブール値。安全なデフォルトは **false** です。スキップは、binlog が処理されたときにデータの損失や分割が発生する可能性があるため、注意して使用する必要があります。 |
| **database.history.store.only.monitored.tables.ddl** | **false** | コネクターがすべての DDL ステートメントを記録する必要があるかどうか、または( **true**の場合) Debezium によって監視されるテーブルに関連するもののみを記録するかどうかを指定するブール値（フィルター設定を使用）。安全なデフォルトは **false** です。この機能は、フィルターの変更時に不足しているデータが必要になる可能性があるため、注意して使用してください。 |
| **database.ssl.mode** | **disabled** | 暗号化された接続を使用するかどうかを指定します。デフォルトは が **無効** で、暗号化されていない接続の使用を指定します。<br><br>**推奨される** オプションは、サーバーがセキュアな接続をサポートしますが、暗号化されていない接続にフォールバックします。<br><br>**required** オプションは暗号化された接続を確立しますが、何らかの理由で作成できない場合は失敗します。<br><br>**verify_ca** オプションは **required** のように動作しますが、さらに、設定された認証局(CA)証明書に対してサーバー TLS 証明書を検証し、有効な CA 証明書と一致しない場合は失敗します。<br><br>**verify_identity** オプションは **verify_ca** のように動作しますが、さらにサーバー証明書がリモート接続のホストと一致することを確認します。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **binlog.buffer.size** | 0 | binlog リーダーによって使用される先読みバッファーのサイズ。<br>特定の条件下では、MySQL binlog に**ROLLBACK** ステートメントによって終了したコミットされていないデータが含まれる可能性があります。一般的な例としては、セーブポイントを使用したり、一時的なテーブルの変更と通常のテーブルの変更が 1 つのトランザクションに混在する場合などです。<br>トランザクションの開始が検出されると、Debezium は binlog の位置をロール転送して **COMMIT** または **ROLLBACK** を見つけ、トランザクションからの変更をストリーミングするかどうかを判断できるようにします。バッファーサイズは、トランザクション境界の検索中に Debezium がバッファーできるトランザクションの最大変更数を定義します。トランザクションのサイズがバッファーよりも大きい場合、Debezium はストリーミング中にバッファーに収まらないイベントを巻き戻し、再読み取りする必要があります。値 **0** はバッファーを無効にします。<br>デフォルトでは無効です。<br>**注記：** この機能は、実行中の機能として考慮する必要があります。お客様からのフィードバックが必要ですが、完全に有効ではないことが期待されています。 |
| **snapshot.mode** | **Initial** | コネクターの起動時にスナップショットを実行する基準を指定します。デフォルトは **initial** で、論理サーバー名に対してオフセットが記録されていない場合にのみコネクターがスナップショットを実行できます。**when_needed** オプションは、必要とみなされるたびにコネクターが起動時にスナップショットを実行するように指定します（オフセットが利用できない場合、以前に記録されたオフセットがサーバーで利用可能ではない binlog の場所または GTID を指定する場合）。**never** オプションは、接続でスナップショットを使用することはなく、論理サーバー名を使用して最初の起動時にコネクターが binlog の開始から読み取る必要があることを指定します。これは、binlog にデータベースの履歴全体が含まれることが保証されている場合にのみ有効であるため、注意して使用する必要があります。トピックにデータの整合性のあるスナップショットが含まれる必要はありませんが、コネクターの開始以降の変更のみを必要とする場合は、コネクターはスキーマ（データではなく）のみをスナップショットする **schema_only** オプションを使用できます。<br><br>**schema_only_recovery** は、既存のコネクターが破損または失われたデータベース履歴トピックをリカバリーしたり、予期せずに増加するデータベース履歴トピックを定期的にクリーンアップ（無限保持が必要）するためのリカバリーオプションです。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.locking.mode** | **最小** | スナップショットの実行中にコネクターがグローバル MySQL 読み取りロック（データベースへの更新を行なう）に保持するかどうか、およびその期間を制御します。許容値は、**extended**、 および **none** の 3 つです。<br><br>**minimal**: コネクターは、データベーススキーマと他のメタデータを読み取る間、スナップショットの最初の部分に対してのみグローバル読み取りロックを保持します。スナップショットの残りの作業では、各テーブルからすべての行を選択する必要があります。これは、グローバル読み取りロックが保持されなくなり、他の MySQL クライアントがデータベースを更新している場合でも、REEPEATABLE READ トランザクションを使用して一貫した方法で実行できます。<br><br>**拡張**: MySQL が REPEATABLE READ セマンティクスから除外する操作を送信するクライアントが、スナップショットの期間中すべての書き込みをブロックすることが望ましい場合があります。このような場合は、このオプションを使用します。<br><br>**none** Will は、スナップショットプロセス中にコネクターがテーブルロックを取得できないようにします。この値はすべてのスナップショットモードで使用できますが、スナップショットの作成中にスキーマの変更が行われていない場合に **のみ** 安全に使用できます。MyISAM エンジンで定義されたテーブルの場合、MyISAM によってテーブルロックが取得されるようにこのプロパティーが設定されていても、テーブルはロックされます。この動作は、行レベルのロックを取得する InnoDB エンジンとは異なります。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.select.statement.overrides** | | スナップショットに含まれるテーブルの行を制御します。<br>このプロパティーには、完全修飾テーブル (DB_NAME.TABLE_NAME) のコンマ区切りリストが含まれます。各テーブルの select ステートメントは、ID **snapshot.select.statement.overrides.[DB_NAME].[TABLE_NAME]** によって識別される、テーブルごとに 1 つずつ追加の設定プロパティーに指定されます。これらのプロパティーの値は、スナップショットの実行中に特定のテーブルからデータを取得するときに使用する SELECT ステートメントです。**大規模な追加専用テーブルで可能なユースケースとしては、前のスナップショットが中断された場合にスナップショットの開始 (再開) 点を設定することが挙げられます。**<br>**注記**: この設定はスナップショットにのみ影響します。binlog からキャプチャーされたイベントは全く影響を受けません。 |
| **min.row.count.to.stream.results** | **1000** | スナップショット操作中に、コネクターは含まれる各テーブルをクエリーし、そのテーブルのすべての行に対する読み取りイベントを生成します。このパラメーターは、MySQL 接続がテーブルのすべての結果をメモリーにプルする（高速ですが、大量のメモリーを必要とする）か、または結果が代わりにストリーミングされる（速度が遅い可能性がありますが、非常に大きなテーブルに対して機能する）かどうかを決定します。この値は、コネクターが結果をストリーミングする前にテーブルに含まれる必要がある行の最小数を指定します。デフォルトは 1,000 です。すべてのテーブルサイズチェックを省略し、スナップショットの実行中に常にすべての結果をストリーミングするには、このパラメーターを 0 に設定します。 |
| **heartbeat.interval.ms** | **0** | ハートビートメッセージが送信される頻度を制御します。<br>このプロパティーには、コネクターがハートビートメッセージをハートビートトピックに送信する頻度を定義するミリ秒単位の間隔が含まれます。このプロパティーを **0** に設定して、ハートビートメッセージが全く送信されないようにします。<br>デフォルトでは無効にされています。 |
| **heartbeat.topics.prefix** | **__debezium-heartbeat** | ハートビートメッセージが送信されるトピックの命名を制御します。<br>トピックは、**<heartbeat.topics.prefix>.<server.name>** パターンに従って名前が付けられます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **database.initial.statements** | | データベースへの JDBC 接続（トランザクションログの読み取り接続ではない）が確立されたときに実行される SQL ステートメントのセミコロン区切りリスト。セミコロン(';;')を使用してセミコロンを区切り文字としてではなく、文字として使用します。<br>**注記：コネクターは独自の判断で JDBC 接続を確立するため、通常これはセッションパラメーターの設定にのみ使用してくださいが、DML ステートメントの実行にはは使用しないでください。** |
| **snapshot.delay.ms** | | コネクターの起動後、スナップショットを取得するまで待機する間隔 (ミリ秒単位)。<br>クラスター内で複数のコネクターを開始する際に スナップショットが中断されないようにするために使用でき、コネクターのリバランスが実行される可能性があります。 |
| **snapshot.fetch.size** | | スナップショットの実行中に各テーブルから 1 度に読み取る必要がある行の最大数を指定します。コネクターは、このサイズの複数のバッチでテーブルの内容を読み取ります。 |
| **snapshot.lock.timeout.ms** | **10000** | スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する正の整数値。この時間間隔でテーブルロックを取得できない場合、スナップショットは失敗します。How the MySQL connector perform database snapshots を参照してください。 |
| **enable.time.adjuster** | | MySQL では、ユーザーは 2 桁の数字または 4 桁の年値を挿入できます。2 桁の数値の場合、値は自動的に 1970 - 2069 の範囲にマッピングされます。通常、これはデータベースで実行されます。<br>Debezium が変換を行う場合は **true** （デフォルト）に設定します。<br>変換が完全にデータベースに委譲されている場合は **false** に設定します。 |
| **sanitize.field.names** | コネクター設定が、Avro を使用するように **key.converter** または **value.converter** パラメーターを明示的に指定する場合は **true** です。それ以外の場合のデフォルトは **false** です。 | Avro の命名要件 に準拠するためにフィールド名がサニタイズされるかどうか。 |

### 1.3.4. MySQL コネクターの監視メトリクス

Debezium MySQL コネクターには、Zookeeper、Kafka、および Kafka Connect が持つ JMX メトリクスの組み込みサポートに加えて、3 つのメトリクスタイプがあります。

- スナップショットの実行時にコネクターを監視するための、スナップショットメトリクス;
- binlog メトリクス; CDC テーブルデータの読み取り時にコネクターを監視する
- コネクターのスキーマ履歴の状態を監視するための、スキーマ履歴メトリクス。

#### 1.3.4.1. スナップショットメトリクス

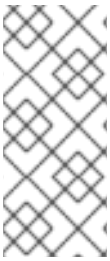
MBean は debezium.mysql:type=connector-metrics,context=snapshot,server=<database.server.name> です。

| 属性 | 型 | 説明 |
|---|---|---|
| TotalTableCount | int | スナップショットに含まれているテーブルの合計数。 |
| RemainingTableCount | int | スナップショットによってまだコピーされていないテーブルの数。 |
| HoldingGlobalLock | boolean | コネクターが現在グローバルまたはテーブルの書き込みロックを保持するかどうか。 |
| SnapshotRunning | boolean | スナップショットが起動されたかどうか。 |
| SnapshotAborted | boolean | スナップショットが中断されたかどうか。 |
| SnapshotCompleted | boolean | スナップショットが完了したかどうか。 |
| SnapshotDurationInSeconds | long | スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。 |
| RowsScanned | Map<String, Long> | スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。 |

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **LastEvent** | **string** | コネクターが読み取りした最後のスナップショットイベント。 |
| **MilliSecondsSinceLastEvent** | **long** | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| **TotalNumberOfEventsSeen** | **long** | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| **NumberOfEventsFiltered** | **long** | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |
| **MonitoredTables** | **string[]** | コネクターによって監視されるテーブルの一覧。 |
| **QueueTotalCapcity** | **int** | スナップショットリーダーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | スナップショットリーダーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |

#### 1.3.4.2. binlog メトリクス

**MBean は debezium.mysql:type=connector-metrics,context=binlog,server=<database.server.name> です。**



**注記**

**トランザクション関連の属性は、binlog イベントバッファーが有効になっている場合にのみ利用できます。詳細は、高度なコネクター設定プロパティーの binlog.buffer.size を参照してください。**

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **オンラインインストール** | **boolean** | コネクターが現在 MySQL サーバーに接続されているかどうかを示すフラグ。 |
| **BinlogFilename** | **string** | コネクターが最後に読み取られた binlog ファイル名の名前。 |

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **BinlogPosition** | **long** | コネクターによって読み取られた binlog 内の最新の位置 (バイト単位)。 |
| **IsGtidModeEnabled** | **boolean** | コネクターが現在 MySQL サーバーから GTID を追跡しているかどうかを示すフラグ。 |
| **GtidSet** | **string** | binlog の読み取り時にコネクターによって表示される最新の GTID セットの文字列表現。 |
| **LastEvent** | **string** | コネクターによって読み取られた最後の binlog イベント。 |
| **MilliSecondsBehindSource** | **long** | 最後のイベントの MySQL タイムスタンプとそれを処理するコネクターの間隔（ミリ秒単位）。値には、MySQL サーバーと MySQL コネクターが実行されているマシンのクロックの違いが組み込まれています。 |
| **TotalNumberOfEventsSeen** | **long** | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| **NumberOfSkippedEvents** | **long** | MySQL コネクターによってスキップされたイベントの数。通常、MySQL の binlog からの不正形式のイベントまたは解析不可能なイベントが原因で、イベントがスキップされます。 |
| **NumberOfEventsFiltered** | **long** | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |
| **NumberOfDisconnects** | **long** | MySQL コネクターによる切断の数。 |
| **SourceEventPosition** | **map<string, string>** | 最後に受信したイベントの位置。 |
| **LastTransactionId** | **string** | 最後に処理されたトランザクションのトランザクション識別子。 |
| **LastEvent** | **string** | コネクターによって読み取られた最後の binlog イベント。 |
| **MilliSecondsSinceLastEvent** | **long** | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| **MonitoredTables** | **string[]** | Debezium によって監視されるテーブルの一覧。 |

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **QueueTotalCapcity** | **int** | binlog リーダーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | binlog リーダーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |
| **NumberOfCommittedTransactions** | **long** | コミットされた処理済みトランザクションの数。 |
| **NumberOfRolledBackTransactions** | **long** | ロールバックされ、ストリーミングされなかった処理済みトランザクションの数。 |
| **NumberOfNotWellFormedTransactions** | **long** | 予想されるプロトコル **BEGIN** + **COMMIT**/**ROLLBACK** に準拠していないトランザクションの数。通常の条件下では **0** である必要があります。 |
| **NumberOfLargeTransactions** | **long** | 先読みバッファーに収まらないトランザクションの数。最適なパフォーマンスを得るには、**NumberOfCommittedTransactions** および **NumberOfRolledBackTransactions** よりも大幅に小さくする必要があります。 |

#### 1.3.4.3. スキーマ履歴メトリクス

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

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **Status** | **string** | データベース履歴の状態を記述する **STOPPED**、**RECOVERING**（ストレージから履歴を復元）、**RUNNING** のいずれか。 |
| **RecoveryStartTime** | **long** | リカバリーが開始された時点のエポック秒の時間。 |
| **ChangesRecovered** | **long** | リカバリーフェーズ中に読み取られた変更の数。 |
| **ChangesApplied** | **long** | リカバリーおよびランタイム中に適用されるスキーマ変更の合計数。 |
| **MilliSecondsSinceLastRecoveredChange** | **long** | 最後の変更が履歴ストアから復元された時点からの経過時間 (ミリ秒単位)。 |

| 属性 | 型 | 説明 |
| --- | --- | --- |
| **MilliSecondsSinceLastAppliedChange** | **long** | 最後の変更が適用された時点からの経過時間 (ミリ秒単位)。 |
| **LastRecoveredChange** | **string** | 履歴ストアから復元された最後の変更の文字列表現。 |
| **LastAppliedChange** | **string** | 最後に適用された変更の文字列表現。 |

### 1.4. MYSQL コネクターの一般的な問題

#### 1.4.1. 設定および起動エラー

**Debezium MySQL コネクターが失敗し、エラーを報告し、以下の起動エラーが発生した場合に実行を停止します。**

- **コネクターの設定が無効である。**
- **指定の接続パラメーターを使用してコネクターを MySQL サーバーに接続できません。**
- **MySQL に履歴が含まれなくなった binlog の位置でコネクターが再起動を試みます。**

**これらのエラーのいずれかを受け取ると、エラーメッセージの詳細が表示されます。エラーメッセージには、可能な場合は回避策も含まれます。**

#### 1.4.2. MySQL が利用できない

**MySQL サーバーが利用できなくなると、Debezium MySQL コネクターはエラーで失敗し、コネクターが停止します。サーバーが利用できる場合は、コネクターを再起動する必要があります。**

##### 1.4.2.1. GTID の使用

**GTID が有効で、可用性の高い MySQL クラスターがある場合は、コネクターが単にクラスター内の別の MySQL サーバーに接続し、最後のトランザクションを表すサーバーの binlog の場所を見つけ、その特定の場所から新しいサーバーの binlog の読み取りを開始するため、コネクターをすぐに再起動します。**

#### 1.4.2.2. GTID を使用しない

GTID が有効でない場合、コネクターは接続した MySQL サーバーの binlog の位置のみを記録します。正しい binlog の位置から再起動するには、その特定のサーバーに再接続する必要があります。

### 1.4.3. Kafka Connect が停止しました。

Kafka Connect が停止すると、いくつかの問題が発生するシナリオが 3 つあります。

- 「Kafka Connect が正常に停止する」
- 「Kafka Connect プロセスのクラッシュ」
- 「Kafka が使用不可能になる」

#### 1.4.3.1. Kafka Connect が正常に停止する

Kafka Connect が正常に停止すると、Debezium MySQL コネクタータスクが停止され、新しい Kafka Connect プロセスで再起動される間は短い遅延のみが発生します。

#### 1.4.3.2. Kafka Connect プロセスのクラッシュ

Kafka Connect がクラッシュすると、プロセスが停止し、最後に処理されたオフセットが記録されずに Debezium MySQL コネクタータスクが終了します。分散モードでは、Kafka Connect は他のプロセスでコネクタータスクを再起動します。ただし、MySQL コネクターは以前のプロセスで記録された最後のオフセットから再開します。つまり、代替タスクはクラッシュ前に処理された同じイベントの一部を生成し、重複したイベントを作成することを意味します。

ヒント

各変更イベントメッセージには、以下に関するソース固有の情報が含まれます。

- イベント元

- MySQL サーバーのイベント時間
- binlog のファイル名および位置
- GTID (使用されている場合)

#### 1.4.3.3. Kafka が使用不可能になる

Kafka Connect フレームワークは、Kafka プロデューサー API を使用して Debezium 変更イベントを記録します。Kafka ブローカーが利用できなくなると、Debezium MySQL コネクターは接続が再確立されるまで一時停止し、コネクターは最後に停止した場所を再開します。

### 1.4.4. MySQL が binlog ファイルをパージする

Debezium MySQL コネクターが長時間停止すると、MySQL サーバーは古い binlog ファイルをパージするため、コネクターの最後の位置が失われる可能性があります。コネクターが再起動すると、MySQL サーバーに開始点がなくなり、コネクターは別の最初のスナップショットを実行します。スナップショットが無効の場合、コネクターはエラーによって失敗します。

ヒント

初期スナップショットの詳細は、MySQL コネクターによるデータベーススナップショットの実行方法 を 参照してください。

# 第2章 POSTGRESQL の DEBEZIUM コネクター

Debezium の PostgreSQL コネクターは、PostgreSQL データベースのスキーマで行レベルの変更を監視および記録できます。

PostgreSQL サーバー/クラスターに初めて接続すると、すべてのスキーマの整合性スナップショットが読み込まれます。スナップショットが完了すると、コネクターは PostgreSQL 9.6 以降にコミットされた変更を継続的にストリーミングし、対応する insert、update、および delete イベントを生成します。各テーブルのすべてのイベントは、アプリケーションやサービスで簡単に使用できる個別の Kafka トピックに記録されます。

## 2.1. 概要

PostgreSQL の 論理デコード 機能はバージョン 9.4 で最初に導入されました。は、トランザクションログにコミットされた変更の抽出と、出力プラグイン を使用してユーザーフレンドリーな方法でこれらの変更の処理を可能にするメカニズムです。クライアントが変更を使用できるようにするには、PostgreSQL サーバーを実行する前にこの出力プラグインをインストールし、レプリケーションスロットと共に有効にする必要があります。

PostgreSQL コネクターには、サーバーの変更の読み取りおよび処理を可能にするために連携する 2 つの異なる部分が含まれています。

- PostgreSQL サーバーにインストールおよび設定する必要がある論理デコード出力プラグイン。
- PostgreSQL JDBC ドライバーを介して PostgreSQL の ストリーミングレプリケーションプロトコル を使用して、プラグインによって生成された変更を読み取る Java コード（実際の Kafka Connect コネクター）

その後、コネクターは受信したすべての行レベルの insert、update、および delete 操作の 変更イベント を生成し、個別の Kafka トピックの各テーブルの変更イベントをすべて記録します。クライアントアプリケーションは、対象のデータベーステーブルに対応する Kafka トピックを読み取り、これらのトピックに表示されるすべての行レベルのイベントに対応します。

通常、PostgreSQL は一定期間後に WAL セグメントをパージします。つまり、コネクターにはデータベースに加えられたすべての変更の完全な履歴はありません。そのため、PostgreSQL コネクターが最初に特定の PostgreSQL データベースに接続すると、データベーススキーマごとに 整合性スナップショット を実行して起動します。コネクターは、スナップショットの完成後に、スナップショットが作成された正確な時点から変更のストリーミングを続行します。これにより、すべてのデータの一貫した

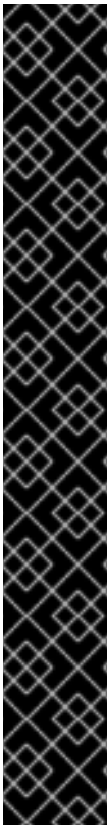
ビューから開始しますが、スナップショットの実行中に加えられた変更を失うことなく読み取りを続行します。

コネクターはフォールトトラレントでもあります。コネクターは変更を読み取り、イベントを生成すると、各イベントで write-ahead ログの位置を記録します。コネクターが何らかの理由で停止した場合（通信障害、ネットワークの問題、クラッシュなど）、再起動時に最後に停止した場所で WAL の読み取りを続行します。これにはスナップショットが含まれます。コネクターの停止時にスナップショットが完了しなかった場合、再起動時に新しいスナップショットが開始されます。

### 2.1.1. 論理デコード出力プラグイン

pgoutput 論理デコーダーは、Debezium の Tecnhology Preview リリースで唯一対応している論理デコーダーです。

PostgreSQL 10+ の標準的な論理デコードプラグインである pgoutput は Postgres コミュニティーにより維持され、Postgres によって 論理レプリケーション にも使用されます。pgoutput プラグインは常に存在します。つまり、追加のライブラリーがインストールされず、コネクターは raw レプリケーションイベントストリームを変更イベントに直接解釈します。



重要

コネクターの機能は、PostgreSQL の論理デコード機能に依存します。コネクターによっても反映される以下の制限事項に注意してください。

1. 論理デコードは DDL の変更をサポートしません。これは、コネクターが DDL の変更イベントをコンシューマーに報告できないことを意味します。

2. 論理デコードレプリケーションスロットは プライマリー サーバーでのみサポートされます。つまり、PostgreSQL サーバーのクラスターがある場合、コネクターはアクティブな プライマリー サーバー上でのみ実行できます。hot または warm スタンバイのレプリカでは実行できません。プライマリー サーバーが失敗するか降格されると、コネクターは停止します。プライマリー が回復した後、コネクターを再起動することができます。別の PostgreSQL サーバーが プライマリー に昇格された場合は、コネクターを再起動する前にコネクター設定を調整する必要があります。問題が 発生したときのコネクターの動作について、詳しく確認し てください。

重要

Debezium は現在、UTF-8 文字エンコーディングのデータベースのみをサポートします。1 バイト文字エンコーディングでは、拡張 ASCII コード文字を含む文字列を正しく処理できません。

## 2.2. POSTGRESQL の設定

本リリースの Debezium は、ネイティブの pgoutput 論理レプリケーションストリームのみをサポートします。pgoutput を使用して PostgreSQL を設定するには、レプリケーションスロットを有効にし、レプリケーションを実行するのに十分な権限を持つユーザーを設定する必要があります。

### 2.2.1. レプリケーションスロットの設定

PostgreSQL の論理デコード機能はレプリケーションスロットを使用します。

最初に、レプリケーションスロットを設定します。

**postgresql.conf**

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

- wal_level は、先行書き込みログで論理デコードを使用するようにサーバーに指示します。
- max_wal_senders は、WAL 変更を処理するために最大 1 つの別個のプロセスを使用するようにサーバーに指示します
- max_replication_slots は、WAL 変更をストリーミングするために最大 1 つのレプリケーションスロットを作成できるようにサーバーに指示します。

レプリケーションスロットは、Debezium の停止中でも Debezium に必要なすべての WAL を保持

することが保証されます。この理由により、レプリケーションスロットを密接に監視して、ディスク消費量が多すぎることや、レプリケーションスロットが過剰に使用されなくなった場合にカタログブロットなどの他の状態が発生する可能性を避けることが重要です。詳細は、Postgres のドキュメント を参照し てください。

注記

PostgreSQL 先行書き込みログのメカニズムおよび 設定に関する WAL 設定ドキュメント を読み、理解することが推奨されます。

### 2.2.2. パーミッションの設定

次に、レプリケーションを実行できるデータベースユーザーを設定します。

レプリケーションは、適切なパーミッションを持つデータベースユーザーがのみ実行でき、設定された数のホストに対してのみ実行できます。

ユーザーレプリケーションの権限を付与するには、少なくとも REPLICATION および LOGIN パーミッションを持つ PostgreSQL ロールを定義します。以下に例を示します。

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

注記

スーパーユーザーはデフォルトで上記の両方のロールを持ちます。

最後に、PostgreSQL サーバーを設定して、サーバーマシンと PostgreSQL コネクターが実行されているホスト間でレプリケーションが行われるようにします。

**pg_hba.conf**

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

1

ローカルで < youruser> のレプリケーション を許可するようにサーバーに指示します（つまりサーバーマシン上）。

2

IPV4を使用してレプリケーションの変更を受け取るように、localhost の <youruser > を許可するようサーバーに指示します。

3

IPV6を使用してレプリケーションの変更を受け取るように、localhost の <youruser > を許可するようサーバーに指示します。

注記

ネットワークマスクの詳細は、PostgreSQL のドキュメント を 参照してください。

### 2.2.3. WAL ディスク領域の使用

場合によっては、WAL ファイルによって使用される PostgreSQL ディスク領域が急増したり、通常の比例を上回ったりする可能性があります。この状況を説明する可能性のある理由は 3 つあります。

- Debezium は、処理されたイベントの LSN をデータベースに定期的に確認します。これは、pg_replication_slots スロットテーブルの confirmed_flush_lsn として表示されます。データベースはディスク領域を回収し、WAL サイズは同じテーブルの restart_lsn から計算できます。そのため、verify ed_flush_lsn が定期的に増加し、restart_lsn ラグが発生する場合、データベースは領域を解放する必要があります。通常、ディスク領域はバッチブロックで回収されるため、これは予想される動作であり、ユーザー側でのアクションは必要ありません。

- 監視対象のデータベースには多くの更新がありますが、監視されるテーブルやスキーマに関連するマイナス量のみがあります。この状況は、heartbeat.interval.ms 設定オプションを使用して定期的なハートビートイベントを有効にすることで簡単に解決できます。

- PostgreSQL インスタンスには、複数のデータベースが含まれており、そのうちの 1 つがトラフィックが多いデータベースです。Debezium は、トラフィックが少ない別のデータベースを監視します。レプリケーションスロットがデータベースごとに機能し、Debezium が呼び出しされないため、Debezium は LSN を確認できません。WAL はすべてのデータベースで共

有されているため、Debezium によって監視されるデータベースによってイベントが出力されるまで増大する傾向があります。

3 番目の原因を解決するには、以下を行う必要があります。

- heartbeat.interval.ms 設定オプションを使用した定期的なハートビートレコードの生成の有効化
- Debezium によって追跡されるデータベースから変更イベントを定期的に出力

その後、別のプロセスがテーブルを定期的に更新します（新しいイベントを挿入したり、同じ行をすべて更新したりします）。次に PostgreSQL は Debezium を呼び出して、最新の LSN を確認し、データベースが WAL 領域を回収できるようにします。

ヒント

Postgres を使用した AWS RDS のユーザーの場合は、AWS RDS が頻繁にユーザーに表示されないため、3 つ目の原因と同様の状況がアイドル状態の環境で発生する可能性があります。再びイベントを定期的に出力すると、問題が解決されます。

### 2.2.4. PostgreSQL コネクターの仕組み

#### 2.2.4.1. スナップショット

ほとんどの PostgreSQL サーバーは WAL セグメントにデータベースの完全な履歴を保持しないよう設定されているため、PostgreSQL コネクターは WAL を読み取るだけでデータベースの履歴全体を確認できません。そのため、デフォルトではコネクターは初回起動時にデータベースの最初の 整合性スナップショット を実行します。各スナップショットは以下の手順で設定されます（組み込みスナップショットモードを使用する場合、カスタム スナップショットモードはこれを上書きする可能性があります）。

1. SERIALIZABLE、READ ONLY、DEFERRABLE 分離レベルでトランザクションを開始し、 このトランザクション内の後続のすべての読み取りがデータの単一バージョンに対して実行されるようにします。他のクライアントによる後続の INSERT、UPDATE、DELETE 操作によるデータへの変更は、このトランザクションでは認識されません。
2. スナップショットの実行中に、監視されるテーブルごとに ACCESS SHARE MODE ロックを取得し、テーブルの構造が変更されないようにします。これらのロックは、操作中にテーブルの INSERTS、UPDATES、DELETES が実行されないようにしないことに注意してくださ

い。この手順は、エクスポートしたスナップショットモードを使用してロックのないスナップショット を許可する場合、省略 されます。

3. サーバーのトランザクションログの現在の位置を読み取ります。

4. すべてのデータベーステーブルとスキーマをスキャンし、各行の READ イベントを生成し、そのイベントを適切なテーブル固有の Kafka トピックに書き込みます。

5. トランザクションをコミットします。

6. コネクターオフセットにスナップショットの正常な完了を記録します。

コネクターに障害が発生した場合、コネクターのリバランスが発生した場合、または 1 の開始後、ステップ 6 の完了前に停止した場合、コネクターは再起動後に新しいスナップショットを開始します。コネクターが最初のスナップショットを完了すると、PostgreSQL コネクターはステップ 3 の実行時に読み取られた位置からのストリーミングを続行し、更新を見逃さないようにします。何らかの理由でコネクターが再び停止した場合、再起動時に、最後に停止した場所から変更のストリーミングを続行します。

2 つ目のスナップショットモードでは、コネクターは 常 にスナップショットを実行できます。この動作は、起動時に 常 にスナップショットを実行するようコネクターに指示します。スナップショットの完了後に、上記の手順 3 からの変更のストリーミングを続行します。このモードは、WAL セグメントが削除され、使用できなくなったことが確認された場合や、新しいプライマリーがプロモートされた後にクラスターに障害が発生した場合に、コネクターが新しいプライマリーがプロモートされた後に行われた可能性のある変更に見逃さないようにする場合に使用できます。

3 つ目のスナップショットモードは、スナップショットを実行し ないようにコネクター に指示します。この方法で新しいコネクターを設定すると、が以前の保存済みオフセットから変更のストリーミングを続行するか、PostgreSQL の論理レプリケーションスロットがサーバー上で最初に作成された時点から開始します。このモードは、対象のすべてのデータがまだ WAL に反映されている場合にのみ便利です。

4 番目のスナップショットモードは、初期のみ で、データベーススナップショットを実行し、その他の変更をストリーミングする前に停止します。コネクターが起動していても、停止前にスナップショットを完了しなかった場合、コネクターはスナップショットプロセスを再起動し、スナップショットが完了すると停止します。

エクスポートさ れた 5 番目のスナップショットモードは、レプリケーションスロットが作成された時点に基づいてデータベーススナップショットを実行します。このモードは、ロックのない方法でス

ナップショットを実行するのに最適です。

#### 2.2.4.2. 変更のストリーミング

通常、PostgreSQL コネクターは、接続されている PostgreSQL サーバーから変更をストリーミングするのに多くの時間を費やします。このメカニズムは、クライアントが特定の位置（ログ シーケンス番号 または短い LSN とも呼ばれる）でサーバーのトランザクションログにコミットされたときにサーバーから変更を受け取る PostgreSQL のレプリケーションプロトコル に依存します。

サーバーがトランザクションをコミットするたびに、別のサーバープロセスが 論理デコードプラグイン からコールバック関数を呼び出します。この関数はトランザクションからの変更を処理し、特定の形式(Debezium プラグインの場合は Protobuf または JSON)に変換して、クライアントが使用できる出力ストリームに書き込みます。

PostgreSQL コネクターは PostgreSQL クライアントとして機能し、これらの変更を受信すると、イベントを Debezium の 作成、更新、またはイベントの LSN の位置が含まれるイベントに変換します。*PostgreSQL コネクターはこれらの変更イベントを Kafka Connect フレームワーク（同じプロセスで実行されている）に転送するため、適切な Kafka トピックに同じ順序で非同期に書き込みます。Kafka Connect は、Debezium が各イベントに含まれるソース固有の位置情報に オフセット を使用し、Kafka Connect は別の Kafka トピックに最新のオフセットを定期的に記録します。*

*Kafka Connect が正常にシャットダウンすると、コネクターを停止し、すべてのイベントを Kafka にフラッシュし、各コネクターから受け取った最後のオフセットを記録します。再起動時に、Kafka Connect は各コネクターの最後に記録されたオフセットを読み取り、その時点からコネクターを起動します。PostgreSQL コネクターは、各変更イベントに記録された LSN をオフセットとして使用するため、コネクターを再起動すると PostgreSQL サーバーはその位置の直後に開始するイベントを送信します。*

*注記*

*PostgreSQL コネクターは、論理デコーダープラグインによって送信されるイベントの一部としてスキーマ情報を取得します。唯一の例外は、プライマリーキーを設定する列に関する情報です。この情報は JDBC メタデータ（サイドチャネル）から取得されるためです。テーブルのプライマリーキー定義が変更されると(PK 列の追加、削除、または名前変更による)、JDBC からのプライマリーキー情報が論理デコードイベントの変更データと同期されず、一貫性のないキー構造でメッセージが少なって作成されます。これが発生すると、コネクターの再起動とメッセージの再処理により問題が修正されます。問題を完全に回避するには、以下の操作シーケンスを使用して Debezium のプライマリーキー構造への更新を同期することが推奨されます。*

- *データベースまたはアプリケーションを読み取り専用モードにする*
- *Debezium が残りのイベントをすべて処理させる*
- *Debezium の停止*
- *プライマリーキー定義の更新*
- *データベースまたはアプリケーションを読み取り/書き込み状態にし、Debezium を再び開始します。*

#### *2.2.4.3. PostgreSQL 10+ Logical Decoding Support (pgoutput)*

*PostgreSQL 10+ の時点で、*pgoutput *と呼ばれる新しい論理レプリケーションストリームモードが導入されました。この論理レプリケーションストリームモードは PostgreSQL によってネイティブにサポートされるため、このコネクターは追加のプラグインをインストールしなくてもレプリケーションストリームを使用できます。これは、プラグインのインストールがサポートされていない、または許可されない環境で特に便利です。*

*詳細は、PostgreSQL の設定 を 参照してください。*

#### *2.2.4.4. トピック名*

*PostgreSQL コネクターは、単一テーブルのすべての挿入、更新、および削除操作をのイベントを単一の Kafka トピックに書き込みます。デフォルトでは、Kafka トピック名は* serverName.schemaName *です。*serverName *は database.server.name 設定プロパティーで指定し*

*たコネクターの論理名で、schemaName は操作が発生したデータベーススキーマの名前、tableName は操作が発生したデータベーステーブルの名前です。*

*たとえば、postgres データベースを備えた PostgreSQL インストールと、製品、products_on_hand、customers、および orders の 4 つのテーブルが含まれる インベントリースキーマについて考えてみましょう。コネクターが監視するこのデータベースの論理サーバー名 fulfillment が指定されている場合、コネクターは以下の 4 つの Kafka トピックでイベントを生成します。*

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

*一方、テーブルが特定のスキーマの一部ではなく、デフォルトの パブリック PostgreSQL スキーマで作成された場合、Kafka トピックの名前は以下のようになります。*

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

#### *2.2.4.5. メタ情報*

*PostgreSQL コネクターによって生成された各 レコード には、データベースイベント の他に、サーバーでイベントが発生した場所、ソースパーティションの名前、イベントを配置する Kafka トピックおよびパーティションの名前に関する一部のメタ情報があります。*

-

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

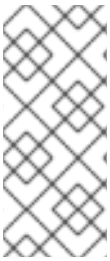
PostgreSQL コネクターは 1 つの Kafka Connect パーティション のみを使用し、生成されたイベントを 1 つの Kafka パーティションに配置します。そのため、sourcePartition の名前は常に database.server.name 設定プロパティーの名前にデフォルト設定されますが、kafkaPartition の値は null で、コネクターは特定の Kafka パーティションを使用しないことを意味します。

メッセージの sourceOffset 部分には、イベントが発生したサーバーの場所に関する情報が含まれます。

- LSN は トランザクションログの PostgreSQL ログシーケンス番号 または オフセット を表します
- txId はイベントの原因となったサーバートランザクションの識別子を表します
- ts_ms は、トランザクションがコミットされたサーバー時間として Unix Epoch からのマイクロ秒の数を表します。

#### 2.2.4.6. イベント

PostgreSQL コネクターによって生成されたすべてのデータ変更イベントにはキーと値がありますが、キーと値の構造は変更イベントの発生元となるテーブルによって異なります( Topic namesを参照)。



注記

Kafka 0.10 以降、Kafka はオプションでメッセージキーで記録でき、メッセージが作成（プロデューサーによって記録）されたタイムスタンプ、または Kafka によってログに書き込まれたタイムスタンプを値として記録できます。

*警告*

*PostgreSQL コネクターは、すべての Kafka Connect スキーマ名が 有効な Avro スキーマ名 になるようにします。つまり、論理サーバー名はラテン文字またはアンダースコア（例 : [a-z,A-Z,_]）で開始し、スキーマおよびテーブル名の残りの文字（例 : [a-z,A-Z,0-9,_]）で始まり、ラテン文字、数字、またはアンダースコア（例 : [a-z,A-Z,0-9,_]）で始まる必要があります。そうでない場合は、すべての無効な文字が自動的にアンダースコア文字に置き換えられます。*

*これにより、論理サーバー名、スキーマ名、およびテーブル名に他の文字が含まれ、テーブルのフルネームを区別する唯一の文字が無効になり、アンダースコアに置き換えられたため、予期せぬ競合が発生する可能性があります。*

*Debezium および Kafka Connect はイベント メッセージの継続的なストリームに基づいて設計されており、これらのイベント の構造は時間の経過とともに変更される可能性があります。これは、コンシューマーが処理するのが困難な場合があるため、Kafka Connect を容易にすることで、各イベントを自己完結させることができます。すべてのメッセージキーと値には、スキーマ と ペイロード の 2 つの部分で設定されます。スキーマはペイロードの構造を記述しますが、ペイロードには実際のデータが含まれます。*

#### *2.2.4.6.1. 変更イベントのキー*

*指定のテーブルでは、変更イベントのキーの構造には、イベントの作成時にテーブルのプライマリーキー（または REPLICA IDENTITY が FULL または USING INDEX に設定された一意のキー制約）の各列のフィールドが含まれます。*

*public データベーススキーマに定義されている customers テーブルについて考えてみましょう。*

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

*database.server.name 設定プロパティーに PostgreSQL_server の値がある場合、この定義がある限り customers テーブルの変更イベントはすべて同じキー構造を特長とし、JSON では以下のようになります。*

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

キーの スキーマ 部分には、キーの部分の内容を記述する Kafka Connect スキーマが含まれます。この場合、ペイロード 値はオプションではなく、PostgreSQL_server.public.customers.Key という名前のスキーマによって定義された構造であり、タイプ int32 の id という名前の必須フィールドが 1 つあります。キーの payload フィールドの値を確認すると、値が 1 つの id フィールドを持つ構造 (JSON では単なるオブジェクト)であることがわかります。

したがって、この鍵は、id プライマリーキー列の値が 1 である public.customers テーブルの行( PostgreSQL_serverという名前のコネクターからの出力)を説明するものとして解釈されます。

注記

column.blacklist 設定プロパティーを使用するとイベント値から列を削除できますが、プライマリーキーまたは一意キーのすべての列は常にイベントのキーに含まれます。

警告

テーブルにプライマリーキーまたは一意キーがない場合は、変更イベントのキーは null になります。これは、プライマリーキー制約または一意キー制約のないテーブルの行は一意に識別できないために理にかなっています。

#### 2.2.4.6.2. 変更イベントの値

変更イベントメッセージの値は、少し複雑です。message キーと同様に、schema セクションと payload セクションがあります。PostgreSQL コネクターによって生成されたすべての変更イベント値の payload セクションには、以下のフィールドを含む エンベロープ 構造があります。

- op は、操作のタイプを記述する文字列値が含まれる必須フィールドです。PostgreSQL コネクターの値は、c （作成または挿入）、u （更新）、d （削除）、および r （読み取り、スナップショットの場合）です。

- before は任意のフィールドであり、存在する場合はイベント発生 前 の行の状態が含まれます。この構造は、PostgreSQL_server.public.customers.Value Kafka Connect スキーマによって記述され、PostgreSQL_server コネクターは public.customers テーブルのすべての行に使用します。

警告

このフィールドが利用可能かどうかは、各テーブルの REPLICA IDENTITY 設定によって異なります。

- after はオプションのフィールドで、存在する場合はイベント発生 後 の行の状態が含まれます。構造は、以前 で使用されるのと同じ PostgreSQL_server.public.customers.Value Kafka Connect スキーマによって記述され ます。

- source は、イベントのソースメタデータを記述する構造が含まれる必須のフィールドです。PostgreSQL の場合は、Debezium バージョン、コネクター名、影響を受けるデータベースの名前、スキーマ、テーブルの名前、イベントが継続中のスナップショットの一部であるかどうか、レコードの メタ情報 セクションからの同じフィールドが含まれます。

- ts_ms は任意です。存在する場合は、コネクターがイベントを処理した時間(Kafka Connect タスクを実行している JVM のシステムクロックを使用)が含まれます。

当然ながら、イベントメッセージの値のschema 部分には、このエンベロープ構造と、その中のネストされたフィールドを記述するスキーマが含まれます。

#### 2.2.4.6.3. レプリカ ID

REPLICA IDENTITY は UPDATE および DELETE イベントの場合に 論理デコード に使用できる情報量を決定する PostgreSQL 固有のテーブルレベルの設定です。具体的には、上記のイベントのいずれかが発生するたびに、関係するテーブル列の以前の値に関する利用可能な情報（ある場合）を制御します。

REPLICA IDENTITY には 4 つの可能性があります。

- DEFAULT - UPDATE および DELETE イベントには、テーブルのプライマリーキー列の以前の値のみが含まれます。UPDATE の場合、値が変更されたプライマリー列のみが存在します。
- NOTHING: UPDATE および DELETE イベントには、テーブル列の以前の値に関する情報は含まれません。
- FULL: UPDATE および DELETE イベントには、すべてのテーブルの列の以前の値が含まれます。
- INDEX インデックス名: UPDATE および DELETE イベントには、index name という名前のインデックス定義に含まれる列の以前の値が含まれます。UPDATE の場合は、値が変更されたインデックス化された列のみが存在します。

#### 2.2.4.6.4. イベントの作成

customers テーブルの create イベント値を見てみましょう。

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

```
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "PostgreSQL_server.inventory.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "PostgreSQL_server.inventory.customers.Value",
        "field": "after"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
```

```
            "field": "connector"
        },
        {
            "type": "string",
            "optional": false,
            "field": "name"
        },
        {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
        },
        {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
        },
        {
            "type": "string",
            "optional": false,
            "field": "db"
        },
        {
            "type": "string",
            "optional": false,
            "field": "schema"
        },
        {
            "type": "string",
            "optional": false,
            "field": "table"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "txId"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "lsn"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "xmin"
        }
    ],
    "optional": false,
    "name": "io.debezium.connector.postgresql.Source",
    "field": "source"
},
{
    "type": "string",
    "optional": false,
```

```
            "field": "op"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
          }
        ],
        "optional": false,
        "name": "PostgreSQL_server.inventory.customers.Envelope"
      },
      "payload": {
        "before": null,
        "after": {
          "id": 1,
          "first_name": "Anne",
          "last_name": "Kretchmar",
          "email": "annek@noanswer.org"
        },
        "source": {
          "version": "1.0.3.Final",
          "connector": "postgresql",
          "name": "PostgreSQL_server",
          "ts_ms": 1559033904863,
          "snapshot": true,
          "db": "postgres",
          "schema": "public",
          "table": "customers",
          "txId": 555,
          "lsn": 24023128,
          "xmin": null
        },
        "op": "c",
        "ts_ms": 1559033904863
      }
    }
```

*このイベントの値 の スキーマ 部分を確認すると、エンベロープ の スキーマ、ソース 構造のスキーマ(PostgreSQL コネクターに固有ですべてのイベントで再利用)、before および after フィールドのテーブル固有のスキーマを確認できます。*

*注記*

*before および after フィールドのスキーマ名は* logicalName.schemaName *の形式であるため、.Value は他のすべてのテーブルのスキーマから完全に独立しています。*

*つまり、Avro コンバーター を使用する場合、各 論理ソース の各テーブル の Avro スキーマには独自の進化と履歴があります。*

このイベントの値 の ペイロード 部分を確認すると、イベント内の情報、行が作成されたことを説明するものが表示されます( op=c以降)、after フィールドの値には新しい挿入された行の ID、first_name、last_name、および email 列の値が含まれます。

注記

イベントの JSON 表現はそれが記述する行よりもはるかに大きいように見えることがあります。JSON 表現にはメッセージのスキーマ 部分と ペイロード 部分を含める必要があるため、これは True です。

Avro コンバーター を使用して、Kafka トピックに書き込まれた実際のメッセージのサイズを大幅に減らすこともできます。

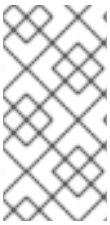
#### 2.2.4.6.5. 更新イベント

このテーブルの 更新 変更イベントの値は、実際にはまったく同じ スキーマ を持ち、そのペイロードは同じように設定されますが、異なる値を保持します。以下に例を示します。

```
{
   "schema": { ... },
   "payload": {
      "before": {
         "id": 1
      },
      "after": {
         "id": 1,
         "first_name": "Anne Marie",
         "last_name": "Kretchmar",
         "email": "annek@noanswer.org"
      },
      "source": {
         "version": "1.0.3.Final",
         "connector": "postgresql",
         "name": "PostgreSQL_server",
         "ts_ms": 1559033904863,
         "snapshot": null,
         "db": "postgres",
         "schema": "public",
         "table": "customers",
         "txId": 556,
         "lsn": 24023128,
         "xmin": null
      },
      "op": "u",
      "ts_ms": 1465584025523
   }
}
```

これを 挿入 イベントの 値と比較すると、payload セクションにいくつかの違いがあります。

- op フィールドの値は u になっており、更新によってこの行が変更されたことを示しています。
- before フィールドは、データベースのコミット前の行と値の状態を表していますが、プライマリーキー列 ID に対してのみ表示され ます。これは、デフォルトで DEFAULT の REPLICA IDENTITY が原因です。

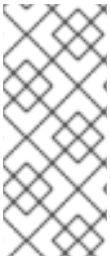


注記

行のすべての列で以前の値を確認する場合は、最初に ALTER TABLE customers REPLICA IDENTITY FULLを実行して customers テーブルを変更する必要があります。

- after フィールドは更新された行の状態を持ち、ここで first_name の値が Anne Marie になっていることを確認できます。
- source フィールド構造には以前と同じフィールドがありますが、このイベントは WAL の異なる位置からのものであるため、値は異なります。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

この ペイロード のセクションを参照することで学習できる点がいくつかあります。before と after の構造を比較 する と、コミットが原因で、この行で実際に何が変更されたかを判断できます。source 構造は、この変更の記録（トレーサビリティーを提供）に関する情報を示しますが、これにはこのトピックや他のイベントの他のイベントと比較できる情報があり、このイベントが他のイベントの前、後、またはその一部として他のイベントとして発生したかどうかを確認できます。



注記

行のプライマリーキー/一意キーの列が更新されると、行のキーの値が変更され、Debezium は 行の古いキーを持つ DELETE イベントと tombstone イベント、行の新しいキーを持つ INSERT イベントという 3 つ のイベントを出力します。

#### 2.2.4.6.6. イベントの削除

ここまでで、create と update イベントのサンプルを確認しました。次に、同じテーブルの削除イベントの値を見てみましょう。ここでも、値の schema 部分は create および update イベントと全く同じになります。

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "id": 1
        },
        "after": null,
        "source": {
            "version": "1.0.3.Final",
            "connector": "postgresql",
            "name": "PostgreSQL_server",
            "ts_ms": 1559033904863,
            "snapshot": null,
            "db": "postgres",
            "schema": "public",
            "table": "customers",
            "txId": 556,
            "lsn": 46523128,
            "xmin": null
        },
        "op": "d",
        "ts_ms": 1465581902461
    }
}
```

ペイロード部分を確認すると、create または updateイベントペイロードと比べて多くの違いがあります。

- op フィールドの値は d になっており、この行が削除されたことを示しています。
- before フィールドは、データベースのコミットで削除した行の状態を表しています。ここでも、REPLICA IDENTITY 設定によるプライマリーキー列のみが含まれます。
- after フィールドが null で、行が存在しなくなったことが分かります。
- source フィールド構造には以前と同じ値が多数ありますが、ts_ms フィールド、lsn フィールド、および txId フィールドは変更されました。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

このイベントでは、この行の削除の処理に使用できるあらゆる種類の情報をコンシューマーに提供します。

警告

PK のないテーブルに注意してください。REPLICA IDENTITY DEFAULT の付いたテーブルからの削除メッセージは、パーツの 前 には指定されません（デフォルトの ID レベルの唯一のフィールドは PK であるため）、完全に空になるようにスキップされます。PK を REPLICA IDENTITY を設定せずにテーブルからのメッセージを FULL レベルに処理できるようにするには、以下を行います。

PostgreSQL コネクターのイベントは、Kafka ログコンパクション と動作するように設計されています。これにより、すべてのキーの最新のメッセージが保持される限り、古いメッセージを削除できます。これにより、トピックに完全なデータセットが含まれ、キーベースの状態のリロードに使用できるようにするとともに、 Kafka がストレージ領域を確保できるようにします。

行が削除された場合でも、Kafka は同じキーを持つ以前のメッセージをすべて削除できるため、上記の削除 イベントの値はログコンパクションで動作します。ただし、メッセージ値が null の場合のみ、Kafka は同じキーを持つ すべてのメッセージ を削除できることを認識します。これを可能にするために、PostgreSQL コネクターは、 null 値以外で同じキーを持つ特別な廃棄( tombstone )イベントで削除 イベントに従います。

#### 2.2.4.7. データ型

上記のように、PostgreSQL コネクターは、行が存在するテーブルのように構造化されたイベントを含む行への変更を表します。イベントには各列値のフィールドが含まれ、その値がイベントでどのように表されるかは、列の PostgreSQL データ型によって異なります。本セクションでは、このマッピングを説明します。

以下の表は、各 PostgreSQL データ型をイベントのフィールド内の リテラル 型およびセマンティック型 にマッピングする方法を示しています。

ここで リテラル型 は、Kafka Connect スキーマタイプ( INT8、INT16、INT32 INT64、FLOAT32、FLOAT64、BOOLEAN、STRING、BYTES、ARRAY、MAP、STRUCT )を使用して値をリテラルで表す方法を記述します。

*セマンティック型 は、フィールドの Kafka Connect スキーマの名前を使用して Kafka Connect スキーマがフィールドの 意味 をキャプチャーする方法を記述します。*

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **BOOLEAN** | **BOOLEAN** | 該当なし | |
| **BIT(1)** | **BOOLEAN** | 該当なし | |
| **ビット(> 1 )、ビットバージング[(M)]** | **BYTES** | **io.debezium.data.Bits** | **length** スキーマパラメーターには、ビット数を表す整数が含まれます。生成される **byte[]** にはビットがリトルエンディアン形式で含まれ、指定数のビットが含まれるようにサイズが指定されます（例：**numBytes = n/8 +(n%8== 0 ?)。0 : 1**）（**n** はビット数） |
| **SMALLINT**, **SMALLSERIAL** | **INT16** | 該当なし | |
| **INTEGER**, **SERIAL** | **INT32** | 該当なし | |
| **BIG SERIAL** | **INT64** | 該当なし | |
| **REAL** | **FLOAT32** | 該当なし | |
| **DOUBLE PRECISION** | **FLOAT64** | 該当なし | |
| **CHAR[(M)]** | **STRING** | 該当なし | |
| **VARCHAR[(M)]** | **STRING** | 該当なし | |
| **CHARACTER[(M)]** | **STRING** | 該当なし | |
| **CHARACTER VARYING[(M)]** | **STRING** | 該当なし | |
| **TIMESTAMPTZ**, **TIMESTAMP WITH TIME ZONE** | **STRING** | **io.debezium.time.ZonedTimestamp** | タイムゾーン情報を含むタイムスタンプの文字列表現。タイムゾーンは GMT です。 |

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TIMETZ**, **TIME WITH TIME ZONE** | **STRING** | **io.debezium.time.ZonedTime** | タイムゾーン情報を含む時間値の文字列表現。タイムゾーンは GMT です。 |
| **INTERVAL [P]** | **INT64** | **io.debezium.time.MicroDuration** (default) | 月平均日に **365.25 / 12.0** 式を使用した時間間隔の概算数（マイクロ秒） |
| **INTERVAL [P]** | **文字列** | **io.debezium.time.Interval** ( **interval.handling.mode** が **string**に設定されている場合) | パターン **P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S に続く interval 値の文字列表現（例： P1Y2M3DT4H5M6.78S）** |
| **BYTEA** | **BYTES** | 該当なし | |
| **JSON**, **JSONB** | **STRING** | **io.debezium.data.Json** | JSON ドキュメント、配列、またはスケーラーの文字列表現が含まれます。 |
| **XML** | **STRING** | **io.debezium.data.Xml** | XML ドキュメントの文字列表現が含まれます。 |
| **UUID** | **STRING** | **io.debezium.data.Uuid** | PostgreSQL UUID 値の文字列表現が含まれます。 |
| **POINT** | **STRUCT** | **io.debezium.data.geometry.Point** | 2 つの **FLOAT64** フィールドを持つ構造 **(x,y)** が含まれます。それぞれはジオメトリックポイントの座標を表します。 |
| **LTREE** | **STRING** | **io.debezium.data.Ltree** | PostgreSQL LTREE 値の文字列表現が含まれます。 |
| **CITEXT** | **STRING** | 該当なし | |
| **INET** | **STRING** | 該当なし | |
| **INT4RANGE** | **STRING** | 該当なし | 整数の範囲 |
| **INT8RANGE** | **STRING** | 該当なし | bigint の範囲 |
| **NUMRANGE** | **STRING** | 該当なし | 数値の範囲 |

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TSRANGE** | **STRING** | 該当なし | タイムゾーンのないタイムスタンプの範囲の文字列表現が含まれます。 |
| **TSTZRANGE** | **STRING** | 該当なし | （ローカルシステム）タイムゾーンを持つタイムスタンプの範囲の文字列表現が含まれます。 |
| **DATERANGE** | **STRING** | 該当なし | 日付範囲の文字列表現が含まれます。上限は常に排他的です。 |
| **ENUM** | **STRING** | **io.debezium.data.Enum** | PostgreSQL ENUM 値の文字列表現が含まれます。許可される値のセットは、許可される という名前のスキーマパラメーターで維持 さ れます。 |

***その他のデータ型マッピングは、以下のセクションで説明します。***

#### *2.2.4.7.1. 時間の値*

***PostgreSQL の TIMESTAMPTZ および TIMETZ データ型（タイムゾーン情報が含まれる）以外に、他の時間型は time.precision.mode 設定プロパティーの値によって異なります。time.precision.mode 設定プロパティーが adaptive (デフォルト) に設定された場合、コネクターは列のデータ型を基に時間型のリテラルおよびセマンティック型を決定し、イベントが正確 にデータベースの値を表すようにします。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **DATE** | **INT32** | **io.debezium.time.Date** | エポックからの日数を表します。 |
| **TIME(1)**, **TIME(2)**, **TIME(3)** | **INT32** | **io.debezium.time.Time** | 午前 0 時から経過した時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **TIME(4)**, **TIME(5)**, **TIME(6)** | **INT64** | **io.debezium.time.MicroTime** | 午前 0 時から経過した時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |
| **TIMESTAMP(1)**, **TIMESTAMP(2)**, **TIMESTAMP(3)** | **INT64** | **io.debezium.time.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TIMESTAMP(4)**, **TIMESTAMP(5)**, **TIMESTAMP(6)**, **TIMESTAMP** | **INT64** | **io.debezium.time.MicroTimestamp** | エポックからの経過時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |

***time.precision.mode 設定プロパティーが adaptive_time_microseconds に設定されている場合、コネクターは列のデータ型を基に時間型のリテラル型とセマンティック型を決定し、イベントが 正確 にデータベースの値を表すようにします。ただし、すべての TIME フィールドはマイクロ秒としてキャプチャーされます。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **DATE** | **INT32** | **io.debezium.time.Date** | エポックからの日数を表します。 |
| **TIME([P])** | **INT64** | **io.debezium.time.MicroTime** | 時間の値をマイクロ秒単位で表し、タイムゾーン情報は含まれません。PostgreSQL では、範囲 が 0 - 6 の精度 **P** が許可され、マイクロ秒の精度まで保存されます。 |
| **TIMESTAMP(1)**, **TIMESTAMP(2)**, **TIMESTAMP(3)** | **INT64** | **io.debezium.time.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **TIMESTAMP(4)**, **TIMESTAMP(5)**, **TIMESTAMP(6)**, **TIMESTAMP** | **INT64** | **io.debezium.time.MicroTimestamp** | エポックからの経過時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |

***time.precision.mode 設定プロパティーが connect に設定された場合、コネクターは事前定義された Kafka Connect の論理型を使用します。これは、コンシューマーが組み込みの Kafka Connect の論理型のみを認識し、可変精度の時間値を処理できない場合に便利です。一方、PostgreSQL はマイクロ秒の精度をサポートするため、connect 時間精度モードでコネクターによって生成されたイベントは、データベース列の 少数秒 の精度値が 3 よりも大きい場合に、精度 が失わ れます。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **DATE** | **INT32** | **org.apache.kafka.connect.data.Date** | エポックからの日数を表します。 |

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TIME([P])** | **INT64** | **org.apache.kafka.connect.data.Time** | 午前 0 時からの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。PostgreSQL では、範囲が 0-6 の **P** が許可され、マイクロ秒の精度まで保存されますが、**P** > 3 では、このモードでは精度が失われます。 |
| **TIMESTAMP([P])** | **INT64** | **org.apache.kafka.connect.data.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。PostgreSQL では、範囲が 0-6 の **P** が許可され、マイクロ秒の精度まで保存されますが、**P** > 3 では、このモードでは精度が失われます。 |

#### *2.2.4.7.2. TIMESTAMP 値*

***TIMESTAMP 型は、タイムゾーン情報のないタイムスタンプを表します。このような列は、UTC を基にして同等の Kafka Connect 値に変換されます。たとえば、2018-06-20 15:13:16.945104 という TIMESTAMP の値は、1529507596945104 という値の io.debezium.time.MicroTimestamp で表されます( time.precision.mode が 接続に設定されていないと仮定します)。***

***Kafka Connect および Debezium を実行している JVM のタイムゾーンは、この変換には影響しないことに注意してください。***

#### *2.2.4.7.3. 10 進数値*

***decimal.handling.mode 設定プロパティーが precise に設定されている場合、コネクターはすべての DECIMAL および NUMERIC 列に事前定義された Kafka Connect org.apache.kafka.connect.data.Decimal 論理型を使用します。これはデフォルトのモードです。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **NUMERIC[(M[,D])]** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **スケーリング** されたスキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。 |
| **DECIMAL[(M[,D])]** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **スケーリング** されたスキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。 |

*このルールには例外があります。NUMERIC または DECIMAL 型がスケーリング制約なしで使用される場合、データベースから取得される値のスケールは値ごとに異なる（変数）スケーリングできることを意味します。この場合、io.debezium.data.VariableScaleDecimal タイプが使用され、転送された値の値とスケールの両方が含まれます。*

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **NUMERIC** | **STRUCT** | **io.debezium.data.VariableScaleDecimal** | 2 つのフィールドを持つ構造が含まれます。type **INT32** 型の **scale** には、転送された値のスケールと、非スケール形式の元の値が含まれる **BYTES** 型の値が含まれます。 |
| **DECIMAL** | **STRUCT** | **io.debezium.data.VariableScaleDecimal** | 2 つのフィールドを持つ構造が含まれます。type **INT32** 型の **scale** には、転送された値のスケールと、非スケール形式の元の値が含まれる **BYTES** 型の値が含まれます。 |

*ただし、decimal.handling.mode 設定プロパティーが double に設定されている場合、コネクターはすべての DECIMAL および NUMERIC 値を Java double 値として表し、以下のようにエンコードします。*

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **NUMERIC[(M[,D])]** | **FLOAT64** | | |
| **DECIMAL[(M[,D])]** | **FLOAT64** | | |

*decimal.handling.mode 設定プロパティーの最後のオプションは string です。この場合、コネクターはすべての DECIMAL および NUMERIC 値をフォーマットされた文字列表現として表し、以下のようにエンコードします。*

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **NUMERIC[(M[,D])]** | **STRING** | | |
| **DECIMAL[(M[,D])]** | **STRING** | | |

***PostgreSQL は、DECIMAL/NUMERIC 値に保存される NaN （数字ではない）固有の値をサポートします。文字 列 および 二重 モードのみが、Double.NaN または文字列定数 NAN のようにエンコードできます。***

#### *2.2.4.7.4. hstore の値*

***hstore.handling.mode 設定プロパティーが json （デフォルト）に設定されている場合、コネクターはすべての HSTORE 値を文字列化された JSON 値として表し、以下のようにエンコードします。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **HSTORE** | **STRING** | **io.debezium.data.Json** | 例：JSON コンバーターを使用した出力表現は **{\"key\" : \"val\"}**です。 |

***hstore.handling.mode 設定プロパティーが マップ に設定された場合、コネクターはすべての HSTORE 列に MAP スキーマタイプを使用します。***

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **HSTORE** | **MAP** | | 例：JSON コンバーターを使用した出力表現は **{"key" : "val"}**です。 |

#### *2.2.4.8. PostgreSQL ドメインタイプ*

***PostgreSQL は、他の基礎となるタイプに基づくユーザー定義の型もサポートしています。このような列型を使用すると、Debezium は完全な型階層に基づいて列の表現を公開します。***

*重要*

*ドメインタイプを使用する列を監視する場合は、特別な考慮する必要があります。*

*デフォルトのデータベースタイプの 1 つを拡張するドメインタイプを使用して列が定義され、ドメインタイプがカスタムの長さ/scale を定義する場合、生成されたスキーマは定義された長さ/scale を継承します。*

*カスタムの長さ/スケールを定義する別のドメインタイプを拡張するドメインタイプを使用して列を定義すると、PostgreSQL ドライバーの列メタデータの実装により、生成されたスキーマは定義された長さ/scale を継承し ません。*

#### *2.2.4.8.1. ネットワークアドレスタイプ*

*PostgreSQL には、IPv4、IPv6、および MAC アドレスを保存できるデータタイプもあります。これらのタイプは入力エラーチェックと特殊演算子および機能を提供するため、プレーンテキスト型の代わりにこれらを使用することが適切です。*

| PostgreSQL データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **INET** | **STRING** | | IPv4 ネットワークおよび IPv6 ネットワーク |
| **CIDR** | **STRING** | | IPv4 および IPv6 のホストおよびネットワーク |
| **MACADDR** | **STRING** | | MAC アドレス |
| **MACADDR 8** | **STRING** | | EUI-64 形式の MAC アドレス |

#### *2.2.4.8.2. PostGIS タイプ*

*また、PostgreSQL コネクターはすべての PostGIS データ型を完全にサポートしています。*

| PostGIS データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|

| PostGIS データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **GEOMETRY** (planar) | **STRUCT** | **io.debezium.data.geometry.Geometry** | 2 つのフィールドを持つ構造が含まれます。<br>• **srid (INT32)** - 構造に保存されるジオメトリーオブジェクトの型を定義する、空間参照システム識別子<br>• **wkb (BYTES)** - Well-Known-Binary 形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。形式の詳細については、Open Geospatial Consortium Simple Features Access specification を参照してください。 |
| **GEOGRAPHY** (spherical) | **STRUCT** | **io.debezium.data.geometry.Geography** | 2 つのフィールドを持つ構造が含まれます。<br>• **srid (INT32)** - 構造に保存されるジオグラフーオブジェクトの型を定義する、空間参照システム識別子<br>• **wkb (BYTES)** - Well-Known-Binary 形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。形式の詳細については、Open Geospatial Consortium Simple Features Access specification を参照してください。 |

#### *2.2.4.8.3. TOAST 化された値*

*PostgreSQL ではページサイズにハード制限があります。つまり、ca より大きい値。8 KB は TOAST ストレージ を使用して保存する必要があります。これは、TOAST メカニズムを使用して保存され、テーブルのレプリカ ID の一部でない限り、変更されていない値がメッセージに含まれていないため、データベースからのレプリケーションメッセージに影響します。Debezium が不足している値を直接データベースから読み取る安全な方法はありません。これにより競合状態が発生する可能性があります。そのため、Debezium は以下のルールに従って、Toasted の値を処理します。*

- *REPLICA IDENTITY FULL: TOAST 列値を持つテーブルは、他の列として変更イベントの before および after ブロックの一部になります。*

- *REPLICA IDENTITY DEFAULT: データベースから UPDATE イベントを受信すると、レプリカ ID の一部ではない変更されていない TOAST 列値は、そのイベントの一部になりません。同様に、DELETE イベントを受信すると、そのような TOAST 列は before ブロックの一部になりません。この場合、Debezium は列値を安全に提供できないため、設定オプション で定義されているプレースホルダー値を返します。これは、asted .value.placeholder です。*

*重要*

*Amazon RDS インスタンスに関連する特定の問題があります。wal2json プラグインは時間とともに進化し、帯域外から貼り付けられた値を提供するリリースがありました。Amazon は、PostgreSQL のバージョンごとに異なるバージョンのプラグインをサポートします。バージョンからバージョンへのマッピングを取得するには、Amazon のドキュメント を参照してください。一貫性のある貼り付けられた値の処理には、以下を推奨します。*

- *PostgreSQL 10+ インスタンス用の pgoutput プラグインの使用*
- *slot.stream.params 設定オプションを使用して、古いバージョンの wal2json プラグインに include-unchanged-toast=0 を設定します。*

### *2.3. POSTGRESQL コネクターのデプロイ*

*PostgreSQL コネクターのインストールは、JAR をダウンロードして Kafka Connect 環境に抽出し、プラグインの親ディレクトリーが Kafka Connect 環境に指定されていることを確認する必要がある単純なプロセスです。*

*前提条件*

- *Zookeeper、Kafka、および Kafka Connect がインストールされている。*
- *PostgreSQL がインストールされ、設定している。*

*手順*

1. *Debezium PostgreSQL コネクター をダウンロードします。*
2. *ファイルを Kafka Connect 環境に展開します。*

3. プラグインの親ディレクトリーを Kafka Connect プラグインパスに追加します。

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

注記

上記の例では、Debezium PostgreSQL コネクターを /kafka/connect/Debezium-connector-postgresql パスに展開したことを前提としています。

4. Kafka Connect プロセスを再起動します。これにより、新しい JAR が確実に選択されるようになります。

関連情報

デプロイメントプロセス、および AMQ Streams でのコネクターのデプロイに関する詳細は、Debezium のインストールガイドを参照してください。

- Debezium の OpenShift へのインストール
- Debezium の RHEL へのインストール

### 2.3.1. 設定例

コネクターを使用して特定の PostgreSQL サーバーまたはクラスターの変更イベントを生成するには、以下を実行します。

1. 論理デコードプラグインのインストール
2. 論理レプリケーションをサポートするように PostgreSQL サーバー を設定する
3. PostgreSQL コネクターの設定ファイルを作成します。

コネクターが起動すると、PostgreSQL サーバーのデータベースの整合性スナップショットを取得し、変更のストリーミングを開始し、挿入、更新、削除されたすべての行に対してイベントを生成しま

*す。スキーマおよびテーブルのサブセットに対してイベントを生成することもできます。必要に応じて、機密、大きすぎる列、または不要な列を無視、マスク、または切り捨てます。*

*以下は、192.168.99.100 のポート 5432 で PostgreSQL サーバーを監視する PostgreSQL コネクターの設定例で、これは fullfillment という論理的な名前になります。通常、コネクターに使用できる設定プロパティーを使用して、.yaml ファイルに Debezium PostgreSQL コネクターを設定します。*

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

1

*コネクターの名前。*

2

*1 度に 1 つのタスクのみが動作する必要があります。PostgreSQL コネクターは PostgreSQL サーバーの 192.168.99.100 を読み取るため、単一のコネクタータスクを使用することで、順序とイベントの処理が適切に行われるようになります。Kafka Connect サービスはコネクターを使用して作業を行う 1 つ以上のタスクを開始し、実行中のタスクを自動的に Kafka Connect サービスのクラスター全体に分散します。いずれかのサービスが停止またはクラッシュすると、これらのタスクは稼働中のサービスに再分散されます。*

3

*コネクターの設定。*

4

*データベースホスト。PostgreSQL サーバーを実行しているコンテナーの名前（）です。*

5

*一意のサーバー名。サーバー名は、PostgreSQL サーバーまたはサーバーのクラスターの論理識別子です。この名前は、すべての Kafka トピックの接頭辞として使用されます。*

6

*public.inventory データベースの変更のみが検出されます。*

*これら の設定で指定できる コネクタープロパティーの完全リスト を参照してください。*

*この設定は、POST 経由で稼働中の Kafka Connect サービスに送信できます。その後、設定を記録し、PostgreSQL データベースに接続し、イベントを Kafka トピックに記録します。*

### *2.3.2. モニタリング*

*Kafka、Zookeeper、および Kafka Connect はすべて JMX メトリクスのサポートが組み込まれています。また、PostgreSQL コネクターは、JMX を介して監視できるコネクターのアクティビティーに関する多数のメトリクスを公開します。コネクターには 2 種類のメトリクスがあります。スナップショットメトリクスは、スナップショットアクティビティーの監視に役立ち、コネクターがスナップショットを実行している場合に利用できます。ストリーミングメトリクスは、コネクターが論理レプリケーションストリームを処理している間に進捗とアクティビティーをモニターするのに役立ちます。*

#### *2.3.2.1. スナップショットメトリクス*

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

| 属性名 | タイプ | 説明 |
|---|---|---|
| LastEvent | string | コネクターが読み取りした最後のスナップショットイベント。 |
| MilliSecondsSinceLastEvent | long | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| TotalNumberOfEventsSeen | long | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| NumberOfEventsFiltered | long | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |
| MonitoredTables | string[] | コネクターによって監視されるテーブルの一覧。 |

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **QueueTotalCapcity** | **int** | snapshotter とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | snapshotter とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |
| **TotalTableCount** | **int** | スナップショットに含まれているテーブルの合計数。 |
| **RemainingTableCount** | **int** | スナップショットによってまだコピーされていないテーブルの数。 |
| **SnapshotRunning** | **boolean** | スナップショットが起動されたかどうか。 |
| **SnapshotAborted** | **boolean** | スナップショットが中断されたかどうか。 |
| **SnapshotCompleted** | **boolean** | スナップショットが完了したかどうか。 |
| **SnapshotDurationInSeconds** | **long** | スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。 |
| **RowsScanned** | **Map<String, Long>** | スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。 |

#### *2.3.2.2. ストリーミングメトリクス*

##### *2.3.2.2.1. MBean: debezium.postres:type=connector-metrics,context=streaming,server=<database.server.name>*

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **LastEvent** | **string** | コネクターが読み取られた最後のストリーミングイベント。 |
| **MilliSecondsSinceLastEvent** | **long** | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| **TotalNumberOfEventsSeen** | **long** | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| **NumberOfEventsFiltered** | **long** | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |

| 属性名 | タイプ | 説明 |
|---|---|---|
| **MonitoredTables** | **string[]** | コネクターによって監視されるテーブルの一覧。 |
| **QueueTotalCapcity** | **int** | streamer とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | ストリーマーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |
| **オンラインインストール** | **boolean** | コネクターが現在データベースサーバーに接続されているかどうかを示すフラグ。 |
| **MilliSecondsBehindSource** | **long** | 最後の変更イベントのタイムスタンプとそれを処理するコネクターとの間の期間 (ミリ秒単位)。値には、データベースサーバーとコネクターが実行されているマシンのクロックの差異が含まれます。 |
| **NumberOfCommittedTransactions** | **long** | コミットされた処理済みトランザクションの数。 |
| **SourceEventPosition** | **map<string, string>** | 最後に受信したイベントの位置。 |
| **LastTransactionId** | **string** | 最後に処理されたトランザクションのトランザクション識別子。 |

### *2.3.3. コネクタープロパティー*

***以下の設定プロパティーは、デフォルト値がない場合は必須です。***

| プロパティー | デフォルト | 説明 |
|---|---|---|
| **name** | | コネクターの一意名。同じ名前で再登録を試みると失敗します。(このプロパティーはすべての Kafka Connect コネクターに必要です) |
| **connector.class** | | コネクターの Java クラスの名前。Postgre SQL コネクターには、常に **io.debezium.connector.postgresql.PostgresConnector** の値を使用してください。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **tasks.max** | **1** | このコネクターのために作成する必要のあるタスクの最大数。PostgreSQL コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。 |
| **plugin.name** | **decoderbufs** | サーバーにインストールされている Postgres 論理デコードプラグイン の名前。サポートされている値は **pgoutput** のみです。<br><br>処理されたトランザクションが非常に大きい場合は、トランザクションにすべての変更が含まれる **JSON** バッチイベントが、サイズが 1 GB のハードコーディングされたメモリーバッファーに収まらないことがあります。このような場合は、トランザクションの変更がすべて PostgreSQL から Debezium とは別のメッセージとして送信されると、so-called **streaming** モードに切り替えることができます。 |
| **slot.name** | **debezium** | プラグインおよびデータベースインスタンスから変更をストリーミングするために作成される Postgres 論理デコードスロットの名前。値は Postgres レプリケーションスロットの命名ルール に準拠する必要があります。**各レプリケーションスロットには名前があり、これには小文字、数字、およびアンダースコア文字が含まれます。"** |
| **slot.drop.on.stop** | **false** | コネクターが順番に終了したときに論理レプリケーションスロットをドロップするかどうか。テストまたは開発環境でのみ **true** に設定する必要があります。スロットを削除すると、WAL セグメントをデータベースで破棄できるため、再起動後にコネクターが停止した WAL の位置から再開できないことがあります。 |
| **publication.name** | **dbz_publication** | **pgoutput** の使用時に変更をストリーミングするために作成される PostgreSQL パブリケーションの名前。<br><br>**すべてのテーブル** を含めるためにこのパブリケーションがまだ存在しない場合は、起動時にこのパブリケーションが作成されます。その後、Debezium は独自の white-/blacklist フィルターリング機能を使用して、変更イベントを該当するテーブル（設定されている場合）に制限します。コネクターユーザーはこのパブリケーションを作成するにはスーパーユーザー権限が必要であるため、通常はパブリケーションを事前に作成することが推奨されます。<br><br>パブリケーションがすでに存在する場合（すべてのテーブルまたはテーブルのサブセットのいずれかに対して）、Debezium は代わりに定義された通りにパブリケーションを使用します。 |
| **database.hostname** | | PostgreSQL データベースサーバーの IP アドレスまたはホスト名。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **database.port** | **5432** | PostgreSQL データベースサーバーのポート番号 (整数)。 |
| **database.user** | | PostgreSQL データベースサーバーへの接続時に使用する PostgreSQL データベースの名前。 |
| **database.password** | | PostgreSQL データベースサーバーへの接続時に使用するパスワード。 |
| **database.dbname** | | 変更をストリーミングする PostgreSQL データベースの名前 |
| **database.server.name** | | 監視対象の特定の PostgreSQL データベースサーバー/クラスターの namespace を識別および提供する論理名。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターからのすべての Kafka トピック名の接頭辞として使用されるためです。英数字とアンダースコアのみを使用する必要があります。 |
| **schema.whitelist** | | 監視するスキーマ名と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないスキーマ名は監視から除外されます。デフォルトでは、システム以外のスキーマがすべて監視されます。**schema.blacklist** と併用できません。 |
| **schema.blacklist** | | 監視から除外されるスキーマ名と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないスキーマ名は、システムスキーマを除き監視されます。**schema.whitelist** と併用できません。 |
| **table.whitelist** | | 監視するテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないテーブルはすべて監視から除外されます。各識別子の形式は schemaName.tableName です。デフォルトでは、コネクターは監視される各スキーマのシステム以外のテーブルをすべて監視します。**table.blacklist** と併用できません。 |
| **table.blacklist** | | 監視から除外されるテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないテーブルはすべて監視されます。各識別子の形式は schemaName.tableName です。**table.whitelist** と併用できません。 |
| **column.blacklist** | | 変更イベントメッセージの値から除外される必要がある列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。列の完全修飾名の形式は schemaName です。tableName.columnName |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **time.precision.mode** | **adaptive** | 時間、日付、およびタイムスタンプは、以下を含む異なる精度の種類で表すことができます。**adaptive**（デフォルト）は、データベース列のタイプに基づいて、ミリ秒、マイクロ秒、またはナノ秒の精度値のいずれかを使用して、データベースの値と全く同じように時間とタイムスタンプ値をキャプチャーします。**adaptive_time_microseconds** は、ミリ秒のいずれかを使用してデータベースの値と全く同じように日付、日時、およびタイムスタンプ値をキャプチャーします。データベース列の型に基づいたマイクロ秒、またはナノ秒の精度値。ただし、常にマイクロ秒としてキャプチャーされる TIME 型フィールドを除き、**connect** は Kafka Connect の Time、Date、および Timestamp の組み込み表現を使用して、常に時間とタイムスタンプ値を表します。これは、データベース列の精度に関わらず、ミリ秒の精度を使用します。時間値 を参照してください。 |
| **decimal.handling.mode** | **precise** | コネクターによる **DECIMAL** 列および **NUMERIC** 列の値の処理方法を指定します。**precise**（デフォルト）はバイナリー形式で変更イベントで表される **java.math.BigDecimal** 値を使用して正確に表します。または **double** は **double** 値を使用して表します。精度が失われる可能性はありますが、非常に簡単に使用できます。**string** オプションは値をフォーマットされた文字列としてエンコードします。これは簡単に使用できますが、実際の型に関するセマンティック情報は失われます。「10 進数値」を参照してください。 |
| **hstore.handling.mode** | **map** | コネクターによる **hstore** 列の値の処理方法を指定します。**map**（デフォルト）は **MAP** を使用して表します。または json は **json 文字** 列を使用して値を表します。**json** オプションは、値を **{"key" : "val"}** などのフォーマットされた文字列としてエンコードします。「hstore の値」を参照してください。 |
| **interval.handling.mode** | **numeric** | コネクターによる **間隔** 列の値の処理方法を指定します。**数値**（デフォルト）はマイクロ **秒の概算数を使用して間隔を表します。文字 列は、文字列パターン表現 P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S**、例： **P1Y2M3DT4H5M6.78S** を使用してそれらを正確に表します。「データ型」を参照してください。 |

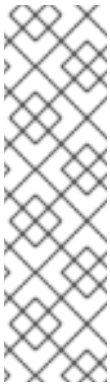
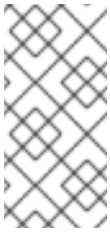
| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **database.sslmode** | **disable** | PostgreSQL サーバーへの暗号化された接続を使用するかどうか。オプションには以下が含まれます。暗号化されていない接続を使用するには を **無効** にします。安全な（暗号化された）接続を使用し、接続を確立できない場合は失敗します。 **require** のような verify-ca は、設定された認証局(CA)証明書に対してサーバー TLS 証明書を追加で検証するか、有効な一致する CA 証明書が見つからない場合は失敗します。verify-full は **verify-ca** としますが、サーバー証明書が接続を試行するホストと一致することを確認します。詳細は PostgreSQL のドキュメント を参照してください。 |
| **database.sslcert** | | クライアントの SSL 証明書を含むファイルへのパス。詳細は PostgreSQL のドキュメント を参照してください。 |
| **database.sslkey** | | クライアントの SSL 秘密鍵が含まれるファイルへのパス。詳細は PostgreSQL のドキュメント を参照してください。 |
| **database.sslpassword** | | **database.sslkey** で指定されたファイルからクライアントの秘密鍵にアクセスするためのパスワード。詳細は PostgreSQL のドキュメント を参照してください。 |
| **database.sslrootcert** | | サーバーが検証されるルート証明書が含まれるファイルへのパス。詳細は PostgreSQL のドキュメント を参照してください。 |
| **database.tcpKeepAlive** | | TCP keep-alive プローブを有効にして、データベース接続がまだ有効であることを確認します（デフォルトでは有効）。詳細は PostgreSQL のドキュメント を参照してください。 |
| **tombstones.on.delete** | **true** | 削除イベント後に廃棄 (tombstone) イベントを生成するかどうかを制御します。<br>**true** の場合、削除操作は 削除 イベントと後続の廃棄 (tombstone)イベントで表されます。**false** の場合、削除イベントのみが送信されます。<br>廃棄 (tombstone) イベントを生成すると (デフォルトの動作)、Kafka はソースレコードが削除されると、指定のキーに関連するすべてのイベントを完全に削除できます。 |
| **column.truncate.to.length.chars** | **該当なし** | フィールド値が指定された文字数より長い場合に、変更イベントメッセージ値で値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数である必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **column.mask.with.length.chars** | **該当なし** | 文字ベースの列の完全修飾名にマッチする正規表現のコンマ区切りリスト (オプション) で、変更イベントメッセージの値を、指定された数のアスタリスク (*) 文字で設定されるフィールド値に置き換える必要があります。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数またはゼロである必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **column.propagate.source.type** | **該当なし** | 出力された変更メッセージの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **message.key.columns** | **空の文字列** | プライマリーキーをマップする完全修飾テーブルおよび列と一致する正規表現のセミコロン区切りリスト。<br>各項目（正規表現）は、カスタムキーを表す完全修飾 **<fully-qualified table>:<a comma-separated list of columns>** と一致する必要があります。<br>特定のコネクターに応じて、完全修飾テーブルは **DB_NAME.TABLE_NAME** または **SCHEMA_NAME.TABLE_NAME** として定義できます。 |

***以下の 高度な 設定プロパティーには、ほとんどの状況で機能する適切なデフォルト設定があるため、コネクターの設定で指定する必要はほとんどありません。***

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.mode** | **Initial** | コネクターの起動時にスナップショットを実行する基準を指定します。デフォルトは **initial** で、論理サーバー名に対してオフセットが記録されていない場合にのみコネクターがスナップショットを実行できます。**always** オプションは、起動時にコネクターが常にスナップショットを実行するように指定します。**never** オプションは、接続でスナップショットを使用しないことを指定し、論理サーバー名を使用して初回起動時に、コネクターは最後に停止した場所（最後の LSN の位置）から読み取るか、論理レプリケーションスロットのビューから最初から開始するように指定します。**initial_only** オプションは、後続の変更を処理せずにコネクターが最初のスナップショットのみを取得し、停止することを指定します。**exported** オプションは、データベーススナップショットがレプリケーションスロットが作成された時点に基づいていることを指定します。これは、ロックのない方法でスナップショットを実行するための優れた方法です。 |
| **snapshot.lock.timeout.ms** | **10000** | スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する正の整数値。この時間間隔でテーブルロックを取得できない場合、スナップショットは失敗します。スナップショットを参照してください。 |
| **snapshot.select.statement.overrides** | | スナップショットに含まれるテーブルの行を制御します。このプロパティーには、完全修飾テーブル (DB_NAME.TABLE_NAME) のコンマ区切りリストが含まれます。各テーブルの select ステートメントは、ID **snapshot.select.statement.overrides.[DB_NAME].[TABLE_NAME]** によって識別される、テーブルごとに 1 つずつ追加の設定プロパティーに指定されます。これらのプロパティーの値は、スナップショットの実行中に特定のテーブルからデータを取得するときに使用する SELECT ステートメントです。**大規模な追加専用テーブルで可能なユースケースとしては、前のスナップショットが中断された場合にスナップショットの開始 (再開) 点を設定することが挙げられます。**<br>注記：この設定はスナップショットにのみ影響します。論理デコーダによって生成されるイベントは、それによる影響を受けません。 |
| **event.processing.failure.handling.mode** | **fail** | イベントの処理中にコネクターが例外に対応する方法を指定します。**fail** は例外 (問題のあるイベントのオフセットを示す) を伝達するため、コネクターが停止します。**warn** を指定すると問題のあるイベントがスキップされ、問題のあるイベントのオフセットがログに記録されます。**skip** を指定すると、問題のあるイベントがスキップされます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **max.queue.size** | **20240** | ストリーミングレプリケーションを介して受信される変更イベントが Kafka に書き込まれる前に配置されるブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などにバックプレシャーを提供できます。 |
| **max.batch.size** | **10240** | このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。 |
| **poll.interval.ms** | **1000** | 各反復処理の実行中に新しい変更イベントが表示されるまでコネクターが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。 |
| **include.unknown.datatypes** | **false** | Debezium がデータタイプが不明なフィールドを満たす場合、デフォルトでは、フィールドは変更イベントから省略され、警告がログに記録されます。フィールドを組み込み、不透明なバイナリー表現のクライアントにダウンストリームを送信して、クライアントが自分でデコードできるようにする方が望ましい場合があります。イベントから不明なデータをフィルターリングする場合は **false** に設定し、バイナリー形式で保持するには **true** に設定します。<br><br>  <br>**注記**<br>クライアントが後方互換性の問題を危険にさらす。リリース間でデータベース固有のバイナリー表現の変更があるだけでなく、最終的に Debezium によってデータタイプがサポートされる場合でも、論理型でダウンストリームに送信され、コンシューマーによる調整が必要になります。一般的に、サポートされていないデータ型に遭遇する場合は、機能リクエストを提出して、サポートを追加できるようにします。 |
| **database.initial.statements** | | データベースへの JDBC 接続（トランザクションログの読み取り接続ではない）が確立されたときに実行される SQL ステートメントのセミコロン区切りリスト。セミコロン(';;')を使用してセミコロンを区切り文字としてではなく、文字として使用します。<br><br>  <br>**注記**<br>コネクターは独自の判断で JDBC 接続を確立するため、通常これはセッションパラメーターの設定にのみ使用してくださいが、DML ステートメントの実行にはは使用しないでください。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **heartbeat.interval.ms** | **0** | ハートビートメッセージが送信される頻度を制御します。このプロパティーには、コネクターがメッセージをハートビートトピックに送信する頻度を定義するミリ秒単位の間隔が含まれます。これは、コネクターがデータベースから変更イベントを受信しているかどうかを監視するために使用できます。また、長期に渡り変更されるのはキャプチャーされていないテーブルのレコードのみである場合は、ハートビートメッセージを利用する必要があります。このような場合、コネクターはデータベースからのログの読み取りを続行しますが、変更メッセージを Kafka に出力しないため、オフセットの更新は Kafka にコミットされません。これにより、（コネクターが実際に処理されているが、最新の LSN をデータベースにフラッシュする機会がない）WAL ファイルがデータベースによって保持され、コネクターの再起動後により多くの変更イベントが再送信される可能性があります。このプロパティーを **0** に設定して、ハートビートメッセージが全く送信されないようにします。<br>デフォルトでは無効にされています。 |
| **heartbeat.topics.prefix** | **__debezium-heartbeat** | ハートビートメッセージが送信されるトピックの命名を制御します。<br>トピックは、**<heartbeat.topics.prefix>.<server.name>** パターンに従って名前が付けられます。 |
| **schema.refresh.mode** | **columns_diff** | テーブルのインメモリースキーマの更新をトリガーする条件を指定します。<br><br>**columns_diff**（デフォルト）は安全なモードで、インメモリースキーマが常にデータベーステーブルのスキーマと同期したままになるようにします。<br><br>**columns_diff_exclude_unchanged_toast** は、変更されていない TOASTable データが完全に不一致がない限り、受信メッセージから派生するスキーマとの間に不一致がある場合に備えて、インメモリースキーマキャッシュを更新するようコネクターに指示します。<br><br>この設定は、更新の一部がほとんどない TOASTed データを持つ頻繁に更新されるテーブルがある場合、コネクターのパフォーマンスが大幅に向上します。ただし、TOASTable 列がテーブルから削除されると、インメモリースキーマが古い状態になる可能性があります。 |
| **snapshot.delay.ms** | | コネクターの起動後、スナップショットを取得するまで待機する間隔 (ミリ秒単位)。<br>クラスター内で複数のコネクターを開始する際に スナップショットが中断されないようにするために使用でき、コネクターのリバランスが実行される可能性があります。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.fetch.size** | **10240** | スナップショットの実行中に各テーブルから 1 度に読み取る必要がある行の最大数を指定します。コネクターは、このサイズの複数のバッチでテーブルの内容を読み取ります。デフォルトは 10240 です。 |
| **slot.stream.params** | | 設定された論理デコードプラグインに渡されるパラメーターのオプションの一覧です。例えば、**add-tables=public.table,public.table2;include-lsn=true** のようにします。 |
| **sanitize.field.names** | コネクター設定が、Avro を使用するように **key.converter** または **value.converter** パラメーターを明示的に指定する場合は **true** です。それ以外の場合のデフォルトは **false** です。 | Avro の命名要件 に準拠するためにフィールド名がサニタイズされるかどうか。詳細は Avro の命名 を参照してください。 |
| **slot.max.retries** | 6 | 試行に失敗した場合にレプリケーションスロットへの接続を再試行する回数。 |
| **slot.retry.delay.ms** | 10000 (10 秒) | コネクターがレプリケーションスロットへの接続に失敗した場合に再試行するまで待機する時間（ミリ秒単位）。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| toasted.value.placeholder | __debezium_unavailable_value | 元の値がデータベースによって提供されないトランピングされた値であることを示す定数を指定します。が hex: 接頭辞で始まる場合は、残りの文字列が 16 進数でエンコードされたオクテットであることが想定されます。補足情報は、セクション を参照してください。 |

*コネクターは、Kafka プロデューサーおよびコンシューマーの作成時に使用される パススルー 設定プロパティーもサポートします。*

*Kafka プロデューサーおよびコンシューマーのすべての設定プロパティーについては、必ず Kafka ドキュメント を参照してください。(PostgreSQL コネクターは 新しいコンシューマー を使用します。)*

### *2.4. POSTGRESQL の一般的な問題*

*Debezium は、複数のアップストリームデータベースのすべての変更をキャプチャーする分散システムであり、イベントの見逃しや損失は発生しません。システムが正常に操作している場合や、慎重に管理されている場合は、Debezium は変更イベントごとに 1 度だけ 配信します。ただし、障害から復旧している間は、変更イベントが繰り返えされる可能性はありますが、障害が発生してもシステムはイベントを失いません。そのため、これらの異常な状況では、Kafka と同様に Debezium は変更イベントを少なくとも 1 回 配信します。*

*本セクションのこれ以降では、Debezium がどのようにさまざまな種類の障害や問題を処理するかを説明します。*

#### *2.4.1. 設定および起動エラー*

*コネクターは起動時に失敗し、コネクターの設定が無効な場合にエラー/例外を報告し、コネクターが指定された接続パラメーターを使用して PostgreSQL に接続できない場合に実行を停止します。コネクターが指定された接続パラメーターを使用して PostgreSQL に接続できない場合、またはコネクターが PostgreSQL WAL (LSN 値経由)の以前に記録された位置から再起動し、PostgreSQL ではその履歴を利用できなくなります。*

*このような場合、エラーには問題の詳細が含まれ、場合によっては回避策が提示されます。設定が修正されたり、PostgreSQL の問題が解決されたときにコネクターを再起動できます。*

#### *2.4.2. PostgreSQL が利用不可になる*

コネクターの実行後、接続先の PostgreSQL サーバーが何らかの理由で利用できなくなると、コネクターはエラーを出して失敗し、コネクターは停止します。サーバーが利用できる場合は、コネクターを再起動するだけです。

PostgreSQL コネクターは、最後に処理されたオフセット(PostgreSQL ログシーケンス番号 値の形式)を外部に保存します。コネクターが再起動し、サーバーインスタンスに接続すると、その特定のオフセットからストリーミングを継続するようサーバーに要求します。このオフセットは、Debezium レプリケーションスロットがそのまま残っている限り、常に利用可能のままになります。プライマリーのレプリケーションスロットを削除しないでください。削除しないと、データが失われます。スロットが削除された場合の障害ケースについては、次のセクションを参照してください。

### 2.4.3. Cluster Failures

12 以降、PostgreSQL は プライマリーサーバーでのみ 論理レプリケーションスロットを許可します。つまり、PostgreSQL コネクターはデータベースクラスターのアクティブなプライマリーにのみポイントできます。また、レプリケーションスロット自体はレプリカに伝播されません。プライマリーノードがダウンした場合、新しいプライマリーが昇格された後に( 論理デコードプラグイン がインストールされた状態で)、レプリケーションスロットが作成された後にのみ、コネクターを再起動して新しいサーバーを参照できます。

フェイルオーバーには非常に重要な注意事項があります。レプリケーションスロットがそのまま残っており、データを失わないことを確認するまで Debezium を一時停止する必要があります。フェイルオーバー後、フェイルオーバーの管理にアプリケーションが 新しい プライマリーへの書き込みが許可される前に Debezium レプリケーションスロットを再作成するプロセスが含まれない限り、変更イベントはありません。また、古いプライマリーが失敗する前 に、Debezium がスロットのすべての変更を読み取ることができるフェイルオーバー状況でも確認する必要がある場合があります。

失われた変更を復元して検証する信頼できる方法（管理が困難）は、失敗したプライマリーのバックアップを、失敗した直ちにその時点に復元することです。これにより、レプリケーションスロットで、消費されていない変更について検査できます。いずれの場合も、書き込みを許可する前に、新しいプライマリーでレプリケーションスロットを再作成することが重要です。

### 2.4.4. Kafka Connect プロセスが正常な停止

Kafka Connect が分散モードで実行され、Kafka Connect プロセスが正常に停止された場合は、Kafka Connect はプロセスのシャットダウン前に、すべてのプロセスのコネクタータスクをそのグループの別の Kafka Connect プロセスに移行し、新しいコネクタータスクは、以前のタスクが停止した場所で開始されます。コネクタータスクが正常に停止され、新しいプロセスで再起動されると、処理に短い遅延が発生します。

### 2.4.5. Kafka Connect プロセスクラッシュ

Kafka Connector プロセスが予期せず停止した場合、最後に処理されたオフセットを記録せずに、実行中のコネクタータスクは明らかに終了します。Kafka Connect が分散モードで実行されている場合は、他のプロセスでこれらのコネクタータスクを再起動します。ただし、PostgreSQL コネクターは以前のプロセスで記録され た最後のオフセットから再開します。つまり、新しい置換タスクにより、クラッシュの直前に処理された同じ変更イベントが生成される可能性があります。重複イベントの数は、オフセットのフラッシュ期間とクラッシュの直前のデータ変更の量によって異なります。

注記

障害からの復旧中に一部のイベントが重複された可能性があるため、コンシューマーは常に一部のイベントが重複している可能性があることを想定する必要があります。Debezium の変更はべき等であるため、一連のイベントは常に同じ状態になります。

Debezium の各変更イベントメッセージには、イベントの送信元に関するソース固有の情報が含まれます。これには、PostgreSQL サーバーのイベントの時間、サーバートランザクションの ID、トランザクションの変更が書き込まれたログ先行書き込みの位置などが含まれます。コンシューマーは、この情報（特に LSN の位置）を追跡し、それらが特定のイベントをすでに認識しているかどうかを確認できます。

### 2.4.6. Kafka が利用不能になる

変更イベントはコネクターによって生成されるため、Kafka Connect フレームワークは、Kafka プロデューサー API を使用してこれらのイベントを記録します。また、Kafka Connect は、これらの変更イベントに表示される最新のオフセットを、Kafka Connect ワーカー設定で指定した頻度で定期的に記録します。Kafka ブローカーが利用できなくなると、コネクターを実行する Kafka Connect ワーカープロセスは Kafka ブローカーへの再接続を繰り返し試行します。つまり、コネクタータスクは接続が再確立されるまで一時停止します。接続が再確立されると、コネクターは停止した場所から再開します。

### 2.4.7. コネクターの期間停止

コネクターが正常に停止された場合、データベースは引き続き使用でき、新しい変更は PostgreSQL WAL に記録されます。コネクターが再起動されると、最後に停止した時点で変更のストリーミングを再開し、コネクターの停止中に発生したすべての変更の変更イベントを記録します。

適切に設定された Kafka クラスターは、大量のスループット を処理できます。Kafka Connect は Kafka のベストプラクティスを使用して記述され、十分なリソースがあれば非常に多くのデータベース変更イベントを処理できます。このため、コネクターがしばらくすると再起動されると、データベースに追いつく可能性が高くなりますが、Kafka の機能やパフォーマンスや PostgreSQL のデータへの変更の量によって異なります。

# 第3章 MONGODB の DEBEZIUM コネクター

重要

テクノロジープレビュー機能は、Red Hat の実稼働環境のサービスレベルアグリーメント (SLA) ではサポートされません。また、機能的に完全ではない可能性があるため、Red Hat はテクノロジープレビュー機能を実稼働環境に実装することは推奨しません。テクノロジープレビュー機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Debezium の MongoDB コネクターは、データベースおよびコレクションにおけるドキュメントの変更に対して、MongoDB レプリカセットまたは MongoDB シャードクラスターを追跡し、これらの変更を Kafka トピックのイベントとして記録します。コネクターは、シャードクラスターにおけるシャードの追加または削除、各レプリカセットのメンバーシップの変更、各レプリカセット内の選出、および通信問題の解決待ちを自動的に処理します。

## 3.1. 概要

MongoDB のレプリケーションメカニズムは冗長性と高可用性を提供し、実稼働環境における MongoDB の実行に推奨される方法です。MongoDB コネクターは、レプリカセットまたはシャードクラスターの変更をキャプチャーします。

MongoDB レプリカセット は、すべてが同じデータのコピーを持つサーバーのセットで設定され、レプリケーションによって、クライアントがレプリカセットの プライマリー のドキュメントに追加したすべての変更が、セカンダリーと呼ばれる別のレプリカセットのサーバーに適用されるようにします。MongoDB のレプリケーションでは、プライマリーが oplog (または操作ログ) に変更を記録した後、各セカンダリーがプライマリーの oplog を読み取って、すべての操作を順番に独自のドキュメントに適用します。新しいサーバーがレプリカセットに追加されると、そのサーバーは最初にプライマリー上のすべてのデータベースおよびコレクションの最初の 同期 を実行し、次にプライマリーの oplog を読み取り、最初の同期の開始後に加えられた可能性のあるすべての変更を適用します。この新しいサーバーは、プライマリーの oplog の最後に到達するとセカンダリーになり、クエリーを処理できます。

MongoDB コネクターはこのレプリケーションメカニズムを使用しますが、実際にはレプリカセットのメンバーにはなりません。ただし、MongoDB のセカンダリーと同様に、コネクターはレプリカセットのプライマリーの oplog を常に読み取ります。また、コネクターが初めてレプリカセットを確認すると、oplog を確認して最後に記録されたトランザクションを取得し、プライマリーのデータベースおよびコレクションの意図的な 同期 を実行します。すべてのデータがコピーされると、コネクターは先に読み込んだ位置から oplog の読み取りを開始します。MongoDB oplog における操作は べき等 であるため、操作の適用回数に関係なく、同じ最終状態になります。

MongoDB コネクターは oplog を処理すると、イベントの発信先の oplog の位置を定期的に記録します。MongoDB コネクターが停止すると、最後に処理した oplog の位置を記録するため、再起動時に

その位置から oplog の読み取りが開始されます。つまり、コネクターを停止、アップグレード、または維持でき、後で再起動できます。イベントを何も失うことなく、停止した場所を正確に特定します。当然ながら、MongoDB の oplogs は通常は最大サイズに制限されているため、コネクターを長時間停止しないようにしてください。長時間停止すると、oplog の操作によってはコネクターによって読み取られる前にパージされる可能性があります。この場合、コネクターは不足している oplog 操作を検出し、初期同期を実行してから oplog の調整に進みます。

MongoDB コネクターは、レプリカセットのメンバーシップとリーダーシップの変更、シャードクラスター内でのシャードの追加と削除、および通信障害の原因となる可能性のあるネットワーク問題にも非常に寛容です。コネクターは常にレプリカセットのプライマリーノードを使用して oplog を調整するため、レプリカセットの選択が行われ、別のノードがプライマリーになると、コネクターは即座に oplog の追跡を停止し、新しいプライマリーに接続し、新しいプライマリーを使用して oplog のチューニングを開始します。同様に、コネクターがレプリカセットのプライマリーとの通信で問題が発生した場合は、再接続を試みます（ネットワークまたはレプリカセットに圧倒しないように指数バックオフを使用）。また、最後に停止した oplog の調整を続行します。これにより、コネクターはレプリカセットメンバーシップの変更を動的に調整でき、通信の失敗を自動的に処理できます。

その他のリソース

- レプリケーションメカニズム
- レプリカセット
- レプリカセットの選出
- シャードクラスター
- シャードの追加
- シャードの削除

### 3.2. MONGODB の設定

MongoDB コネクターは MongoDB の oplog を使用して変更をキャプチャーするため、コネクターは MongoDB レプリカセットと、各シャードが個別のレプリカセットであるシャードクラスターとのみ動作します。レプリカセット または シャードクラスター の設定については、MongoDB ドキュメントを参照してください。また、レプリカセットで アクセス制御と認証 を有効にする方法についても理解するようにしてください。

oplog が読み取られる admin データベースを読み取るために適切なロールを持つ MongoDB ユーザーも必要です。さらに、ユーザーはシャードクラスターの設定サーバーで config データベースを読み取りできる必要もあります。

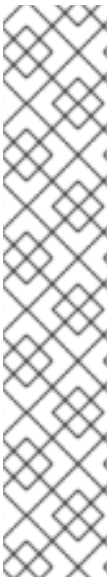
### 3.3. サポートされる MONGODB トポロジー

MongoDB コネクターはさまざまな MongoDB トポロジーで使用できます。

#### 3.3.1. MongoDB レプリカセット

MongoDB コネクターは単一の MongoDB レプリカセット から変更をキャプチャーできます。実稼働のレプリカセットには、少なくとも 3 つのメンバー が必要です。

レプリカセットで MongoDB コネクターを使用するには、コネクターの mongodb.hosts プロパティーを使用して、1 つ以上のレプリカセットサーバーのアドレスを シードアドレス として提供します。コネクターはこれらのシードを使用してレプリカセットに接続した後、レプリカセットからメンバーの完全セットを取得し、どのメンバーがプライマリーであるかを認識します。コネクターは、プライマリーに接続するタスクを開始し、プライマリーの oplog から変更をキャプチャーします。レプリカセットが新しいプライマリーを選出すると、タスクは自動的に新しいプライマリーに切り替えます。



注記

MongoDB がプロキシーと面する場合 (Docker on OS X や Windows などのように)、クライアントがレプリカセットに接続し、メンバーを検出すると、MongoDB クライアントはプロキシーを有効なメンバーから除外し、プロキシーを経由せずに直接メンバーに接続しようとし、失敗します。

このような場合、コネクターのオプションの mongodb.members.auto.discover 設定プロパティーを false に設定して、コネクターにメンバーシップの検出を見送るように指示し、代わりに最初のシードアドレス (mongodb.hosts プロパティーによって指定) をプライマリーノードとして使用するよう指示します。これは機能する可能性がありますが、選出が行われるときに問題が発生します。

#### 3.3.2. MongoDB のシャードクラスター

MongoDB のシャードクラスター は以下で設定されます。

- レプリカセットとしてデプロイされる 1 つ以上のシャード。

- クラスターの設定サーバーとして動作する個別のレプリカセット。

- クライアントが接続し、要求を適切なシャードにルーティングする 1 つ以上の ルーター ( mongos とも呼ばれます)。

シャードクラスターで MongoDB コネクターを使用するには、コネクターを設定サーバーレプリカセットのホストアドレスで設定します。コネクターがこのレプリカセットに接続すると、シャードクラスターの設定サーバーとして動作していることを検出し、クラスターでシャードとして使用される各レプリカセットに関する情報を検出した後、各レプリカセットから変更をキャプチャーするために別のタスクを起動します。新しいシャードがクラスターに追加される場合または既存のシャードが削除される場合、コネクターはそのタスクを自動的に調整します。

### 3.3.3. MongoDB スタンドアロンサーバー

スタンドアロンサーバーには oplog がないため、MongoDB コネクターはスタンドアロン MongoDB サーバーの変更を監視できません。スタンドアロンサーバーが 1 つのメンバーを持つレプリカセットに変換されると、コネクターが動作します。

注記

MongoDB は、実稼働でのスタンドアロンサーバーの実行を 推奨しません。

## 3.4. MONGODB コネクターの仕組み

MongoDB コネクターが設定およびデプロイされると、シードアドレスの MongoDB サーバーに接続して起動し、利用可能な各レプリカセットの詳細を判断します。各レプリカセットには独立した独自の oplog があるため、コネクターはレプリカセットごとに個別のタスクの使用を試みます。コネクターは、使用するタスクの最大数を制限でき、十分なタスクが利用できない場合は、コネクターは各タスクに複数のレプリカセットを割り当てます。ただし、タスクはレプリカセットごとに個別のスレッドを使用します。

注記

シャードクラスターに対してコネクターを実行する場合は、レプリカセットの数よりも大きい tasks.max の値を使用します。これにより、コネクターはレプリカセットごとに 1 つのタスクを作成でき、Kafka Connect が利用可能なワーカープロセス全体でタスクを調整、配布、および管理できるようにします。

### 3.4.1. 論理コネクター名

コネクター設定プロパティー mongodb.name は、MongoDB レプリカセットまたはシャードされたクラスターの 論理名 として提供されます。コネクターは、論理名をさまざまな方法で使用します。すべてトピック名のプレフィックスとして使用したり、各レプリカセットの oplog の位置を記録するときに一意の識別子として使用したりします。

各 MongoDB コネクターに、ソース MongoDB システムを意味する一意の論理名を命名する必要があります。論理名は、アルファベットまたはアンダースコアで始まり、残りの文字を英数字またはアンダースコアとすることが推奨されます。

#### 3.4.2. 初期同期

タスクがレプリカセットを使用して起動すると、コネクターの論理名とレプリカセット名を使用して、コネクターが以前に読み取りを停止したレプリカセット oplog の位置を記述する オフセット を検索します。オフセットが検出され、oplog にある場合は、記録されたオフセット の位置から即座に oplog の追跡を続行します。

ただし、オフセットが見つからない場合や、oplog にその位置が含まれなくなった場合、タスクは最初に最初の 同期 を実行してレプリカセットの内容の現在の状態を取得する必要が あります。このプロセスは、oplog の現在の位置を記録し、オフセット（および最初の同期が開始されたことを示すフラグとともに）として記録します。その後、タスクは各コレクションをコピーし、できるだけ多くのスレッドを生成し( initial.sync.max.threads 設定プロパティーの値まで)、この作業を並行して実行します。コネクターは、確認した各ドキュメントの個別の 読み取りイベント を記録します。読み取りイベントにはオブジェクトの識別子、オブジェクトの完全な状態、およびオブジェクトが見つかった MongoDB レプリカセットの ソース 情報が含まれます。ソース情報には、最初の同期中にイベントが生成されたことを示すフラグも含まれます。

この最初の同期は、コネクターのフィルターに一致するすべてのコレクションをコピーするまで継続されます。タスクの初期同期が完了する前にコネクターが停止した場合、コネクターは再起動時に初期同期を再開します。

注記

コネクターがレプリカセットの意図的な同期を実行している間は、タスクの再割り当てと再設定を避けてください。コネクターは最初の同期の進捗とともにメッセージをログに記録します。最大限の制御を行う場合は、各コネクターに対して Kafka Connect の個別のクラスターを実行します。

#### 3.4.3. oplog の調整

レプリカセットのコネクタータスクがオフセットを持つと、オフセットを使用して読み取りを開始する oplog の位置を判断します。その後、タスクはレプリカセットのプライマリーノードに接続し、その位置から oplog の読み取りを開始し、すべての作成、挿入、および削除操作を処理し、それらを

Debezium 変更イベント に変換します。各変更イベントには操作が検出された oplog の位置が含まれ、コネクターはこれを最新のオフセットとして定期的に記録します。（オフセットが記録される間隔は、offset.flush.interval.ms Kafka Connect ワーカー設定プロパティー によって制御されます）。

コネクターが正常に停止されると、処理された最後のオフセットが記録され、再起動時にコネクターは停止した場所から続行されます。しかし、コネクターのタスクが予期せず終了した場合、最後にオフセットが記録された後、最後のオフセットが記録される前に、タスクによってイベントが処理および生成されることがあります。再起動時に、コネクターは最後に 記録された オフセットから開始し、クラッシュの前に生成された同じイベントを生成する可能性があります。

注記

すべてが通常どおり動作している場合、Kafka コンシューマーは実際にすべてのメッセージを 1 度だけ 確認します。ただし、問題が発生した場合は、Kafka はコンシューマーが 少なくとも 1 度 各メッセージを確認することのみを保証します。したがって、コンシューマーが複数回メッセージを確認することを想定する必要があります。

上記のように、コネクタータスクは常にレプリカセットのプライマリーノードを使用して oplog を維持し、コネクターが可能な限り最新の操作を認識し、代わりにセカンダリーが使用されるよりも短いレイテンシーで変更をキャプチャーできるようにします。レプリカセットが新しいプライマリーを選出すると、コネクターは即座に oplog の追跡を停止し、新しいプライマリーの oplog の調整を開始し、新しいプライマリーの oplog を同じ位置で開始します。同様に、コネクターとレプリカセットメンバーとの通信で問題が発生した場合は、再接続を試みます（レプリカセットを過剰にさらしないように指数バックオフを使用）、接続すると、最後に停止した oplog の調整を続行します。これにより、コネクターはレプリカセットメンバーシップの変更を動的に調整でき、通信の失敗を自動的に処理できます。

下部の行では、MongoDB コネクターはほとんどの状況で引き続き実行されますが、通信の問題により、問題が解決されるまでコネクターが待機する可能性があります。

### 3.4.4. トピック名

MongoDB コネクターは、各コレクションのドキュメントに対するすべての挿入、更新、および削除操作のイベントを 1 つの Kafka トピックに書き込みます。Kafka トピックの名前は常に logicalName.databaseName.collectionName の形式を取ります。logicalName は、mongodb.name 設定プロパティーで指定されるコネクターの 論理名、databaseName は操作が発生したデータベースの名前、collectionName は影響を受けるドキュメントが存在する MongoDB コレクションの名前です。

たとえば、products, products_on_hand, customers, and orders の 4 つのコレクションで設定される inventory データベースを含む MongoDB レプリカセットについて考えてみましょう。コネクターが監視するこのデータベースの論理名が fulfillment である場合、コネクターは以下の 4 つの Kafka トピックでイベントを生成します。

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

*トピック名には、レプリカセット名やシャード名が含まれないことに注意してください。その結果、シャード化コレクションへの変更 (各シャードにコレクションのドキュメントのサブセットが含まれる) はすべて同じ Kafka トピックに移動します。*

*Kafka を設定して、必要に応じてトピックを 自動作成 できます。そうでない場合は、Kafka 管理ツールを使用してコネクターを起動する前にトピックを作成する必要があります。*

### *3.4.5. パーティション*

*MongoDB コネクターは、イベントのトピックパーティションを明示的に決定しません。代わりに、Kafka がキーに基づいてパーティションを判断できるようにします。Kafka Connect ワーカー設定で Partitioner 実装の名前を定義することで、Kafka のパーティショニングロジックを変更できます。*

*Kafka は、1 つのトピックパーティション に書き込まれたイベントの合計順序のみを維持することに注意してください。キーによるイベントのパーティション設定は、同じキーを持つすべてのイベントが常に同じパーティションに移動し、特定のドキュメントのすべてのイベントが常に完全に順序付けられることを意味します。*

### *3.4.6. イベント*

*MongoDB コネクターによって生成されたすべてのデータ変更イベントにはキーと値があります。*

*注記*

*Kafka 0.10 以降、Kafka はオプションでメッセージキーで記録でき、メッセージが作成（プロデューサーによって記録）された タイムスタンプ、または Kafka によってログに書き込まれたタイムスタンプを値として記録できます。*

Debezium および Kafka Connect はイベント メッセージの継続的なストリームを中心に設計されており、これらの イベントのソースが構造で変更された場合や、コネクターが改善または変更される場合に、これらのイベントの構造が時間とともに変更される可能性があります。これは、コンシューマーが処理するのが難しい場合があるため、Kafka Connect が非常に簡単に各イベントの自己完結性を持たせることができます。すべてのメッセージキーと値には、スキーマ と ペイロード の 2 つの部分で設定されます。スキーマはペイロードの構造を記述しますが、ペイロードには実際のデータが含まれます。

#### 3.4.6.1. 変更イベントのキー

特定のコレクションでは、変更イベントのキーには単一の id フィールドが含まれます。この値は、文字列として表されるドキュメントの識別子 で、厳密なモードの MongoDB 拡張 JSON シリアライゼーション から派生します。論理名が fulfillment のコネクター、inventory データベースを含むレプリカセットと、以下のようなドキュメントが含まれる customers コレクションについて考えてみましょう。

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

customers コレクションのすべての変更イベントは、JSON 内の同じキー構造を特長としています。

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

キーの スキーマ 部分には、ペイロード部分の内容を記述する Kafka Connect スキーマが含まれます。この場合、ペイロード 値は任意ではなく、fulfillment.inventory.customers.Key という名前のスキーマによって定義された構造であり、型 string の id という名前の必須フィールドが 1 つあります。

*キーの payload フィールドの値を確認すると、id フィールドが 1 つあり、値は整数 1004 を含む文字列で構造(JSON は単なるオブジェクト)であることがわかります。*

*この例では、整数識別子でドキュメントを使用していますが、有効な MongoDB ドキュメント識別子（ドキュメントを含む）は機能します。ペイロードの id フィールドの値は、元のドキュメントの _id フィールドの MongoDB 拡張 JSON シリアライゼーション（制限モード）を表す文字列になります。以下の例は、異なるタイプの _id フィールドがイベントキーのペイロードとしてエンコードされる方法を示しています。*

| タイプ | MongoDB _id の値 | キーのペイロード |
|---|---|---|
| Integer | 1234 | **{ "id" : "1234" }** |
| 浮動小数点 (Float) | 12.34 | **{ "id" : "12.34" }** |
| 文字列 | "1234" | **{ "id" : "\"1234\"" }** |
| Document | { "hi" : "kafka", "nums" : [10.0, 100.0, 1000.0] } | **{ "id" : "{\"hi\" : \"kafka\", \"nums\" : [10.0, 100.0, 1000.0]}" }** |
| ObjectId | ObjectId("596e275826f08b2730779e1f") | **{ "id" : "{\"$oid\" : \"596e275826f08b2730779e1f\"}" }** |
| バイナリー | BinData("a2Fma2E=",0) | **{ "id" : "{\"$binary\" : \"a2Fma2E=\", \"$type\" : \"00\"}" }** |

#### *3.4.6.2. 変更イベントの値*

*変更イベントメッセージの値は、少し複雑です。キーメッセージと同様に、schema セクションと payload セクションがあります。MongoDB コネクターによって生成されたすべての変更イベント値の payload セクションには、以下のフィールドを含む エンベロープ 構造があります。*

- *op は、操作のタイプを記述する文字列値が含まれる必須フィールドです。MongoDB コネクターの値は、c （作成または挿入）、u （更新）、d （削除）、および r （読み取り（初期同期の場合））です。*
- *after はオプションのフィールドであり、存在する場合はイベント発生 後 のドキュメントの状態が含まれます。MongoDB の oplog エントリーには 作成 イベントのドキュメントの完全な状態のみが含まれるため、これらは after フィールドが含まれるイベントのみです。*

- source は、イベントのソースメタデータを記述する構造が含まれる必須のフィールドです。MongoDB の場合には、Debezium バージョン、論理名、レプリカセット名、コレクションの namespace、MongoDB タイムスタンプ（タイムスタンプ内のイベントの ordinal）、MongoDB 操作の識別子（例：MongoDB 操作の識別子）が含まれます。 oplog イベントの h フィールド、およびイベントが意図的な同期中に発生した場合の初期同期フラグ。

- ts_ms は任意です。存在する場合は、コネクターがイベントを処理した時間(Kafka Connect タスクを実行している JVM のシステムクロックを使用)が含まれます。

当然ながら、イベントメッセージの値のschema 部分には、このエンベロープ構造と、その中のネストされたフィールドを記述するスキーマが含まれます。

customers テーブルの 作成/読み取り イベントの値を見てみましょう。

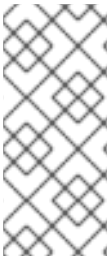
```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "name": "io.debezium.data.Json",
        "version": 1,
        "field": "after"
      },
      {
        "type": "string",
        "optional": true,
        "name": "io.debezium.data.Json",
        "version": 1,
        "field": "patch"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
```

```
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": false,
            "field": "rs"
          },
          {
            "type": "string",
            "optional": false,
            "field": "collection"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "ord"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "h"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mongo.Source",
        "field": "source"
      },
      {
        "type": "string",
        "optional": true,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "dbserver1.inventory.customers.Envelope"
```

```
    },
  "payload": {
    "after": "{\"_id\" : {\"$numberLong\" : \"1004\"},\"first_name\" : \"Anne\",\"last_name\" :
\"Kretchmar\",\"email\" : \"annek@noanswer.org\"}",
    "patch": null,
    "source": {
      "version": "1.0.3.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 31,
      "h": 1546547425148721999
    },
    "op": "r",
    "ts_ms": 1558965515240
  }
}
```

このイベントの値 の スキーマ 部分を確認すると、エンベロープ のスキーマがコレクションに固有のものであること、および ソース 構造のスキーマ(MongoDB コネクターに固有ですべてのイベントで再利用)を確認できます。また、after の値は常に文字列であり、慣例によりドキュメントの JSON 表現が含まれることに注意してください。

このイベントの値 の payload 部分を確認すると、イベントの情報を見ることができます。つまり、初期同期の一部としてドキュメントが読み取られたことが記述されています( op=r および initsync=true以降)。また、after フィールドの値にドキュメントの JSON 文字列表現が含まれていることを確認します。



注記

イベントの JSON 表現はそれが記述する行よりもはるかに大きいように見えることがあります。JSON 表現にはメッセージのスキーマ 部分と ペイロード 部分を含める必要があるため、これは True です。

このコレクションの更新 変更イベントの値は、実際にはまったく同じ スキーマ を持ち、そのペイロードは同じですが、異なる値を保持します。具体的には、更新 イベントには after の値がなく、代わりにべき等更新操作の JSON 表現が含まれる パッチ 文字列があります。以下に例を示します。

```
{
  "schema": { ... },
  "payload": {
    "op": "u",
    "ts_ms": 1465491461815,
```

```
  "patch": "{\"$set\":{\"first_name\":\"Anne Marie\"}}",
  "source": {
    "version": "1.0.3.Final",
    "connector": "mongodb",
    "name": "fulfillment",
    "ts_ms": 1558965508000,
    "snapshot": true,
    "db": "inventory",
    "rs": "rs0",
    "collection": "customers",
    "ord": 6,
    "h": 1546547425148721999
  }
 }
}
```

これを 挿入 イベントの 値と比較すると、payload セクションにいくつかの違いがあります。

- op フィールドの値は u になっており、更新によってこのドキュメントが変更されたことを示しています。
- パッチ フィールドが表示され、ドキュメントに対する実際の MongoDB べき等変更の文字列化された JSON 表現があります。この例では、first_name フィールドを新しい値に設定する必要があります。
- after フィールドが表示されなくなる
- source フィールド構造には以前と同じフィールドがありますが、このイベントは oplog の異なる位置にあるため、値は異なります。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

*警告*

*patch フィールドの内容は MongoDB 自体で提供され、正確な形式は特定のデータベースバージョンによって異なります。したがって、MongoDB インスタンスを新しいバージョンにアップグレードする際に、形式の変更の可能性を準備する必要があります。*

*本書のすべての例は MongoDB 3.4 から取得され、別のサンプルを使用する場合は異なる場合があります。*

*注記*

*MongoDB の oplog の更新イベントには変更されたドキュメントの before または after 状態がないため、コネクターがこの情報を提供する方法はありません。ただし、create または read イベントには 開始状態が含まれるため、ストリームのダウンストリームコンシューマーは、各ドキュメントの最新状態を維持し、各イベントをその状態に適用することで、実際に状態を完全に再構築できます。Debezium コネクターはこのような状態を維持できないため、これを行うことができません。*

*これまでは、作成/読み取り と 更新 イベントの例を見てきました。次に、同じテーブルの 削除 イベントの値を見てみましょう。このコレクションの 削除 イベントの値には全く同じ スキーマ があり、そのペイロードは同じですが、異なる値を保持します。特に、削除 イベントには after の値や patch の値は含まれません。*

```
{
  "schema": { ... },
  "payload": {
    "op": "d",
    "ts_ms": 1465495462115,
    "source": {
      "version": "1.0.3.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
      "ord": 6,
      "h": 1546547425148721999
    }
  }
}
```

これを他のイベントの値と比較すると、payload セクションにいくつかの違いがあります。

- op フィールドの値は d になっており、このドキュメントが削除されたことを示しています。
- パッチ フィールドが表示されない
- after フィールドが表示されない
- source フィールド構造には以前と同じフィールドがありますが、このイベントは oplog の異なる位置にあるため、値は異なります。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

MongoDB コネクターは実際には他の種類のイベントを提供します。各 削除 イベントの後に、同じキーだが null 値を持つ廃棄( tombstone )イベントの後に、Kafka ログコンパクションメカニズムがそのキーを持つ すべて のメッセージを削除できることを示す十分な情報を Kafka に提供します。

注記

MongoDB コネクターイベントはすべて Kafka ログコンパクション と動作するように設計されています。これにより、すべてのキーの最新のメッセージが保持される限り、古いメッセージを削除できます。これは、トピックに完全なデータセットが含まれ、キーベースの状態のリロードに使用できるようにするとともに、Kafka がストレージ領域を回収する方法です。

一意に識別されたドキュメントの MongoDB コネクターイベントはすべて同じキーを持ち、最新のイベントのみが保持される Kafka に通知されます。また、tombstone イベントは、同じキーを持つ すべて のメッセージを削除できることを Kafka に通知します。

### 3.5. DEPLOYING THE MONGODB CONNECTOR

MongoDB コネクターのインストールは、JAR をダウンロードして Kafka Connect 環境に抽出し、プラグインの親ディレクトリーが Kafka Connect 環境に指定されていることを確認する必要がある単

純なプロセスです。

前提条件

- Zookeeper、Kafka、および Kafka Connect がインストールされている。
- MongoDB がインストールされ、設定されていること。

手順

1. Debezium MongoDB コネクター をダウンロードします。
2. ファイルを Kafka Connect 環境に展開します。
3. プラグインの親ディレクトリーを Kafka Connect プラグインパスに追加します。

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

注記

上記の例では、Debezium MongoDB コネクターを /kafka/connect/Debezium-connector-mongodb パスに展開したことを前提としています。

4. Kafka Connect プロセスを再起動します。これにより、新しい JAR が確実に選択されるようになります。

関連情報

デプロイメントプロセス、および AMQ Streams でのコネクターのデプロイに関する詳細は、Debezium のインストールガイドを参照してください。

- Debezium の OpenShift へのインストール
- Debezium の RHEL へのインストール

### 3.5.1. 設定例

コネクターを使用して特定の MongoDB レプリカセットまたはシャードクラスターの変更イベントを生成するには、JSON で設定ファイルを作成します。コネクターが起動すると、MongoDB レプリカセットでコレクションの初期同期を実行し、レプリカセットの oplogs の読み取りを開始し、挿入、更新、および削除されたすべての行に対してイベントを生成します。任意で、不必要なコレクションを除外します。

以下は、MongoDB レプリカセット rs0 を 192.168.99.100 のポート 27017 で監視する MongoDB コネクターの設定例で、論理的に fullfillment という名前が付けられます。通常、コネクターに使用できる設定プロパティーを使用して、.yaml ファイルに Debezium MongoDB コネクターを設定します。

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

1

Kafka Connect サービスに登録する場合のコネクターの名前。

2

MongoDB コネクタークラスの名前。

3

MongoDB レプリカセットへの接続に使用するホストアドレス。

4

生成されたイベントの namespace を形成する MongoDB レプリカセットの論理名。コネクターが書き込む Kafka トピックの名前、Kafka Connect スキーマ名、および Avro コネクターが使用される場合に対応する Avro スキーマの namespace のすべてに使用されます。

5

監視するすべてのコレクションのコレクション namespace (例: <dbName>.<collectionName>) と一致する正規表現のリスト。これは任意です。

*これら の設定で指定できる コネクタープロパティーの完全リスト を参照してください。*

*この設定は、POST 経由で稼働中の Kafka Connect サービスに送信できます。その後、設定を記録し、MongoDB レプリカセットまたはシャードクラスターに接続するコネクタータスクを 1 つ起動し、各レプリカセットにタスクを割り当て、oplog を読み取り、Kafka トピックにイベントを記録します。*

### *3.5.2. コネクタープロパティー*

*以下の設定プロパティーは、デフォルト値がない場合は必須 です。*

| プロパティー | デフォルト | 説明 |
|---|---|---|
| **name** | | コネクターの一意名。同じ名前で再登録を試みると失敗します。(このプロパティーはすべての Kafka Connect コネクターに必要です) |
| **connector.class** | | コネクターの Java クラスの名前。MongoDB コネクターには、常に **io.debezium.connector.mongodb.MongoDbConnector** の値を使用します。 |
| **mongodb.hosts** | | レプリカセットでの MongoDB サーバーのホスト名とポートのペア ('host' または 'host:port' 形式) のコンマ区切りリスト。リストには、ホスト名とポートのペアを 1 つ含めることができます。**mongodb.members.auto.discover** を **false** に設定すると、ホストとポートには、レプリカセット名 (**rs0/localhost:27017**) を接頭辞として付ける必要があります。 |
| **mongodb.name** | | このコネクターが監視するコネクターや MongoDB レプリカセット、またはシャードクラスターを識別する一意の名前。このサーバー名は、MongoDB レプリカセットまたはクラスターから生成される永続化されたすべての Kafka トピックの接頭辞になるため、各サーバーは最大 1 つの Debezium コネクターによって監視される必要があります。英数字とアンダースコアのみを使用する必要があります。 |
| **mongodb.user** | | MongoDB への接続時に使用されるデータベースユーザーの名前。これは MongoDB が認証を使用するように設定されている場合にのみ必要です。 |
| **mongodb.password** | | MongoDB への接続時に使用されるパスワード。これは MongoDB が認証を使用するように設定されている場合にのみ必要です。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **mongodb.ssl.enabled** | **false** | コネクターは SSL を使用して MongoDB インスタンスに接続します。 |
| **mongodb.ssl.invalid.hostname.allowed** | **false** | SSL が有効な場合、接続フェーズ中に厳密なホスト名のチェックを無効にするかどうかを制御する設定です。**true** に設定すると、接続で中間者攻撃は阻止されません。 |
| **database.whitelist** | **空の文字列** | 監視するデータベース名と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないデータベース名は監視から除外されます。デフォルトでは、すべてのデータベースが監視されます。**database.blacklist** と併用できません。 |
| **database.blacklist** | **空の文字列** | 監視から除外されるデータベース名と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないデータベース名が監視されます。**database.whitelist** と併用できません。 |
| **collection.whitelist** | **空の文字列** | 監視する MongoDB コレクションの完全修飾 namespace と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないコレクションはすべて監視から除外されます。各識別子の形式は databaseName.collectionName です。デフォルトでは、**local** および **admin** データベースにあるコレクションを除くすべてのコレクションがコネクターによって監視されます。**collection.blacklist** と併用できません。 |
| **collection.blacklist** | **空の文字列** | 監視から除外される MongoDB コレクションの完全修飾 namespace と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないコレクションはすべて監視されます。各識別子の形式は databaseName.collectionName です。**collection.whitelist** と併用できません。 |
| **snapshot.mode** | **Initial** | コネクターの起動時にスナップショット（初期同期など）を実行する基準を指定します。デフォルトは **initial** で、オフセットが見つからない場合や oplog に以前のオフセットが含まれなくなった場合にコネクターがスナップショットを読み取るように指定します。**never** オプションは、コネクターはスナップショットを使用せずに、ログをの追跡を続行すべきであることを指定します。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **field.blacklist** | **空の文字列** | 変更イベントメッセージ値から除外される必要があるフィールドの完全修飾名のコンマ区切りリスト (任意)。フィールドの完全修飾名の形式は**databaseName.collectionName.fieldName.nestedFieldName** で、**databaseName** および **collectionName** にはすべての文字と一致するワイルドカード (*) が含まれることがあります。 |
| **field.renames** | **空の文字列** | イベントメッセージ値のフィールドの名前を変更するために使用されるフィールドの完全修飾置換のコンマ区切りリスト (任意)。フィールドの完全修飾置換の形式は **databaseName.collectionName.fieldName.nestedFieldName:newNestedFieldName** で、**databaseName** および **collectionName** にはすべての文字と一致するワイルドカード (*) が含まれることがあります。コロン (:) は、フィールドの名前変更マッピングを決定するために使用されます。次のフィールドの置換は、リストの前のフィールド置換の結果に適用されるため、同じパスにある複数のフィールドの名前を変更する場合は、この点に注意してください。 |
| **tasks.max** | **1** | このコネクターのために作成する必要のあるタスクの最大数。MongoDB コネクターは各レプリカセットに個別のタスクの使用しようとします。そのため、コネクターを単一の MongoDB レプリカセットと使用する場合は、デフォルトを使用できます。MongoDB のシャードクラスターでコネクターを使用する場合、クラスターのシャード数以上の値を指定して、各レプリカセットの作業が Kafka Connect によって分散されるようにすることが推奨されます。 |
| **initial.sync.max.threads** | **1** | レプリカセットでコレクションの最初の同期を実行するために使用されるスレッドの最大数を指定する正の整数値。デフォルトは 1 です。 |
| **tombstones.on.delete** | **true** | 削除イベント後に廃棄 (tombstone) イベントを生成するかどうかを制御します。<br>**true** の場合、削除操作は 削除 イベントと後続の廃棄 (tombstone)イベントで表されます。**false** の場合、削除イベントのみが送信されます。<br>廃棄 (tombstone) イベントを生成すると (デフォルトの動作)、Kafka はソースレコードが削除されると、指定のキーに関連するすべてのイベントを完全に削除できます。 |
| **snapshot.delay.ms** | | コネクターの起動後、スナップショットを取得するまで待機する間隔 (ミリ秒単位)。<br>クラスター内で複数のコネクターを開始する際に スナップショットが中断されないようにするために使用でき、コネクターのリバランスが実行される可能性があります。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.fetch.size** | **0** | スナップショットの実行中に各コレクションから 1 度に読み取る必要があるドキュメントの最大数を指定します。コネクターは、このサイズの複数のバッチでコレクションの内容を読み取ります。<br>デフォルトは 0 で、サーバーが適切なフェッチサイズを選択することを示します。 |

***以下の高度な設定プロパティーには、ほとんどの状況で機能する適切なデフォルト設定があるため、コネクターの設定で指定する必要はほとんどありません。***

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **max.queue.size** | **8192** | データベースログから読み取られた変更イベントが Kafka に書き込まれる前に配置される、ブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに、oplog リーダーにバックプレシャーを提供できます。キューに発生するイベントは、このコネクターによって定期的に記録されるオフセットには含まれません。デフォルトは 8192 で、常に **max.batch.size** プロパティーに指定された最大バッチサイズよりも大きくする必要があります。 |
| **max.batch.size** | **2048** | このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。デフォルトは 2048 です。 |
| **poll.interval.ms** | **1000** | 各反復処理の実行中に新しい変更イベントが表示されるまでコネクターが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。 |
| **connect.backoff.initial.delay.ms** | **1000** | 最初に失敗した接続試行の後またはプライマリーが利用できない場合に、プライマリーへの再接続を試行するときの最初の遅延を指定する正の整数値。デフォルトは 1 秒 (1000 ミリ秒) です。 |
| **connect.backoff.max.delay.ms** | **1000** | 接続試行に繰り返し失敗した後またはプライマリーが利用できない場合に、プライマリーへの再接続を試行するときの最大遅延を指定する正の整数値。デフォルトは 120 秒 (120,000 ミリ秒) です。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **connect.max.attempts** | **16** | レプリカセットのプライマリーへの接続を試行する場合の最大失敗回数を指定する正の整数値。この値を越えると、例外が発生し、タスクが中止されます。デフォルトは 16。**connect.backoff.initial.delay.ms** と **connect.backoff.max.delay.ms** のデフォルト値では、20 分強試行した後にのみ失敗します。 |
| **mongodb.members.auto.discover** | **true** | 'mongodb.hosts' 内のアドレスがクラスターまたはレプリカセットの全メンバーを検出するために使用されるシードであるかどうか (**true**)、または **mongodb.hosts** のアドレスをそのまま使用する必要があるかどうか (**false**) を指定するブール値。デフォルトは **true** で、MongoDB がプロキシーと面する 場合を除き、すべてのケースで使用する必要があります。 |
| **heartbeat.interval.ms** | **0** | ハートビートメッセージが送信される頻度を制御します。このプロパティーには、コネクターがメッセージをハートビートトピックに送信する頻度を定義するミリ秒単位の間隔が含まれます。これは、コネクターがデータベースから変更イベントを受信しているかどうかを監視するために使用できます。また、長期に渡り変更されるのはキャプチャーされていないコレクションのレコードのみである場合は、ハートビートメッセージを利用する必要があります。このような場合、コネクターはデータベースからの oplog の読み取りを続行しますが、変更メッセージを Kafka に出力しないため、オフセットの更新が Kafka にコミットされません。これにより、oplog ファイルがローテーションされますが、コネクターはこれを認識しないため、再起動時に一部のイベントが利用できなくなり、最初のスナップショットの再実行が必要になります。<br><br>このプロパティーを **0** に設定して、ハートビートメッセージが全く送信されないようにします。<br>デフォルトでは無効にされています。 |
| **heartbeat.topics.prefix** | **__debezium-heartbeat** | ハートビートメッセージが送信されるトピックの命名を制御します。<br>トピックは、**<heartbeat.topics.prefix>.<server.name>** パターンに従って名前が付けられます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **sanitize.field.names** | コネクター設定が、Avro を使用するように **key.converter** または **value.converter** パラメーターを明示的に指定する場合は **true** です。それ以外の場合のデフォルトは **false** です。 | Avro の命名要件に準拠するためにフィールド名がサニタイズされるかどうか。 |

### *3.6. MONGODB コネクターの一般的な問題*

*Debezium は、複数のアップストリームデータベースのすべての変更をキャプチャーする分散システムであり、イベントの見逃しや損失は発生しません。システムが正常に操作している場合や、慎重に管理されている場合は、Debezium は変更イベントごとに 1 度だけ 配信します。ただし、障害から復旧している間は、変更イベントが繰り返えされる可能性はありますが、障害が発生してもシステムはイベントを失いません。よって、このような正常でない状態では、Debezium は Kafka と同様に、変更イベントを 少なくとも 1 回 配信します。*

*本セクションのこれ以降では、Debezium がどのようにさまざまな種類の障害や問題を処理するかを説明します。*

#### *3.6.1. 設定および起動エラー*

*コネクターの設定が無効な場合や、指定の接続パラメーターを使用してコネクターが繰り返し*

*MongoDB への接続に失敗する場合は、コネクターは起動時に失敗し、エラーや例外をログに報告し、そして、実行を停止します。。再接続は指数バックオフを使用して行われ、試行の最大数は設定可能です。*

*このような場合、エラーには問題の詳細が含まれ、場合によっては回避策が提示されます。設定が修正されたり、MongoDB の問題が解決された場合はコネクターを再起動できます。*

### *3.6.2. MongoDB が使用不可能になる*

*コネクターが実行され、MongoDB レプリカセットのプライマリーノードが利用できなくなったり、アクセスできなくなったりすると、コネクターは指数バックオフを使用してプライマリーノードへの再接続を繰り返し試み、ネットワークやサーバーが飽和状態にならないようにします。設定可能な接続試行回数を超えた後もプライマリーが利用できない状態である場合、コネクターは失敗します。*

*再接の続試行は、3 つのプロパティーで制御されます。*

- *connect.backoff.initial.delay.ms - 初回の再接続を試みるまでの遅延。デフォルトは 1 秒 (1000 ミリ秒) です。*
- *connect.backoff.max.delay.ms - 再接続を試行するまでの最大遅延。デフォルトは 120 秒 (120,000 ミリ秒) です。*
- *connect.max.attempts - エラーが生成されるまでの最大試行回数。デフォルトは 16 です。*

*各遅延は、最大遅延以下で、前の遅延の 2 倍です。以下の表は、デフォルト値を指定した場合の、失敗した各接続試行の遅延と、失敗前の合計累積時間を表しています。*

| 再接続試行回数 | 試行までの遅延 (秒単位) | 試行までの遅延合計 (分および秒単位) |
|---|---|---|
| 1 | 1 | 00:01 |
| 2 | 2 | 00:03 |
| 3 | 4 | 00:07 |
| 4 | 8 | 00:15 |

| 再接続試行回数 | 試行までの遅延 (秒単位) | 試行までの遅延合計 (分および秒単位) |
|---|---|---|
| 5 | 16 | 00:31 |
| 6 | 32 | 01:03 |
| 7 | 64 | 02:07 |
| 8 | 120 | 04:07 |
| 9 | 120 | 06:07 |
| 10 | 120 | 08:07 |
| 11 | 120 | 10:07 |
| 12 | 120 | 12:07 |
| 13 | 120 | 14:07 |
| 14 | 120 | 16:07 |
| 15 | 120 | 18:07 |
| 16 | 120 | 20:07 |

#### *3.6.3. Kafka Connect のプロセスは正常に停止する*

*Kafka Connect が分散モードで実行され、Kafka Connect プロセスが正常に停止された場合は、Kafka Connect はプロセスのシャットダウン前に、すべてのプロセスのコネクタータスクをそのグループの別の Kafka Connect プロセスに移行し、新しいコネクタータスクは、以前のタスクが停止した場所で開始されます。コネクタータスクが正常に停止され、新しいプロセスで再起動されるまでの間、プロセスに短い遅延が発生します。*

*グループにプロセスが 1 つだけあり、そのプロセスが正常に停止された場合、Kafka Connect はコネクターを停止し、各レプリカセットの最後のオフセットを記録します。再起動時に、レプリカセットタスクは停止した場所で続行されます。*

#### *3.6.4. Kafka Connect プロセスのクラッシュ*

*Kafka Connector プロセスが予期せず停止した場合、最後に処理されたオフセットを記録せずに、実行中のコネクタータスクが終了します。Kafka Connect が分散モードで実行されている場合は、他のプロセスでこれらのコネクタータスクを再起動します。ただし、MongoDB コネクターは以前のプロセスによって記録 された最後のオフセットから再開します。つまり、新しい代替タスクによって、ク*

ラッシュの直前に処理された同じ変更イベントが生成される可能性があります。重複するイベントの数は、オフセットのフラッシュ期間とクラッシュの直前のデータ変更の量によって異なります。

注記

障害からの復旧中に一部のイベントが重複された可能性があるため、コンシューマーは常に一部のイベントが重複している可能性があることを想定する必要があります。Debezium の変更はべき等であるため、一連のイベントは常に同じ状態になります。

Debezium の各変更イベントメッセージには、イベントの生成元に関するソース固有の情報が含まれます。これには、MongoDB イベントの一意なトランザクション識別子 (h) やタイムスタンプ (sec and ord) が含まれます。コンシューマーはこれらの値の他の部分を追跡し、特定のイベントがすでに発生しているかどうかを確認することができます。

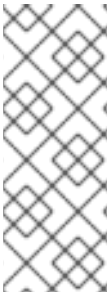
### 3.6.5. Kafka が使用不可能になる

変更イベントはコネクターによって生成されるため、Kafka Connect フレームワークは、Kafka プロデューサー API を使用してこれらのイベントを記録します。また、Kafka Connect は、これらの変更イベントに表示される最新のオフセットを、Kafka Connect ワーカー設定で指定した頻度で定期的に記録します。Kafka ブローカーが利用できなくなると、コネクターを実行する Kafka Connect ワーカープロセスは Kafka ブローカーへの再接続を繰り返し試行します。つまり、コネクタータスクは接続が再確立されるまで一時停止します。接続が再確立されると、コネクターは停止した場所から再開します。

### 3.6.6. コネクターの一定期間の停止

コネクターが正常に停止された場合、レプリカセットは引き続き使用でき、新しい変更は MongoDB の oplog に記録されます。コネクターが再起動されると、最後に停止した各レプリカセットの oplog の読み取りを再開し、コネクターが停止した間に加えられたすべての変更の記録イベントを記録します。コネクターが長時間停止し、コネクターが読み取っていない一部の操作を MongoDB が oplog からパージする場合、コネクターは起動時に最初の同期を実行します。

Kafka クラスターを適切に設定すると、大量のスループット が可能になります。Kafka Connect は Kafka のベストプラクティスを使用して記述され、十分なリソースがあれば非常に多くのデータベース変更イベントを処理できます。そのため、コネクターがしばらくして再起動されると、データベースに追いつく可能性が非常に高くなりますが、遅れを取り戻すまでに掛かる時間は、Kafka の機能やパフォーマンスおよび MongoDB のデータへの変更の量に応じて異なります。



*注記*

*コネクターが長時間停止した場合、MongoDB が古い oplog ファイルをパージし、コネクターの最後の位置が失われる可能性があります。この場合、最初 のスナップショットモード (デフォルト) で設定されたコネクターが最終的に再起動されると、MongoDB サーバーには開始点がなくなり、コネクターはエラーによって失敗します。*

### *3.6.7. MongoDB による書き込みの損失*

*MongoDB は、特定の障害状況でコミットを失う可能性があります。たとえば、プライマリーが変更を適用し、それを oplog に記録した後に予期せずクラッシュした場合、セカンダリーノードはプライマリーがクラッシュした前にプライマリーの oplog からこれらの変更を読み取りできなかった可能性があります。このようなセカンダリーの 1 つがプライマリーとして選出されると、古いプライマリーが記録された最後の変更がなく、それらの変更が行われなくなります。*

*MongoDB でプライマリーの oplog に記録された変更が失われた場合、MongoDB コネクターが失われた変更をキャプチャーしたかどうかは定かではありません。現時点では、MongoDB のこの副次的な影響を防ぐ方法はありません。*

# 第4章 SQL SERVER の DEBEZIUM コネクター

重要

テクノロジープレビュー機能は、Red Hat の実稼働環境のサービスレベルアグリーメント (SLA) ではサポートされません。また、機能的に完全ではない可能性があるため、Red Hat はテクノロジープレビュー機能を実稼働環境に実装することは推奨しません。テクノロジープレビュー機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Debezium の SQL Server コネクターは、SQL Server データベースのスキーマで行レベルの変更を監視および記録できます。

SQL Server データベース/クラスターに初めて接続すると、すべてのスキーマの整合性スナップショットが読み込まれます。スナップショットが完了すると、コネクターは SQL Server にコミットされた変更を継続的にストリーミングし、対応する insert、update、および delete イベントを生成します。各テーブルのすべてのイベントは、アプリケーションやサービスで簡単に使用できる個別の Kafka トピックに記録されます。

## 4.1. 概要

コネクターの機能は、SQL Server Standard によって提供される 変更データキャプチャー 機能 (SQL Server 2016 SP1 以降)または Enterprise の編集に基づいています。このメカニズムを使用すると、SQL Server キャプチャープロセスは、ユーザーが関心のあるすべてのデータベースおよびテーブルを監視し、変更をストアドプロシージャーファサードで特別に作成された CDC テーブルに保存します。コネクターは SQL Server 2017 でテストされていますが、コミュニティーメンバーは 2014 までの以前のバージョンで正常に使用されました(CDC 機能が提供される限り)。

データベース Operator は、コネクターによってキャプチャーされる必要があるテーブルの CDC を有効 にする必要があります。その後、コネクターは CDC API 経由で公開されたすべての行レベルの insert、update、および delete 操作の 変更イベント を生成し、個別の Kafka トピックの各テーブルの変更イベントをすべて記録します。クライアントアプリケーションは、対象のデータベーステーブルに対応する Kafka トピックを読み取り、これらのトピックに表示されるすべての行レベルのイベントに対応します。

データベース Operator は通常、データベース an/またはテーブルの中間期間で CDC を有効にします。つまり、コネクターにはデータベースに加えられたすべての変更の完全な履歴はありません。したがって、SQL Server コネクターが最初に特定の SQL Server データベースに接続すると、データベーススキーマごとに 整合性スナップショット を実行して起動します。コネクターは、スナップショットの完成後に、スナップショットが作成された正確な時点から変更のストリーミングを続行します。これ

により、すべてのデータの一貫したビューから開始しますが、スナップショットの実行中に加えられた変更を失うことなく読み取りを続行します。

コネクターはフォールトトラレントでもあります。コネクターは変更を読み取り、イベントを生成するため、CDC レコードに関連するデータベースログの位置(LSN / ログシーケンス番号)を記録します。コネクターが何らかの理由で停止した場合（通信障害、ネットワークの問題、クラッシュなど）、再起動時に最後に停止した CDC テーブルの読み取りを続行します。これにはスナップショットが含まれます。コネクターの停止時にスナップショットが完了しなかった場合、再起動時に新しいスナップショットが開始されます。

### 4.2. SQL SERVER の設定

SQL Server コネクターを使用して SQL Server でコミットされた変更を監視する前に、最初に監視対象のデータベースで CDC を有効にします。CDC は マスター データベースに対して有効にできないことに注意してください。

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

次に、監視する各テーブルに対して CDC を有効にします。

```
-- =========
-- Enable a Table Specifying Filegroup Option Template
-- =========
USE MyDB
GO

EXEC sys.sp_cdc_enable_table
@source_schema = N'dbo',
@source_name   = N'MyTable',
@role_name     = N'MyRole',
@filegroup_name = N'MyDB_CT',
@supports_net_changes = 0
GO
```

ユーザーが CDC テーブルにアクセスできることを確認します。

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

```
EXEC sys.sp_cdc_help_change_data_capture
GO
```

結果が空の場合は、ユーザーがキャプチャーインスタンスと CDC テーブルの両方にアクセスする権限を持っていることを確認してください。

#### 4.2.1. Azure 上の SQL Server

SQL Server プラグインは Azure の SQL Server でテストされています。管理環境のデータベースでプラグインを試すために、ユーザーからのフィードバックをお寄せください。

### 4.3. SQL SERVER コネクターの仕組み

#### 4.3.1. スナップショット

SQL Server CDC は、データベース変更の完全な履歴を保存するように設計されていません。そのため、Debezium は現在のデータベースコンテンツのベースラインを確立し、それを Kafka にストリーミングする必要があります。これは、snapshotting と呼ばれるプロセスを使用して実行されます。

デフォルトでは（スナップショットモードの最初の）、コネクターは最初の起動時にデータベースの最初の 整合性スナップショット を実行します（コネクターのフィルター設定に従ってキャプチャーされるテーブルの構造およびデータになります）。

各スナップショットは以下の手順で設定されます。

1. キャプチャーするテーブルの決定
2. 監視される各テーブルでロックを取得し、テーブルの構造が変更されないようにします。ロックのレベルは、snapshot.isolation.mode 設定プロパティーによって決定されます。
3. サーバーのトランザクションログの最大 LSN ("log sequence number")の位置を読み取ります。
4. 関連するテーブルの構造をすべてキャプチャーします。
5.

*必要に応じて、手順 2 で取得したロックを解放します。つまり、ロックは通常短期間のみ保持されます。*

6. *ステップ 3 で読み取られた LSN の位置で有効なものとして、関連するデータベーステーブルとスキーマをすべてスキャンし、各行の READ イベントを生成し、そのイベントを適切なテーブル固有の Kafka トピックに書き込みます。*

7. *コネクターオフセットにスナップショットの正常な完了を記録します。*

#### *4.3.2. 変更データテーブルの読み取り*

*初回起動時に、コネクターはキャプチャーされたテーブルの構造のスナップショットを取得し、内部データベース履歴トピックでこの情報を永続化します。その後、コネクターは各ソーステーブルの変更テーブルを特定し、メインループを実行します。*

1. *変更テーブルごとに、最後に保存された最大 LSN から現在の最大 LSN の間に作成された変更をすべて読み取ります。*

2. *コミット LSN および変更 LSN に従って、読み取り変更を段階的に並べ替えます。これにより、変更がデータベースに加えられたのと同じ順序で Debezium によって再生されるようになります。*

3. *コミット LSN および変更 LSN をオフセットとして Kafka Connect に渡します。*

4. *最大 LSN を保存し、ループを繰り返します。*

*再起動後、コネクターは以前に停止したオフセット（コミットおよび変更 LSN）から再開します。*

*コネクターは、実行時にホワイトリスト化されたソーステーブルに対して CDC を有効または無効にするかどうかを検出し、その動作を変更できます。*

#### *4.3.3. トピック名*

*SQL Server コネクターは、単一のテーブルのすべての挿入、更新、および削除操作のイベントを単一の Kafka トピックに書き込みます。Kafka トピックの名前は常に* serverName *の形式を取ります。*schemaName *は*tableName *です。*serverName *は database.server.name 設定プロパティーで指*

定したコネクターの論理名で、schemaName は操作が発生したスキーマの名前、tableName は操作が発生したデータベーステーブルの名前です。

たとえば、製品、products_on_hand、customers 、および schema dbo の 4 つのテーブルが含まれる inventory データベースを含む SQL Server のインストールについて考えてみましょう。コネクターが監視するこのデータベースの論理サーバー名 fulfillment が指定されている場合、コネクターは以下の 4 つの Kafka トピックでイベントを生成します。

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

#### 4.3.4. スキーマ変更トピック

ユーザーに表示されるスキーマ変更トピックはまだ実装されていません({jira-url}/browse/DBZ-1904[DBZ-1904] を参照)。

#### 4.3.5. イベント

SQL Server コネクターによって生成されたすべてのデータ変更イベントにはキーと値がありますが、キーと値の構造は変更イベントの発生元となるテーブルによって異なります( Topic namesを参照)。

*警告*

*SQL Server コネクターは、すべての Kafka Connect スキーマ名が 有効な Avro スキーマ名 になるようにします。つまり、論理サーバー名はラテン文字またはアンダースコア（例 : [a-z,A-Z,_]）で開始し、スキーマおよびテーブル名の残りの文字（例 : [a-z,A-Z,0-9,_]）で始まり、ラテン文字、数字、またはアンダースコア（例 : [a-z,A-Z,0-9,_]）で始まる必要があります。そうでない場合は、すべての無効な文字が自動的にアンダースコア文字に置き換えられます。*

*これにより、論理サーバー名、スキーマ名、およびテーブル名に他の文字が含まれ、テーブルのフルネームを区別する唯一の文字が無効になり、アンダースコアに置き換えられたため、予期せぬ競合が発生する可能性があります。*

*Debezium および Kafka Connect はイベント メッセージの継続的なストリームに基づいて設計されており、これらのイベント の構造は時間の経過とともに変更される可能性があります。これは、コンシューマーが処理するのが困難な場合があるため、Kafka Connect を容易にすることで、各イベントを自己完結させることができます。すべてのメッセージキーと値には、スキーマ と ペイロード の 2 つの部分で設定されます。スキーマはペイロードの構造を記述しますが、ペイロードには実際のデータが含まれます。*

#### *4.3.5.1. イベントキーの変更*

*特定のテーブルでは、変更イベントのキーの構造には、イベントの作成時にテーブルのプライマリーキー（または一意のキー制約）の各列のフィールドが含まれます。*

*inventory データベースのスキーマ dbo で定義された customers テーブルについて考えてみましょう。*

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

*database.server.name 設定プロパティーの値が server1 の場合、この定義がある限り customers テーブルの変更イベントはすべて同じキー構造を特長とし、JSON では以下のようになります。*

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

キーの スキーマ 部分には、キーの部分の内容を記述する Kafka Connect スキーマが含まれます。この場合、ペイロード 値はオプションではなく、server1.dbo.customers.Key という名前のスキーマによって定義された構造であり、タイプ int32 の id という名前の必須フィールドが 1 つあります。キーの payload フィールドの値を確認すると、id フィールドが 1 つあり、その値が 1004 である構造 (JSON では単なるオブジェクト)になっていることがわかります。

そのため、このキーは、ID プライマリーキー列の値が 1004 である dbo.customers テーブルの行(server1という名前のコネクターからの出力)を記述するものとして解釈されます。

#### 4.3.5.2. 変更イベント値

message キーと同様に、変更イベントメッセージの値には schema セクションと payload セクションがあります。SQL Server コネクターによって生成されたすべての変更イベント値の payload セクションには、以下のフィールドを含む エンベロープ 構造があります。

- op は、操作のタイプを記述する文字列値が含まれる必須フィールドです。SQL Server コネクターの値は、c （作成または挿入）、u （更新）、d （削除）、および r （読み取り、スナップショットの場合）です。
- before は任意のフィールドであり、存在する場合はイベント発生 前 の行の状態が含まれます。この構造は、server1.dbo.customers.Value Kafka Connect スキーマによって記述され、server1 コネクターは dbo.customers テーブルのすべての行に使用します。
- after はオプションのフィールドで、存在する場合はイベント発生 後 の行の状態が含まれます。この構造は、の 前 に で使用される同じ server1.dbo.customers.Value Kafka Connect スキーマによって記述されます。

- source は、イベントのソースメタデータを記述する構造が含まれる必須のフィールドです。SQL Server の場合は、Debezium バージョン、コネクター名、イベントが進行中のスナップショットの一部であるかどうか、コミット LSN （スナップショット中ではない）、変更が発生した変更、データベース、スキーマ、テーブルの LSN が含まれます。 ソースデータベースでレコードが変更された時点を表すタイムスタンプ（スナップショット中は、スナップショットの時点になります）。

  また、ストリーミング中にフィールド event_serial_no が存在します。これは、同じコミットおよび変更 LSN を持つイベントを区別するために使用されます。値が 1 と異なる場合は、主に 2 つの状況を確認できます。

  - 更新 イベントの値は 2 に設定されます。これは、更新が SQL Server の CDC 変更テーブルに 2 つのイベントを生成するためです(ソースドキュメント)。最初の値には古い値が含まれ、2 番目の値には新しい値が含まれます。そのため、最初の値は破棄され、2 番目の値は Debezium 変更イベントの作成に使用されます。

  - プライマリーキーが更新されると、SQL Server は 2 つのレコードを出力します。delete は古いプライマリーキー値を持つレコードを削除し、新しいプライマリーキーでレコードを作成するために 挿入 します。どちらの操作も同じコミットおよび変更 LSN を共有します。イベント番号は 1 と 2 です。

- ts_ms は任意です。存在する場合は、コネクターがイベントを処理した時間(Kafka Connect タスクを実行している JVM のシステムクロックを使用)が含まれます。

当然ながら、イベントメッセージの値のschema 部分には、このエンベロープ構造と、その中のネストされたフィールドを記述するスキーマが含まれます。

#### 4.3.5.2.1. 作成 イベント

customers テーブルの create イベント値を見てみましょう。

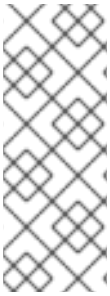
```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
```

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

```
        {
          "type": "string",
          "optional": false,
          "field": "connector"
        },
        {
          "type": "string",
          "optional": false,
          "field": "name"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "ts_ms"
        },
        {
          "type": "boolean",
          "optional": true,
          "default": false,
          "field": "snapshot"
        },
        {
          "type": "string",
          "optional": false,
          "field": "db"
        },
        {
          "type": "string",
          "optional": false,
          "field": "schema"
        },
        {
          "type": "string",
          "optional": false,
          "field": "table"
        },
        {
          "type": "string",
          "optional": true,
          "field": "change_lsn"
        },
        {
          "type": "string",
          "optional": true,
          "field": "commit_lsn"
        },
        {
          "type": "int64",
          "optional": true,
          "field": "event_serial_no"
        }
      ],
      "optional": false,
      "name": "io.debezium.connector.sqlserver.Source",
      "field": "source"
    },
```

```
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "server1.dbo.customers.Envelope"
  },
  "payload": {
    "before": null,
    "after": {
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "source": {
      "version": "1.0.3.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
      "event_serial_no": "1"
    },
    "op": "c",
    "ts_ms": 1559729471739
  }
}
```

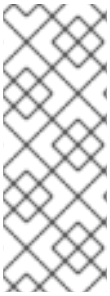
*このイベントの値 の スキーマ 部分を確認すると、エンベロープ の スキーマ、ソース 構造のスキーマ(SQL Server コネクターに固有ですべてのイベントで再利用)、before および after フィールドのテーブル固有のスキーマを確認できます。*



*注記*

*before および after フィールドのスキーマ名は* **logicalName.schemaName** *の形式であるため、.Value は他のすべてのテーブルのスキーマから完全に独立しています。つまり、Avro コンバーター を使用する場合、各 論理ソース の 各テーブル の Avro スキーマには独自の進化と履歴があります。*

*このイベントの値 の ペイロード 部分を確認すると、イベント内の情報、行が作成されたことを説明するものが表示されます( op=c以降)、after フィールドの値には新しい挿入された行の ID、first_name、last_name、および email 列の値が含まれます。*



*注記*

*イベントの JSON 表現はそれが記述する行よりもはるかに大きいように見えることがあります。JSON 表現にはメッセージのスキーマ 部分と ペイロード 部分を含める必要があるため、これは True です。を使用して、Kafka トピックに書き込まれた実際のメッセージのサイズを大幅に小さくすることもできます。*

#### *4.3.5.2.2. 更新イベント*

*このテーブルの 更新 変更イベントの値は、実際にはまったく同じ スキーマ を持ち、そのペイロードは同じですが、異なる値を保持します。以下に例を示します。*

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "after": {
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "source": {
      "version": "1.0.3.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
      "event_serial_no": "2"
    },
    "op": "u",
    "ts_ms": 1559729998706
  }
}
```

これを 挿入 イベントの 値と比較すると、payload セクションにいくつかの違いがあります。

- op フィールドの値は u になっており、更新によってこの行が変更されたことを示しています。
- before フィールドは、データベースのコミット前の行と値の状態を表しています。
- after フィールドは、更新された行の状態を持ち、ここで 電子メール 値が noreply@example.org になったことを確認できます。
- source フィールド構造には以前と同じフィールドがありますが、このイベントはトランザクションログの異なる位置からのものであるため、値は異なります。
- event_serial_no フィールドの値は 2 です。これは、内向きの 2 つのイベントで設定される update イベントが原因で、2 番目のイベントのみを公開します。詳細は、ソースのドキュメント を確認し、$operation フィールドを参照してください。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

この ペイロード のセクションを参照することで学習できる点がいくつかあります。before と after の構造を比較 する と、コミットが原因で、この行で実際に何が変更されたかを判断できます。source 構造は、この変更の記録（トレーサビリティーを提供）に関する情報を示しますが、これには、このトピックや他のトピックの他のイベントと比較できる情報が含まれており、このイベントが他のイベントの前、後、または一部として他のイベントとして発生したかどうかを確認できます。

注記

行のプライマリーキー/一意キーの列が更新されると、行のキーの値が変更されたため、Debezium は DELETE イベントと、行の古いキーを持つ tombstone イベント と、行の新しいキーを持つ INSERT イベントという 3 つ のイベントを出力します。

#### 4.3.5.2.3. 削除イベント

ここまでで、create と update イベントのサンプルを確認しました。次に、同じテーブルの 削除 イベントの値を見てみましょう。ここでも、値の schema 部分は create および update イベントと全く同じになります。

```
{
  "schema": { ... },
 },
 "payload": {
  "before": {
    "id": 1005,
    "first_name": "john",
    "last_name": "doe",
    "email": "noreply@example.org"
  },
  "after": null,
  "source": {
    "version": "1.0.3.Final",
    "connector": "sqlserver",
    "name": "server1",
    "ts_ms": 1559730445243,
    "snapshot": false,
    "db": "testDB",
    "schema": "dbo",
    "table": "customers",
    "change_lsn": "00000027:00000db0:0005",
    "commit_lsn": "00000027:00000db0:0007",
    "event_serial_no": "1"
  },
  "op": "d",
  "ts_ms": 1559730450205
 }
}
```

ペイロード 部分を確認すると、create または updateイベントペイロードと比べて多くの違いがあります。

- op フィールドの値は d になっており、この行が削除されたことを示しています。
- before フィールドは、データベースのコミットで削除した行の状態を表しています。
- after フィールドが null で、行が存在しなくなったことが分かります。
- source フィールド構造には以前と同じ値が多数ありますが、ts_ms フィールド、commit_lsn フィールド、および change_lsn フィールドが変更されました。
- ts_ms は、Debezium がこのイベントを処理したタイムスタンプを示します。

このイベントでは、この行の削除の処理に使用できるあらゆる種類の情報をコンシューマーに提供します。

SQL Server コネクターのイベントは、Kafka ログコンパクション と動作するように設計されています。これにより、すべてのキーの最新のメッセージが保持される限り、古いメッセージを削除できます。これにより、トピックに完全なデータセットが含まれ、キーベースの状態のリロードに使用できるようにするとともに、 Kafka がストレージ領域を確保できるようにします。

行が削除された場合でも、Kafka は同じキーを持つ以前のメッセージをすべて削除できるため、上記の削除 イベントの値はログコンパクションで動作します。ただし、メッセージ値が null の場合のみ、Kafka は同じキーを持つ すべてのメッセージ を削除できることを認識します。これを可能にするには、SQL Server コネクターは、 null 値以外で同じキーを持つ特別な廃棄( tombstone )イベントで削除 イベントに従います。

### 4.3.6. データベーススキーマの進化

Debezium は、時間の経過とともにスキーマの変更をキャプチャーできます。CDC が SQL Server に実装される方法により、スキーマの更新時にコネクターがデータ変更イベントの生成を継続するには、データベース Operator と連携して作業する必要があります。

前述のように、Debezium は SQL Server の変更データキャプチャー機能を使用します。これは、SQL Server がソーステーブルで実行されるすべての変更が含まれるキャプチャーテーブルを作成することを意味します。ただし、キャプチャーテーブルは静的であるため、ソーステーブル構造が変更された場合に更新する必要があります。この更新はコネクター自体によって実行されませんが、昇格された権限を持つ Operator によって実行する必要があります。

通常、スキーマの変更を実行する方法は 2 つあります。

- Cold: これは Debezium が停止したときに実行されます。
- hot - Debezium の実行中に実行されます

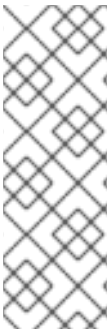
どちらのアプローチも、それぞれ長所と短所があります。

*警告*

*いずれの場合も、同じソーステーブルで新しいスキーマが更新される前に、手順を完全に実行することが重要です。そのため、手順が一度だけ実行されるように、すべての DDL を 1 つのバッチで実行することが推奨されます。*

*注記*

*CDC がソーステーブルに対して有効になっている場合、スキーマの変更がすべてサポートされるわけではありません。このような例外の 1 つが列の名前を変更したり、そのタイプを変更したりすることです。SQL Server では操作を実行できません。*



*注記*

*SQL Server の CDC メカニズム自体では必要ありませんが、列を NULL から NOT NULL に変更する場合やその逆の場合、新しいキャプチャーインスタンスを作成する必要があります。これは、SQL Server コネクターが変更された情報を選択できるようにするために必要です。それ以外の場合は、出力される変更イベントには、元の値に一致するように対応するフィールド(true または false)の オプション の値が設定されます。*

#### *4.3.6.1. コールドスキーマの更新*

*これは最も安全な手順ですが、高可用性要件のあるアプリケーションでは実行できない可能性があります。オペレーターは、以下のステップに従う必要があります。*

1. *データベースレコードを生成するアプリケーションを一時停止します。*
2. *Debezium がストリーミングされていないすべての変更をストリーミングするのを待機します。*
3. *コネクターを停止する*
4. *ソーステーブルスキーマにすべての変更を適用します。*
5.

パラメーター @capture_instanceの一意の値で sys.sp_cdc_enable_table の手順を使用して、更新ソーステーブルの新しいキャプチャーテーブルを作成します。

6. アプリケーションの再開

7. コネクターの起動

8. Debezium が新しいキャプチャーテーブルからストリーミングを開始する場合、パラメーター @capture_instance を古いキャプチャーインスタンス名に設定した sys.sp_cdc_disable_table ストアドプロシージャーを使用すると、古いキャプチャーテーブルからストリーミングを削除できます。

#### 4.3.6.2. ホットスキーマの更新

ホットスキーマの更新では、アプリケーションとデータ処理のダウンタイムは必要ありません。この手順自体は、コールドスキーマ更新の場合よりもはるかに簡単です。

1. ソーステーブルスキーマにすべての変更を適用します。

2. パラメーター @capture_instanceの一意の値で sys.sp_cdc_enable_table の手順を使用して、更新ソーステーブルの新しいキャプチャーテーブルを作成します。

3. Debezium が新しいキャプチャーテーブルからストリーミングを開始する場合、パラメーター @capture_instance を古いキャプチャーインスタンス名に設定した sys.sp_cdc_disable_table ストアドプロシージャーを使用すると、古いキャプチャーテーブルからストリーミングを削除できます。

ホットスキーマの更新には、欠点が 1 つあります。データベーススキーマの更新と新しいキャプチャーインスタンスの作成の間には、期間があります。この期間中に到達するすべての変更は、古い構造を持つ古いインスタンスによってキャプチャーされます。たとえば、これは、新たに追加された列に、この時間内に生成された変更イベントには、その新しい列のフィールドがまだ含まれないことを意味します。アプリケーションがこのような移行期間を許容しない場合は、コールドスキーマの更新に従うことを推奨します。

#### 4.3.6.3. 例

この例では、customers テーブルに列 phone_number が追加されます。

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

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

-- Create the new capture instance
EXEC sys.sp_cdc_enable_table @source_schema = 'dbo', @source_name = 'customers',
@role_name = NULL, @supports_net_changes = 0, @capture_instance = 'dbo_customers_v2';
GO

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

Kafka Connect ログには、以下のようなメッセージが含まれます。

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

最終的には、スキーマに新しい フィールドがあり、Kafka トピックに書き込まれたメッセージの値があります。

```
...
    {
      "type": "string",
      "optional": true,
      "field": "phone_number"
    }
...
  "after": {
```

```
    "id": 1005,
    "first_name": "John",
    "last_name": "Doe",
    "email": "john.doe@example.com",
    "phone_number": "+1-555-123456"
  },
```

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

#### 4.3.7. データタイプ

上記のように、SQL Server コネクターは、行が存在するテーブルのように構造化されたイベントを含む行への変更を表します。イベントには各列値のフィールドが含まれ、その値がイベントでどのように表されるかは、列の SQL データ型によって異なります。本セクションでは、このマッピングを説明します。

以下の表は、各 SQL Server データ型をイベントのフィールド内の リテラル 型とセマンティック型にマッピングする方法を示しています。ここで リテラル型 は、Kafka Connect スキーマタイプ( INT8、INT16、INT32 INT64、FLOAT32、FLOAT64、BOOLEAN、STRING、BYTES、ARRAY、MAP、STRUCT )を使用して値をリテラルで表す方法を記述します。セマンティック型 は、フィールドの Kafka Connect スキーマの名前を使用して Kafka Connect スキーマがフィールドの 意味 をキャプチャーする方法を記述します。

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **BIT** | **BOOLEAN** | 該当なし | |
| **TINYINT** | **INT16** | 該当なし | |
| **SMALLINT** | **INT16** | 該当なし | |
| **INT** | **INT32** | 該当なし | |
| **BIGINT** | **INT64** | 該当なし | |
| **REAL** | **FLOAT32** | 該当なし | |
| **FLOAT[(N)]** | **FLOAT64** | 該当なし | |
| **CHAR[(N)]** | **STRING** | 該当なし | |

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **VARCHAR[(N)]** | **STRING** | 該当なし | |
| **TEXT** | **STRING** | 該当なし | |
| **NCHAR[(N)]** | **STRING** | 該当なし | |
| **NVARCHAR[(N)]** | **STRING** | 該当なし | |
| **NTEXT** | **STRING** | 該当なし | |
| **XML** | **STRING** | **io.debezium.data.Xml** | XML ドキュメントの文字列表現が含まれます。 |
| **DATETIMEOFFSET[(P)]** | **STRING** | **io.debezium.time.ZonedTimestamp** | タイムゾーン情報を含むタイムスタンプの文字列表現。タイムゾーンは GMT です。 |

*その他のデータ型マッピングは、以下のセクションで説明します。*

*列のデフォルト値がある場合は、対応するフィールドの Kafka Connect スキーマに伝達されます。変更メッセージには、フィールドのデフォルト値が含まれます (明示的な列値が指定されていない場合)。そのため、スキーマからデフォルト値を取得する必要はほとんどありません。*

#### *4.3.7.1. 時間値*

*タイムゾーン情報が含まれる SQL Server の DATETIMEOFFSET 以外の時間型は、time.precision.mode 設定プロパティーの値によって異なります。time.precision.mode 設定プロパティーが adaptive (デフォルト) に設定された場合、コネクターは列のデータ型を基に時間型のリテラルおよびセマンティック型を決定し、イベントが正確 にデータベースの値を表すようにします。*

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **DATE** | **INT32** | **io.debezium.time.Date** | エポックからの日数を表します。 |
| **TIME(0)**, **TIME(1)**, **TIME(2)**, **TIME(3)** | **INT32** | **io.debezium.time.Time** | 午前 0 時から経過した時間をミリ秒で表し、タイムゾーン情報は含まれません。 |

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TIME(4)**, **TIME(5)**, **TIME(6)** | **INT64** | **io.debezium.time.MicroTime** | 午前 0 時から経過した時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |
| **TIME(7)** | **INT64** | **io.debezium.time.NanoTime** | 午前 0 時から経過した時間をナノ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME** | **INT64** | **io.debezium.time.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **SMALLDATETIME** | **INT64** | **io.debezium.time.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME2(0)**, **DATETIME2(1)**, **DATETIME2(2)**, **DATETIME2(3)** | **INT64** | **io.debezium.time.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME2(4)**, **DATETIME2(5)**, **DATETIME2(6)** | **INT64** | **io.debezium.time.MicroTimestamp** | エポックからの経過時間をマイクロ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME2(7)** | **INT64** | **io.debezium.time.NanoTimestamp** | エポックからの経過時間をナノ秒で表し、タイムゾーン情報は含まれません。 |

***time.precision.mode 設定プロパティーが connect に設定された場合、コネクターは事前定義された Kafka Connect の論理型を使用します。これは、コンシューマーが組み込みの Kafka Connect の論理型のみを認識し、可変精度の時間値を処理できない場合に便利です。一方で、SQL Server はマイクロ秒の 10 分の 1 の精度をサポートするため、connect 時間精度モードでコネクターによって生成されたイベントは、データ列の 少数秒の精度 値が 3 よりも大きい場合に 精度が失われます。***

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **DATE** | **INT32** | **org.apache.kafka.connect.data.Date** | エポックからの日数を表します。 |

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **TIME([P])** | **INT64** | **org.apache.kafka.connect.data.Time** | 午前 0 時からの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。SQL Server では、範囲 0 - 7 の **P** が許可され、マイクロ秒の 10 分の 1 の精度まで保存されますが、**P** > 3 では、このモードでは精度が失われます。 |
| **DATETIME** | **INT64** | **org.apache.kafka.connect.data.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **SMALLDATETIME** | **INT64** | **org.apache.kafka.connect.data.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。 |
| **DATETIME2** | **INT64** | **org.apache.kafka.connect.data.Timestamp** | エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。SQL Server では、範囲 0 - 7 の **P** が許可され、マイクロ秒の 10 分の 1 の精度まで保存されますが、**P** > 3 では、このモードでは精度が失われます。 |

#### *4.3.7.1.1. タイムスタンプ値*

***DATETIME、SMALLDATETIME および DATETIME2 タイプは、タイムゾーン情報のないタイムスタンプを表します。このような列は、UTC を基にして同等の Kafka Connect 値に変換されます。たとえば、2018-06-20 15:13:16.945104 という DATETIME2 の値は、1529507596945104 という値の io.debezium.time.MicroTimestamp で表されます。***

***Kafka Connect および Debezium を実行している JVM のタイムゾーンは、この変換には影響しないことに注意してください。***

#### *4.3.7.2. 10 進数値*

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|

| SQL Server データ型 | リテラル型 (スキーマ型) | セマンティック型 (スキーマ名) | 注記 |
|---|---|---|---|
| **NUMERIC[(P[,S])]** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。 |
| **DECIMAL[(P[,S])]** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。 |
| **SMALLMONEY** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。 |
| **MONEY** | **BYTES** | **org.apache.kafka.connect.data.Decimal** | **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。 |

### *4.4. DEPLOYING THE SQL SERVER CONNECTOR*

*SQL Server コネクターのインストールは、JAR をダウンロードして Kafka Connect 環境に抽出し、プラグインの親ディレクトリーが Kafka Connect 環境に指定されていることを確認する必要がある単純なプロセスです。*

*前提条件*

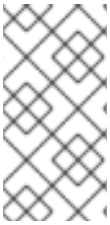
- *Zookeeper、Kafka、および Kafka Connect がインストールされている。*

- SQL Server をインストールし、設定している。

手順

1. Debezium SQL Server コネクター をダウンロードします。
2. ファイルを Kafka Connect 環境に展開します。
3. プラグインの親ディレクトリーを Kafka Connect プラグインパスに追加します。

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



注記

上記の例では、Debezium SQL Server コネクターを /kafka/connect/Debezium-connector-sqlserver パスに展開したことを前提としています。

4. Kafka Connect プロセスを再起動します。これにより、新しい JAR が確実に選択されるようになります。

関連情報

デプロイメントプロセス、および AMQ Streams でのコネクターのデプロイに関する詳細は、Debezium のインストールガイドを参照してください。

- Debezium の OpenShift へのインストール
- Debezium の RHEL へのインストール

#### 4.4.1. 設定例

コネクターを使用して、特定の SQL Server データベースまたはクラスターの変更イベントを生成するには、以下を行います。

1. SQL Server で CDC を有効にして、データベースに CDC イベントを公開します。

2. SQL Server コネクターの設定ファイルを作成します。

コネクターが起動すると、SQL Server データベースのスキーマの整合性スナップショットを取得し、変更のストリーミングを開始し、挿入、更新、および削除されたすべての行に対してイベントを生成します。スキーマおよびテーブルのサブセットに対してイベントを生成することもできます。必要に応じて、機密、大きすぎる列、または不要な列を無視、マスク、または切り捨てます。

以下は、192.168.99.100 のポート 1433 で SQL Server サーバーを監視するコネクターインスタンスの設定例で、これは論理的に fullfillment という名前になります。通常、コネクターに使用できる設定プロパティーを使用して、.yaml ファイルに Debezium SQL Server コネクターを設定します。

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

1. Kafka Connect サービスに登録する場合のコネクターの名前。

2. この SQL Server コネクタークラスの名前。

3. SQL Server インスタンスのアドレス。

4.

5

*SQL Server ユーザーの名前*

6

*SQL Server ユーザーのパスワード*

7

*変更をキャプチャーするデータベースの名前。*

8

*namespace を形成する SQL Server インスタンス/クラスターの論理名で、コネクターが書き込む Kafka トピックの名前、Kafka Connect スキーマ名、および Avro コネクターが使用される場合に対応する Avro スキーマの namespace のすべてに使用されます。*

9

*Debezium が変更をキャプチャーする必要があるすべてのテーブルのリスト。*

10

*DDL ステートメントをデータベース履歴トピックに書き込み、復元するためにコネクターによって使用される Kafka ブローカーのリスト。*

11

*コネクターが DDL ステートメントを書き、復元するデータベース履歴トピックの名前。このトピックは内部使用のみを目的としており、コンシューマーが使用しないようにしてください。*

*これら の設定で指定できる コネクタープロパティーの完全リスト を参照してください。*

*この設定は、POST 経由で稼働中の Kafka Connect サービスに送信できます。その後、設定を記録し、SQL Server データベースに接続するコネクタータスクを 1 つ起動します。トランザクションログを読み取り、イベントを Kafka トピックに記録します。*

#### *4.4.2. モニタリング*

*Kafka、Zookeeper、および Kafka Connect はすべて JMX メトリクスのサポートが組み込まれて*

*います。SQL Server コネクターは、JMX を介して監視できるコネクターのアクティビティーに関する多数のメトリクスを公開します。コネクターには 2 種類のメトリクスがあります。スナップショットメトリクスは、スナップショットアクティビティーの監視に役立ち、コネクターがスナップショットを実行している場合に利用できます。ストリーミングメトリクスは、コネクターが CDC テーブルデータを読み取る間、進捗とアクティビティーをモニターするのに役立ちます。*

#### *4.4.2.1. スナップショットメトリクス*

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

| 属性名 | タイプ | 説明 |
|---|---|---|
| **LastEvent** | **string** | コネクターが読み取りした最後のスナップショットイベント。 |
| **MilliSecondsSinceLastEvent** | **long** | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| **TotalNumberOfEventsSeen** | **long** | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| **NumberOfEventsFiltered** | **long** | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |
| **MonitoredTables** | **string[]** | コネクターによって監視されるテーブルの一覧。 |
| **QueueTotalCapcity** | **int** | snapshotter とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | snapshotter とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |
| **TotalTableCount** | **int** | スナップショットに含まれているテーブルの合計数。 |
| **RemainingTableCount** | **int** | スナップショットによってまだコピーされていないテーブルの数。 |
| **SnapshotRunning** | **boolean** | スナップショットが起動されたかどうか。 |
| **SnapshotAborted** | **boolean** | スナップショットが中断されたかどうか。 |
| **SnapshotCompleted** | **boolean** | スナップショットが完了したかどうか。 |

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **SnapshotDurationInSeconds** | **long** | スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。 |
| **RowsScanned** | **Map<String, Long>** | スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。 |

#### *4.4.2.2. ストリーミングメトリクス*

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

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **LastEvent** | **string** | コネクターが読み取られた最後のストリーミングイベント。 |
| **MilliSecondsSinceLastEvent** | **long** | コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。 |
| **TotalNumberOfEventsSeen** | **long** | 前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。 |
| **NumberOfEventsFiltered** | **long** | コネクターに設定されたホワイトリストまたはブラックリストフィルタールールでフィルターされたイベントの数。 |
| **MonitoredTables** | **string[]** | コネクターによって監視されるテーブルの一覧。 |
| **QueueTotalCapcity** | **int** | streamer とメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの長さ。 |
| **QueueRemainingCapcity** | **int** | ストリーマーとメインの Kafka Connect ループの間でイベントを渡すために使用されるキューの空き容量。 |
| **オンラインインストール** | **boolean** | コネクターが現在データベースサーバーに接続されているかどうかを示すフラグ。 |
| **MilliSecondsBehindSource** | **long** | 最後の変更イベントのタイムスタンプとそれを処理するコネクターとの間の期間 (ミリ秒単位)。値には、データベースサーバーとコネクターが実行されているマシンのクロックの差異が含まれます。 |
| **NumberOfCommittedTransactions** | **long** | コミットされた処理済みトランザクションの数。 |

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **SourceEventPosition** | **map< string, string >** | 最後に受信したイベントの位置。 |
| **LastTransactionId** | **string** | 最後に処理されたトランザクションのトランザクション識別子。 |

#### *4.4.2.3. スキーマ履歴メトリクス*

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

| 属性名 | タイプ | 説明 |
| --- | --- | --- |
| **Status** | **string** | データベース履歴の状態を記述する **STOPPED**、**RECOVERING**（ストレージから履歴を復元）、**RUNNING** のいずれか。 |
| **RecoveryStartTime** | **long** | リカバリーが開始された時点のエポック秒の時間。 |
| **ChangesRecovered** | **long** | リカバリーフェーズ中に読み取られた変更の数。 |
| **ChangesApplied** | **long** | リカバリーおよびランタイム中に適用されるスキーマ変更の合計数。 |
| **MilliSecondsSinceLastRecoveredChange** | **long** | 最後の変更が履歴ストアから復元された時点からの経過時間 (ミリ秒単位)。 |
| **MilliSecondsSinceLastAppliedChange** | **long** | 最後の変更が適用された時点からの経過時間 (ミリ秒単位)。 |
| **LastRecoveredChange** | **string** | 履歴ストアから復元された最後の変更の文字列表現。 |
| **LastAppliedChange** | **string** | 最後に適用された変更の文字列表現。 |

### *4.4.3. コネクタープロパティー*

*以下の設定プロパティーは、デフォルト値がない場合は必須です。*

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **name** | | コネクターの一意名。同じ名前で再登録を試みると失敗します。(このプロパティーはすべての Kafka Connect コネクターに必要です) |
| **connector.class** | | コネクターの Java クラスの名前。SQL Server コネクターには、常に **io.debezium.connector.sqlserver.SqlServerConnector** の値を使用してください。 |
| **tasks.max** | **1** | このコネクターのために作成する必要のあるタスクの最大数。SQL Server コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。 |
| **database.hostname** | | SQL Server データベースサーバーの IP アドレスまたはホスト名。 |
| **database.port** | **1433** | SQL Server データベースサーバーのポート番号 (整数)。 |
| **database.user** | | SQL Server データベースサーバーへの接続時に使用するユーザー名。 |
| **database.password** | | SQL Server データベースサーバーへの接続時に使用するパスワード。 |
| **database.dbname** | | 変更をストリーミングする SQL Server データベースの名前。 |
| **database.server.name** | | 監視対象の特定の SQL Server データベースサーバーの namespace を識別および提供する論理名。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターから生成されるすべての Kafka トピック名の接頭辞として使用されるためです。英数字とアンダースコアのみを使用する必要があります。 |
| **database.history.kafka.topic** | | コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。 |
| **database.history .kafka.bootstrap.servers** | | Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。これは、Kafka Connect プロセスによって使用される同じ Kafka クラスターを示す必要があります。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **table.whitelist** | | 監視するテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ホワイトリストに含まれていないテーブルはすべて監視から除外されます。各識別子の形式は schemaName.tableName です。デフォルトでは、コネクターは監視される各スキーマのシステム以外のテーブルをすべて監視します。**table.blacklist** と併用できません。 |
| **table.blacklist** | | 監視から除外されるテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト（任意）。ブラックリストに含まれていないテーブルはすべて監視されます。各識別子の形式は schemaName.tableName です。**table.whitelist** と併用できません。 |
| **column.blacklist** | **空の文字列** | 変更イベントメッセージの値から除外される必要がある列の完全修飾名と一致する正規表現のコンマ区切りリスト(任意)。列の完全修飾名の形式は schemaName.tableName.columnName です。プライマリーキー列は、値からブラックリストに指定された場合でも、イベントのキーに常に含まれることに注意してください。 |
| **time.precision.mode** | **adaptive** | 時間、日付、およびタイムスタンプは、異なる精度の種類で表すことができます。**adaptive** (デフォルト) は、データベース列の型を基にして、ミリ秒、マイクロ秒、またはナノ秒の精度値のいずれかを使用して、データベースの値と全く同じように時間とタイムスタンプをキャプチャーします。**connect** は、Kafka Connect の Time、Date、および Timestamp の組み込み表現を使用して、常に時間とタイムスタンプ値を表し、データベース列の精度に関わらず、ミリ秒の精度を使用します。時間値 を参照してください。 |
| **tombstones.on.delete** | **true** | 削除イベント後に廃棄 (tombstone) イベントを生成するかどうかを制御します。<br>**true** の場合、削除操作は 削除 イベントと後続の廃棄 (tombstone)イベントで表されます。**false** の場合、削除イベントのみが送信されます。<br>廃棄 (tombstone) イベントを生成すると (デフォルトの動作)、Kafka はソースレコードが削除されると、指定のキーに関連するすべてのイベントを完全に削除できます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **column.truncate.to.length.chars** | **該当なし** | フィールド値が指定された文字数より長い場合に、変更イベントメッセージ値で値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数である必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **column.mask.with.length.chars** | **該当なし** | 文字ベースの列の完全修飾名にマッチする正規表現のコンマ区切りリスト (オプション) で、変更イベントメッセージの値を、指定された数のアスタリスク (*) 文字で設定されるフィールド値に置き換える必要があります。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数またはゼロである必要があります。列の完全修飾名の形式は databaseName です。tableName.columnName または databaseName.schemaName.tableName.columnName. |
| **column.propagate.source.type** | **該当なし** | 出力された変更メッセージの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。列の完全修飾名の形式は schemaName.tableName.columnName です。 |
| **message.key.columns** | **空の文字列** | プライマリーキーをマップする完全修飾テーブルおよび列と一致する正規表現のセミコロン区切りリスト。<br>各項目（正規表現）は、カスタムキーを表す完全修飾 <**fully-qualified table**>:<**a comma-separated list of columns** > と一致する必要があります。<br>特定のコネクターに応じて、完全修飾テーブルは **DB_NAME.TABLE_NAME** または **SCHEMA_NAME.TABLE_NAME** として定義できます。 |

***以下の 高度な 設定プロパティーには、ほとんどの状況で機能する適切なデフォルト設定があるため、コネクターの設定で指定する必要はほとんどありません。***

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.mode** | Initial | キャプチャーされたテーブルの構造 (および必要に応じてデータ) の最初のスナップショットを作成するモード。スナップショットが完了すると、コネクターはデータベースのやり直し(redo)ログから変更イベントの読み取りを続行します。<br><br>サポートされる値は、<br>**initial** です。<br>initial: キャプチャーされたテーブルの構造およびデータのスナップショットを作成します。キャプチャーされたテーブルからデータの完全な表現をトピックに入力する必要がある場合に便利です。**schema_only**: キャプチャーされたテーブルの構造のスナップショットのみを作成します。今後発生する変更のみがトピックに伝達される場合に便利です。 |
| **snapshot.isolation.mode** | repeatable_read | 使用するトランザクション分離レベルと、監視されたテーブルをロックする期間を制御するモード。**可能** な値は、**read_uncommitted**、**read_committed**、**snapshot**、**exclusive** です（実際、**exclusive** モードは繰り返し可能な読み取り分離レベルを使用しますが、読み取るすべてのテーブルで排他的ロックを取ります）。<br><br>**snapshot**、**read_committed** モード、および **read_uncommitted** モードは、最初のスナップショットの実行中に他のトランザクションによるテーブル行の更新を防ぎませんが、**exclusive** および **repeatable_read** が実行されます。<br><br>もう 1 つの側面は、データの一貫性です。**exclusive** と **snapshot** モードのみが完全な整合性を保証します。つまり、最初のスナップショットとログのストリーミングが履歴の線形を保持します。**repeatable_read** および **read_committed** モードの場合は、たとえば、追加されたレコードが初回のスナップショットで 1 回、ストリーミングフェーズで 1 回 の計 2 回表示される可能性があります。しかし、この整合性レベルはデータのミラーリングであれば問題ないはずです。**read_uncommitted** の場合、データの整合性の保証はありません (一部のデータは損失または破損する可能性があります)。 |
| **event.processing.failure.handling.mode** | **fail** | イベントの処理中にコネクターが例外に対応する方法を指定します。**fail** は例外 (問題のあるイベントのオフセットを示す) を伝達するため、コネクターが停止します。<br>**warn** を指定すると問題のあるイベントがスキップされ、問題のあるイベントのオフセットがログに記録されます。<br>**skip** を指定すると、問題のあるイベントがスキップされます。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **poll.interval.ms** | **1000** | 各反復処理の実行中に新しい変更イベントが表示されるまでコネクターが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。 |
| **max.queue.size** | **8192** | データベースログから読み取られた変更イベントが Kafka に書き込まれる前に配置される、ブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに、CDC テーブルリーダーにバックプレシャーを提供できます。キューに発生するイベントは、このコネクターによって定期的に記録されるオフセットには含まれません。デフォルトは 8192 で、常に **max.batch.size** プロパティーに指定された最大バッチサイズよりも大きくする必要があります。 |
| **max.batch.size** | **2048** | このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。デフォルトは 2048 です。 |
| **heartbeat.interval.ms** | **0** | ハートビートメッセージが送信される頻度を制御します。このプロパティーには、コネクターがメッセージをハートビートトピックに送信する頻度を定義するミリ秒単位の間隔が含まれます。これは、コネクターがデータベースから変更イベントを受信しているかどうかを監視するために使用できます。また、長期に渡り変更されるのはキャプチャーされていないテーブルのレコードのみである場合は、ハートビートメッセージを利用する必要があります。このような場合、コネクターはデータベースからログの読み取りを続行しますが、変更メッセージを Kafka に出力しないため、オフセットの更新が Kafka にコミットされません。これにより、コネクターの再起動後に再送信される変更イベントが増える可能性があります。このプロパティーを **0** に設定して、ハートビートメッセージが全く送信されないようにします。<br>デフォルトでは無効にされています。 |
| **heartbeat.topics.prefix** | **__debezium-heartbeat** | ハートビートメッセージが送信されるトピックの命名を制御します。<br>トピックは、**<heartbeat.topics.prefix>.<server.name>** パターンに従って名前が付けられます。 |
| **snapshot.delay.ms** | | コネクターの起動後、スナップショットを取得するまで待機する間隔 (ミリ秒単位)。<br>クラスター内で複数のコネクターを開始する際に スナップショットが中断されないようにするために使用でき、コネクターのリバランスが実行される可能性があります。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **snapshot.fetch.size** | **2000** | スナップショットの実行中に各テーブルから 1 度に読み取る必要がある行の最大数を指定します。コネクターは、このサイズの複数のバッチでテーブルの内容を読み取ります。デフォルトは 2000 です。 |
| **snapshot.lock.timeout.ms** | **10000** | スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する整数値。この時間間隔でテーブルロックを取得できない場合、スナップショットは失敗します( スナップショットも参照してください)。<br>**0** に設定すると、コネクターがロックを取得できない場合、直ちに失敗します。値 **-1** は、無限に待つことを意味します。 |
| **snapshot.select.statement.overrides** | | テーブルのどの行がスナップショットに含まれるかを制御します。<br>このプロパティーには、完全修飾テーブル (SCHEMA_NAME.TABLE_NAME) のコンマ区切りリストが含まれます。各テーブルの select ステートメントは、ID **snapshot.select.statement.overrides.[SCHEMA_NAME].[TABLE_NAME]** によって識別される、テーブルごとに 1 つずつ追加の設定プロパティーに指定されます。これらのプロパティーの値は、スナップショットの実行中に特定のテーブルからデータを取得するときに使用する SELECT ステートメントです。**大規模な追加専用テーブルで可能なユースケースとしては、前のスナップショットが中断された場合にスナップショットの開始 (再開) 点を設定することが挙げられます。**<br>**注記**: この設定はスナップショットにのみ影響します。ログの読み取り中にキャプチャーされたイベントは影響を受けません。 |

| プロパティー | デフォルト | 説明 |
| --- | --- | --- |
| **sanitize.field.names** | コネクター設定が、Avro を使用するように **key.converter** または **value.converter** パラメーターを明示的に指定する場合は **true** です。それ以外の場合のデフォルトは **false** です。 | Avro の命名要件に準拠するためにフィールド名がサニタイズされるかどうか。 |
| **database.server.timezone** | | サーバーのタイムゾーン。<br><br>これは、サーバーから取得したトランザクションタイムスタンプ(ts_ms)のタイムゾーンを定義するために使用されます（実際にはゾーンではありません）。デフォルト値は unset です。SQL Server 2014 以前で実行され、Debezium コネクターを実行するデータベースサーバーおよび JVM に異なるタイムゾーンを使用する場合のみ指定する必要があります。<br>設定しない場合、デフォルトでは Debezium コネクターを実行する仮想マシンのタイムゾーンを使用します。この場合、SQL Server 2014 以前のバージョンで実行し、サーバーとコネクターが異なるタイムゾーンを使用する場合、正しくない ts_ms 値が生成されることがあります。<br>使用できる値には、Z、UTC、+02:00 などのオフセット値、CET などの短いゾーン ID、および Europe/Paris などの長いゾーン ID が含まれます。 |

***コネクターは、Kafka プロデューサーおよびコンシューマーの作成時に使用される パススルー 設定***

プロパティーもサポートします。具体的には、データベース履歴に書き込む Kafka プロデューサーを作成する際に database.history.producer. 接頭辞で始まるすべてのコネクター設定プロパティーが使用されます。また、接頭辞 database.history.consumer. で始まるすべてのプロパティーは、コネクターの起動時にデータベース履歴を読み取る Kafka コンシューマーを作成するときに使用されます。

たとえば、以下のコネクター設定プロパティーを使用すると、Kafka ブローカーへの接続をセキュア にすることができます。

Kafka プロデューサーおよびコンシューマーへのパススルー の他に、データベースで始まるプロパティー（例： database. applicationName=debezium ）は JDBC URL に渡されます。

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

Kafka プロデューサーおよびコンシューマーのすべての設定プロパティーについては、必ず Kafka ドキュメント を参照してください。(SQL Server コネクターは 新しいコンシューマー を使用します。)

# 第5章 DEBEZIUM の監視

Zookeeper および Kafka の提供する JMX メトリクスを使用して、Debezium を監視することができます。これらのメトリクスを使用するには、Zookeeper、Kafka、および Kafka Connect サービスの起動時にメトリクスを有効にする必要があります。JMX を有効にするには、正しい環境変数を設定する必要があります。

注記

同じマシン上で複数のサービスを実行している場合は、サービスごとに異なる JMX ポートを使用するようにしてください。

## 5.1. RHEL での DEBEZIUM の監視

### 5.1.1. Zookeeper JMX 環境変数

Zookeeper には JMX のサポートが組み込まれています。ローカルインストールを使用して Zookeeper を実行する場合、zkServer.sh スクリプトは以下の環境変数を認識します。

**JMXPORT**

JMX を有効にし、JMX に使用するポート番号を指定します。この値は、JVM パラメーター -Dcom.sun.management.jmxremote.port=$JMXPORT を指定するために使用されます。

**JMXAUTH**

接続時に JMX クライアントがパスワード認証を使用する必要があるかどうかを定義します。true または false のどちらかでなければなりません。デフォルトは false です。この値は、JVM パラメーター -Dcom.sun.management.jmxremote.authenticate=$JMXAUTH の指定に使用されます。

**JMXSSL**

JMX クライアントが SSL/TLS を使用して接続するかどうかを定義します。true または false のどちらかでなければなりません。デフォルトは false です。この値は、JVM パラメーター -Dcom.sun.management.jmxremote.ssl=$JMXSSL を指定するために使用されます。

**JMXLOG4J**

Log4J JMX MBean を無効にする必要があるかどうかを定義します。true (デフォルト) または false のいずれかである必要があります。デフォルトは true です。この値は、JVM パラメーター -Dzookeeper.jmx.log4j.disable=$JMXLOG4J の指定に使用されます。

### 5.1.2. Kafka JMX 環境変数

ローカルインストールを使用して Kafka を実行する場合、kafka-server-start.sh スクリプトは次の環境変数を認識します。

JMX_PORT

JMX を有効にし、JMX に使用するポート番号を指定します。この値は、JVM パラメーター -Dcom.sun.management.jmxremote.port=$JMX_PORT を指定するために使用されます。

KAFKA_JMX_OPTS

JMX オプション。起動時に直接 JVM に渡されます。デフォルトのオプションは次のとおりです。

- -Dcom.sun.management.jmxremote
- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

### 5.1.3. Kafka Connect JMX 環境変数

ローカルインストールを使用して Kafka を実行する場合、connect-distributed.sh スクリプトは次の環境変数を認識します。

JMX_PORT

JMX を有効にし、JMX に使用するポート番号を指定します。この値は、JVM パラメーター -Dcom.sun.management.jmxremote.port=$JMX_PORT を指定するために使用されます。

KAFKA_JMX_OPTS

JMX オプション。起動時に直接 JVM に渡されます。デフォルトのオプションは次のとおりです。

- -Dcom.sun.management.jmxremote
- -Dcom.sun.management.jmxremote.authenticate=false

- *-Dcom.sun.management.jmxremote.ssl=false*

### *5.2. OPENSHIFT 上での DEBEZIUM の監視*

*OpenShift 上で Debezium を使用している場合、JMX ポートを 9999 番で開放することで JMX メトリクスを取得することができます。詳細は、JMX オプション を参照してください。*

*また、Prometheus および Grafana を使用して JMX メトリクスを監視することができます。詳細は、メトリクスの導入 を参照してください。*

# 第6章 DEBEZIUM のログ機能

Debezium のコネクターには、さまざまなログ機能が組み込まれています。ログの設定を変更して、ログに表示するメッセージやログの送信先を制御することができます。(Kafka、Kafka Connect、および Zookeeper と同様に) Debezium は Java の Log4j ログフレームワークを使用します。

デフォルトでは、コネクターは起動時に大量の有用な情報を生成しますが、その後コネクターがソースのデータベースとシンクロすると、ほとんどログを生成しません。コネクターが正常に動作している場合はこれで十分ですが、コネクターが予期せぬ動作を示す場合には十分ではない可能性があります。そのような場合は、コネクターがしていること/していないことを記述したより詳細なログメッセージを生成するように、ログのレベルを変更することができます。

## 6.1. ロギングの概念

ログ機能を設定する前に、Log4J の ロガー、ログレベル、および アペンダー について理解しておく必要があります。

### ロガー

アプリケーションによって生成されるそれぞれのログメッセージは、特定の ロガー に送信されます (例: io.debezium.connector.mysql)。ロガーは階層構造を取ります。例えば、io.debezium.connector.mysql ロガーは io.debezium ロガーの子であるio.debezium.connector ロガーの子です。階層最上位の ルートロガー は、それより下位のすべてのロガーのデフォルトロガー設定を定義します。

### ログレベル

アプリケーションによって生成されるすべてのログメッセージには、特定の ログレベル もあります。

1. ERROR: エラー、例外、およびその他の重大な問題に設定される。
2. WARN: 潜在的な問題と課題
3. INFO: ステータスおよび一般的な動作に関する情報 (通常は少量) に設定される。
4. DEBUG: 予期しない挙動の診断に役立つより詳細な動作に関する情報に設定される。

5. *TRACE: 非常に冗長で詳細なアクティビティー (通常は非常に大量のデータを扱う)*

*アペンダー*

*アペンダー は、基本的にログメッセージが書き込まれる宛先です。それぞれのアペンダーは、そのログメッセージのフォーマットを制御します。これにより、ログメッセージの外観をより詳細に制御することができます。*

*ログ機能を設定するには、希望する各ロガーのレベルおよびそれらのログメッセージが書き込まれるアペンダーを指定します。ロガーは階層構造を取るため、ルートロガーの設定は、それより下位のすべてのロガーのデフォルトとして機能します。ただし、子の (または下位の) ロガーをオーバーライドすることができます。*

### *6.2. デフォルトのロギング設定について*

*Kafka Connect プロセスで Debezium コネクターを実行している場合、Kafka Connect は Kafka インストールの Log4j 設定ファイル（例： /opt/kafka/config/connect-log4j.properties）を使用します。デフォルトでは、このファイルには以下の設定が含まれています。*

*connect-log4j.properties*

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

log4j.appender.stdout=org.apache.log4j.ConsoleAppender (2)
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout (3)
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n (4)
...
```

(1) *デフォルトのロガー設定を定義するルートロガー。デフォルトでは、ロガーには INFO、WARN、および ERROR メッセージが含まれます。これらのログメッセージは stdout アペンダーに書き込まれます。*

(2) *stdout アペンダーは、ログメッセージを（ファイルではなく）コンソールに書き込みます。*

(3)

4

stdout アペンダーのパターン (詳しくは、Log4j ドキュメント を参照してください)。

他のロガーを設定しない限り、Debezium によって使用されるすべてのロガーは rootLogger 設定を継承します。

### 6.3. ロギングの設定

デフォルトでは、Debezium コネクターはすべての INFO、WARN、および ERROR メッセージをコンソールに書き込みます。ただし、以下の方法でこの設定を変更することができます。

- ロギングレベルの変更
- マッピングされた診断コンテキストを追加します。

注記

本セクションでは、Log4j で Debezium のロギングを設定するのに使用できる 2 つの方法のみを説明します。Log4j の使用に関する詳細は、アペンダーを設定して使用するチュートリアルを検索し、特定の宛先にログメッセージを送信します。

#### 6.3.1. ログレベルを変更する

デフォルトの Debezium ログレベルで、コネクターが正常かどうかを判断するのに十分な情報が得られます。ただし、コネクターが正常でない場合は、そのログレベルを変更して問題のトラブルシューティングを行うことができます。

一般に、Debezium コネクターは、ログメッセージを生成している Java クラスの完全修飾名と一致する名前のロガーにログメッセージを送信します。Debezium では、パッケージを使用して、類似または関連する機能のコードを取りまとめます。つまり、特定パッケージ内の特定クラスまたは全クラスのすべてのログメッセージを制御することができます。

手順

1. *log4j.properties ファイルを開きます。*

2. *コネクターのロガーを設定します。*

*以下の例では、MySQL コネクターのロガーおよびコネクターが使用するデータベース履歴の実装用ロガーを設定し、それらが DEBUG レベルのメッセージを記録するように設定します。*

*log4j.properties*

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

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

(1) *io.debezium.connector.mysql という名前のロガーを設定して、DEBUG、INFO、WARN、ERROR のメッセージを stdout のアペンダーに送信します。*

(2) *io.debezium.relational.history という名前のロガーを設定して、DEBUG、INFO、WARN、ERROR のメッセージを stdout のアペンダーに送信します。*

(3) (4) additivity *をオフにします。これは、ログメッセージが親ロガーのアペンダーに送信されないことを意味します（これにより、複数のアペンダーを使用する際にログメッセージが重複しているのを防ぐことができます）。*

3. *必要に応じて、コネクター内のクラスの特定サブセットのログレベルを変更します。*

コネクター全体のログレベルを上げるとログがより煩雑になり、状況を把握するのが困難になる場合があります。このような場合は、トラブルシューティングを行う問題に関連するクラスのサブセットのログレベルだけを変更することができます。

a. コネクターのログレベルを DEBUG または TRACE に設定します。

b. コネクターのログメッセージを確認します。

トラブルシューティングを行う問題に関連するログメッセージを探します。それぞれのログメッセージの末尾には、メッセージを生成した Java クラスの名前が表示されます。

c. コネクターのログレベルを INFO に戻します。

d. 識別したそれぞれの Java クラスのロガーを設定します。

たとえば、MySQL コネクターが binlog を処理する際にいくつかのイベントをスキップする理由が不明なシナリオを考えてみます。コネクター全体で DEBUG または TRACE ログを有効にするのではなく、コネクターのログレベルは INFO のままにして、binlog を読み取るクラスについてのみ DEBUG または TRACE を設定することができます。

log4j.properties

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

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

### 6.3.2. マッピングされた診断コンテキストを追加する

ほとんどの Debezium コネクター (および Kafka Connect ワーカー) は、複数のスレッドを使用してさまざまな動作を実行します。そのために、ログファイルを探し、特定の論理動作のログメッセージだけを識別するのが困難な場合があります。容易にログメッセージを探すことができるように、

*Debezium にはそれぞれのスレッドの追加情報を提供するさまざまなマッピングされた診断コンテキスト (MDC) が用意されています。*

*Debezium では、以下の MDC プロパティーを利用することができます。*

*dbz.connectorType*

*コネクタータイプの短縮エイリアス例えば、My Sql、Mongo、Postgres などです。同じタイプ のコネクターに関連付けられたすべてのスレッドは同じ値を使用するので、これを使用して、特定タイプのコネクターによって生成されたすべてのログメッセージを探すことができます。*

*dbz.connectorName*

*コネクターの設定で定義されているコネクターまたはデータベースサーバーの名前例えば、products、serverA などです。特定の コネクターインスタンス に関連付けられたすべてのスレッドは同じ値を使用するので、あるコネクターインスタンスによって生成されたすべてのログメッセージを探すことができます。*

*dbz.connectorContext*

*コネクターのタスク内で実行されている別のスレッドとして実行されている動作の短縮名例えば、main、binlog、snapshot などです。コネクターが特定のリソース (テーブルやコレクション等) にスレッドを割り当てる場合、そのリソースの名前が使用されることがあります。コネクターに関連付けられたスレッドごとに異なる値を使用するので、この特定の動作に関連付けられたすべてのログメッセージを探すことができます。*

*コネクターの MDC を有効にするには、log4j.properties ファイルでアペンダーを設定します。*

*手順*

1. *log4j.properties ファイルを開きます。*
2. *サポートされている Debezium MDC プロパティーのいずれかを使用するようにアペンダーを設定します。*

   *この例では、以下の MDC プロパティーを使用するように stdout アペンダーが設定されます。*

   *log4j.properties*

```
...
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} %-5p
%X{dbz.connectorType}|%X{dbz.connectorName}|%X{dbz.connectorContext}  %m
[%c]%n
...
```

これにより、以下のようなログメッセージが生成されます。

```
...
2017-02-07 20:49:37,692 INFO   MySQL|dbserver1|snapshot  Starting snapshot for
jdbc:mysql://mysql:3306/?
useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnic
ode=true&characterEncoding=UTF-8&characterSetResults=UTF-
8&zeroDateTimeBehavior=convertToNull with user 'debezium'
[io.debezium.connector.mysql.SnapshotReader]
2017-02-07 20:49:37,696 INFO   MySQL|dbserver1|snapshot  Snapshot is using user
'debezium' with these MySQL grants:
[io.debezium.connector.mysql.SnapshotReader]
2017-02-07 20:49:37,697 INFO   MySQL|dbserver1|snapshot   GRANT SELECT,
RELOAD, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.*
TO 'debezium'@'%'   [io.debezium.connector.mysql.SnapshotReader]
...
```

ログのそれぞれの行には、コネクターのタイプ (例: MySQL)、コネクターの名前 (例: dbserver1)、およびスレッドの動作 (例: snapshot) が含まれます。

## 6.4. OPENSHIFT での DEBEZIUM ログ

OpenShift で Debezium を使用している場合、Kafka Connect ロガーを使用して Debezium ロガーおよびログレベルを設定することができます。詳細は、Kafka Connect のロガー を参照してください。