



Red Hat Integration 2023.Q2

Debezium ユーザーガイド

Red Hat Integration 2.1.4 向け

Red Hat Integration 2023.Q2 Debezium ユーザーガイド

Red Hat Integration 2.1.4 向け

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

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

目次

はじめに	4
多様性を受け入れるオープンソースの強化	4
第1章 DEBEZIUM の概要	5
1.1. DEBEZIUM の機能	5
1.2. DEBEZIUM アーキテクチャーの説明	6
第2章 必要となるカスタムリソースのアップグレード	8
第3章 DB2 の DEBEZIUM コネクタ	9
3.1. DEBEZIUM DB2 コネクタの概要	9
3.2. DEBEZIUM DB2 コネクタの仕組み	10
3.3. DEBEZIUM DB2 コネクタのデータ変更イベントの説明	26
3.4. DEBEZIUM DB2 コネクタによるデータ型のマッピング方法	39
3.5. DEBEZIUM コネクタを実行するための DB2 の設定	43
3.6. DEBEZIUM DB2 コネクタのデプロイ	47
3.7. DEBEZIUM DB2 コネクタのパフォーマンスの監視	82
3.8. DEBEZIUM DB2 コネクタの管理	88
3.9. DEBEZIUM コネクタでのキャプチャーモードの DB2 テーブルのスキーマの更新	88
第4章 MONGODB の DEBEZIUM コネクタ	92
4.1. DEBEZIUM MONGODB コネクタの概要	92
4.2. DEBEZIUM MONGODB コネクタの仕組み	93
4.3. DEBEZIUM MONGODB コネクタのデータ変更イベントの説明	107
4.4. DEBEZIUM コネクタと連携する MONGODB の設定	119
4.5. DEBEZIUM MONGODB コネクタのデプロイメント	120
4.6. DEBEZIUM MONGODB コネクタのパフォーマンスの監視	148
4.7. DEBEZIUM MONGODB コネクタによる障害および問題の処理方法	153
第5章 MYSQL の DEBEZIUM コネクタ	157
5.1. DEBEZIUM MYSQL コネクタの仕組み	157
5.2. DEBEZIUM MYSQL コネクタのデータ変更イベントの説明	175
5.3. DEBEZIUM MYSQL コネクタによるデータ型のマッピング方法	190
5.4. DEBEZIUM コネクタを実行するための MYSQL の設定	197
5.5. DEBEZIUM MYSQL コネクタのデプロイメント	203
5.6. DEBEZIUM MYSQL コネクタのパフォーマンスの監視	244
5.7. DEBEZIUM MYSQL コネクタによる障害および問題の処理方法	251
第6章 ORACLE の DEBEZIUM コネクタ	253
6.1. DEBEZIUM ORACLE コネクタの仕組み	253
6.2. DEBEZIUM ORACLE コネクタのデータ変更イベントの説明	271
6.3. DEBEZIUM ORACLE コネクタによるデータ型のマッピング方法	282
6.4. DEBEZIUM と連携させるための ORACLE の設定	291
6.5. DEBEZIUM ORACLE コネクタのデプロイメント	297
6.6. DEBEZIUM ORACLE コネクタ設定プロパティの説明	312
6.7. DEBEZIUM ORACLE コネクタのパフォーマンスの監視	342
6.8. ORACLE コネクタのよくある質問	352
第7章 POSTGRESQL の DEBEZIUM コネクタ	356
7.1. DEBEZIUM POSTGRESQL コネクタの概要	356
7.2. DEBEZIUM POSTGRESQL コネクタの仕組み	357
7.3. DEBEZIUM POSTGRESQL コネクタのデータ変更イベントの説明	371
7.4. DEBEZIUM POSTGRESQL コネクタによるデータ型のマッピング方法	389

7.5. DEBEZIUM コネクタを実行するための POSTGRESQL の設定	401
7.6. DEBEZIUM POSTGRESQL コネクタのデプロイメント	405
7.7. DEBEZIUM POSTGRESQL コネクタのパフォーマンスの監視	448
7.8. DEBEZIUM POSTGRESQL コネクタによる障害および問題の処理方法	452
第8章 SQL SERVER の DEBEZIUM コネクタ	455
8.1. DEBEZIUM SQL SERVER コネクタの概要	455
8.2. DEBEZIUM SQL SERVER コネクタの仕組み	456
8.3. DEBEZIUM コネクタを実行するための SQL SERVER のセットアップ	491
8.4. DEBEZIUM SQL SERVER コネクタのデプロイ	495
8.5. スキーマ変更後のキャプチャーテーブルの更新	532
8.6. DEBEZIUM SQL SERVER コネクタのパフォーマンスの監視	536
第9章 DEBEZIUM の監視	542
9.1. DEBEZIUM コネクタを監視するためのメトリクス	542
9.2. ローカルインストールでの JMX の有効化	542
9.3. OPENSIFT 上での DEBEZIUM の監視	543
第10章 DEBEZIUM のログ機能	544
10.1. DEBEZIUM ログの概念	544
10.2. デフォルトの DEBEZIUM ログ設定	544
10.3. DEBEZIUM ログの設定	545
10.4. OPENSIFT での DEBEZIUM ログ	548
第11章 アプリケーション用 DEBEZIUM コネクタの設定	549
11.1. KAFKA CONNECT 自動トピック作成のカスタマイズ	549
11.2. AVRO シリアライゼーションを使用する DEBEZIUM コネクタの設定	555
11.3. CLOUDEVENTS フォーマットでの DEBEZIUM 変更イベントレコードの出力	561
11.4. DEBEZIUM コネクタへのシグナル送信	565
第12章 APACHE KAFKA で交換されたメッセージを修正するためのトランスフォームの適用	573
12.1. SMT 述語を使用した変換の選択的適用	573
12.2. 指定したトピックへの DEBEZIUM イベントレコードのルーティング	577
12.3. イベントの内容に応じた変更イベントレコードのトピックへのルーティング	582
12.4. DEBEZIUM 変更イベントレコードの絞り込み	587
12.5. DEBEZIUM の変更イベントからステート AFTER ソースレコードを抽出する	591
12.6. DEBEZIUM MONGODB 変更イベントから AFTER 状態のソースドキュメント抽出	599
12.7. 送信トレイパターンを使用する DEBEZIUM コネクタの設定	612
12.8. 送信トレイパターンを使用する DEBEZIUM MONGODB コネクタの設定	621
第13章 DEBEZIUM カスタムデータ型コンバータの開発	631
13.1. DEBEZIUM カスタムデータ型コンバータの作成	631
13.2. DEBEZIUM コネクタでのカスタムコンバータの使用	633

はじめに

Debezium は、データベースの行レベルの変更をキャプチャーする分散サービスのセットで、アプリケーションがそれらの変更を認識し、応答できるようにします。Debezium は、各データベーステーブルにコミットされたすべての行レベルの変更を記録します。各アプリケーションは、対象のトランザクションログを読み取り、発生した順序ですべての操作を確認します。

本ガイドでは、以下の Debezium トピックの使用方法について説明します。

- [1章 Debezium の概要](#)
- [2章 必要となるカスタムリソースのアップグレード](#)
- [3章 Db2 の Debezium コネクタ](#)
- [4章 MongoDB の Debezium コネクタ](#)
- [5章 MySQL の Debezium コネクタ](#)
- [6章 Oracle の Debezium コネクタ](#)
- [7章 PostgreSQL の Debezium コネクタ](#)
- [8章 SQL Server の Debezium コネクタ](#)
- [9章 Debezium の監視](#)
- [10章 Debezium のログ機能](#)
- [11章 アプリケーション用 Debezium コネクタの設定](#)
- [12章 Apache Kafka で交換されたメッセージを修正するためのトランスフォームの適用](#)
- [13章 Debezium カスタムデータ型コンバーターの開発](#)

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#) をご覧ください。

第1章 DEBEZIUM の概要

Debezium は、データベースの変更をキャプチャーする分散サービスのセットです。アプリケーションはこれらの変更を利用し、応答できます。Debezium は、各データベーステーブルの行レベルの変更を1つずつ変更イベントレコードにキャプチャーし、これらのレコードを Kafka トピックにストリーミングします。これらのストリームはアプリケーションによって読み取られ、変更イベントレコードは生成された順に提供されます。

詳細は、以下を参照してください。

- [「Debezium の機能」](#)
- [「Debezium アーキテクチャーの説明」](#)

1.1. DEBEZIUM の機能

Debezium は、Apache Kafka Connect のソースコネクターのセットです。各コネクターは、CDC (Change Data Capture) のデータベースの機能を使用して、異なるデータベースから変更を取り込みます。ログベースの CDC は、ポーリングや二重書き込みなどのその他の方法とは異なり、Debezium によって実装されます。

- **すべてのデータ変更がキャプチャーされたことを確認します。**
- 頻度の高いポーリングに必要な CPU 使用率の増加を防ぎながら、**非常に低遅延な変更イベント**を生成します。たとえば、MySQL または PostgreSQL の場合、遅延はミリ秒の範囲内になります。
- Last Updated (最終更新日時) の列など、**データモデルへの変更は必要ありません。**
- **削除をキャプチャー** できます。
- データベースの機能や設定に応じて、トランザクション ID や原因となるクエリーなどの古いレコードの**状態や追加のメタデータ**をキャプチャーできます。

詳細は、ブログの記事 [Five Advantages of Log-Based Change Data Capture](#) を参照してください。

Debezium コネクターは、さまざまな関連機能やオプションでデータの変更をキャプチャーします。

- **スナップショット**: コネクターが起動し、すべてのログが存在していない場合は、任意でデータベースの現在の状態の初期スナップショットを取得できます。通常、これは、データベースが一定期間稼働していて、トランザクションのリカバリーやレプリケーションに不要となったトランザクションログを破棄してしまった場合に該当します。スナップショットの実行モードは複数あります。これには、コネクターのランタイム時にトリガーされる可能性がある **増分** スナップショットのサポートが含まれます。詳細は、使用しているコネクターのドキュメントを参照してください。
- **フィルター**: キャプチャーされたスキーマ、テーブル、およびコラムは include または exclude リストフィルターで設定できます。
- **マスク**: たとえば、機密データが含まれている場合など、特定の列からの値はマスクできます。
- **監視**: ほとんどのコネクターは JMX を使用して監視できます。
- メッセージルーティング、フィルタリング、イベントフラット化などに、**SMT(すぐに使用できる単一のメッセージ変換)** などを使用できます。Debezium が提供する SMT の詳細は、[Applying transformations to modify messagesd with Apache Kafka](#) を参照してください。

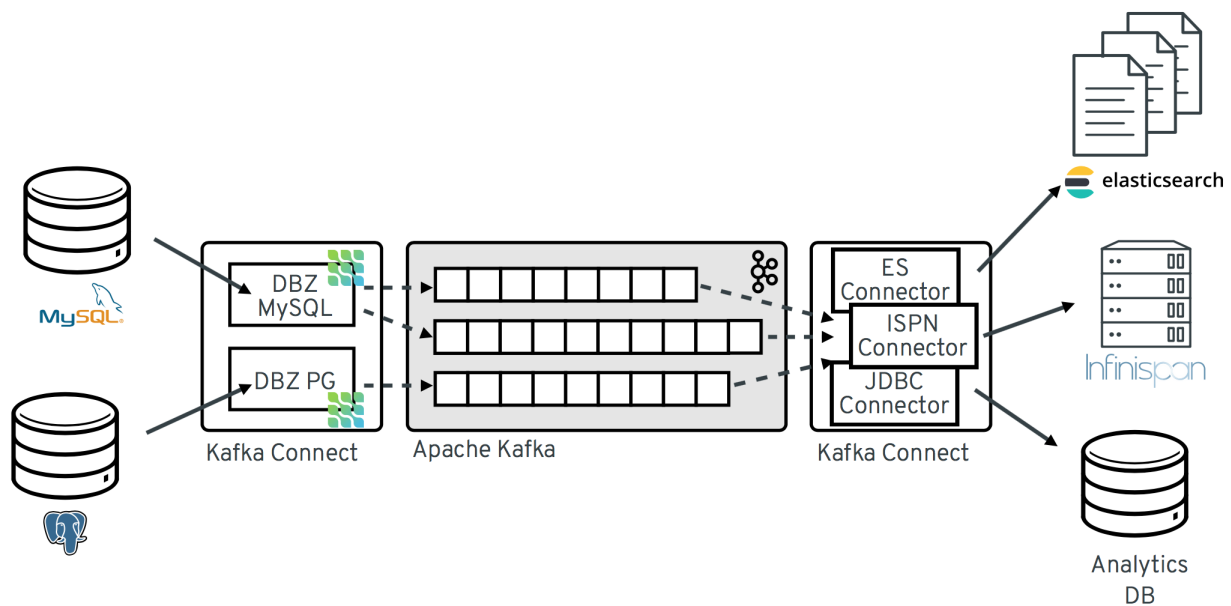
各コネクターのドキュメントには、コネクタ機能と設定オプションの詳細が記載されています。

1.2. DEBEZIUM アーキテクチャの説明

Apache [Kafka Connect](#) を使用して Debezium をデプロイします。Kafka Connect は、以下を実装および操作するためのフレームワークおよびランタイムです。

- レコードを Kafka に送信する Debezium などのソースコネクタ
- Kafka トピックから他のシステムにレコードを伝播するシンクコネクタ

以下の図は、Debezium をベースとした Change Data Capture パイプラインのアーキテクチャを示しています。



イメージにあるように、MySQL と PostgreSQL の Debezium コネクタは、この 2 種類のデータベースへの変更をキャプチャするためにデプロイされます。各 Debezium コネクタは、そのソースデータベースへの接続を確立します。

- MySQL コネクタは、**binlog** へのアクセスにクライアントライブラリーを使用します。
- PostgreSQL コネクタは論理レプリケーションストリームから読み取ります。

Kafka Connect は、Kafka ブローカー以外の別のサービスとして動作します。

デフォルトでは、1つのデータベースからの変更が、名前がテーブル名に対応する Kafka トピックに書き込まれます。必要な場合は、Debezium の [トピックルーティング変換](#) を設定すると、宛先トピック名を調整できます。たとえば、以下を実行できます。

- テーブルの名前と名前が異なるトピックへレコードをルーティングする。
- 複数テーブルの変更イベントレコードを単一のトピックにストリーミングする。

変更イベントレコードが Apache Kafka に存在する場合、Kafka Connect エコシステムの異なるコネクタは、Elasticsearch、データウェアハウス、分析システムなどのその他のシステムおよびデータベースや、Infinispan などのキャッシュにレコードをストリーミングできます。選択したシンクコネクタによっては、Debezium の [新しいレコード状態抽出](#) (Record State Extraction) の変換を設定する必要がある

ある場合があります。この Kafka Connect SMT は、Debezium の変更イベントからシンクコネクタに **after** 構造を伝播します。これは、デフォルトで伝播される詳細な変更イベントレコードの代わりになります。

第2章 必要となるカスタムリソースのアップグレード

Debezium は、AMQ Streams on OpenShift で実行する Apache Kafka クラスターにデプロイされた Kafka コネクタプラグインです。OpenShift CRD **v1** を準備するため、現行バージョンの AMQ Streams で、カスタムリソース定義 (CRD) API の必要なバージョンが **v1beta2** に設定されます。API の **v1beta2** バージョンは、以前にサポートされている **v1beta1** および **v1 alpha1** API バージョンを置き換えます。AMQ Streams では、**v1alpha1** および **v1beta1** API バージョンのサポートが非推奨になりました。以前のバージョンは、Debezium コネクタの設定に使用する KafkaConnect および KafkaConnector リソースを含む、AMQ Streams カスタムリソースから削除されています。

v1beta2 API バージョンをベースとする CRD は OpenAPI 構造スキーマを使用します。後続の **v1alpha1** または **v1beta1** API に基づくカスタムリソースは、構造的なスキーマをサポートしないため、現在のバージョンの AMQ Streams と互換性がありません。AMQ Streams 2.3 にアップグレードする前に、API バージョン **kafka.strimzi.io/v1beta2** を使用するように既存のカスタムリソースをアップグレードする必要があります。AMQ Streams 1.7 にアップグレードした後も、カスタムリソースをいつでもアップグレードできます。AMQ Streams 2.3 以降にアップグレードする前に、**v1beta2** API へのアップグレードを完了する必要があります。

CRD およびカスタムリソースのアップグレードを容易にするため、AMQ Streams は **v1beta2** と互換性のある形式に自動的にアップグレードする API 変換ツールを提供します。ツールおよび AMQ Streams のアップグレード方法は、[OpenShift での AMQ Streams のデプロイおよびアップグレード](#) を参照してください。



注記

カスタムリソースを更新する要件は、AMQ Streams on OpenShift で実行される Debezium デプロイメントにのみ適用されます。要件は、Red Hat Enterprise Linux の Debezium には適用されません。

第3章 DB2 の DEBEZIUM コネクタ

Debezium の Db2 コネクタは、Db2 データベースのテーブルで行レベルの変更をキャプチャーできます。このコネクタと互換性のある Db2 データベースのバージョンについては、[Debezium](#) でサポートされる設定ページを参照してください。

このコネクタは、テーブルをキャプチャーモードにする SQL ベースのポーリングモデルを使用する、SQL Server の Debezium 実装から大きく影響を受けます。テーブルがキャプチャーモードの場合、Debezium Db2 コネクタは、そのテーブルへの行レベルの更新ごとに変更イベントを生成し、ストリーミングします。

キャプチャーモードのテーブルには、関連する変更テーブルがあり、このテーブルは Db2 によって作成されます。キャプチャーモードのテーブルに対する変更ごとに、Db2 はその変更に関するデータをテーブルの関連する変更データテーブルに追加します。変更データテーブルには、行の各状態のエントリが含まれます。また、削除に関する特別なエントリもあります。Debezium Db2 コネクタは変更イベントを変更データテーブルから読み取り、イベントを Kafka トピックに出力します。

Debezium Db2 コネクタが Db2 データベースに初めて接続すると、コネクタが変更をキャプチャーするように設定されたテーブルの整合性スナップショットを読み取ります。デフォルトでは、システム以外のテーブルがすべて対象になります。キャプチャーモードにするテーブルまたはキャプチャーモードから除外するテーブルを指定できるコネクタ設定プロパティがあります。

スナップショットが完了すると、コネクタはコミットされた更新の変更イベントをキャプチャーモードのテーブルに出力し始めます。デフォルトでは、特定のテーブルの変更イベントは、テーブルと同じ名前を持つ Kafka トピックに移動します。アプリケーションとサービスはこれらのトピックから変更イベントを使用します。



注記

コネクタには、Linux 用の Db2 の標準部分として利用できる抽象構文表記 (ASN) ライブラリーを使用する必要があります。ASN ライブラリーを使用するには、IBM InfoSphere Data Replication (IIDR) のライセンスが必要です。ASN ライブラリーを使用するには、IIDR をインストールする必要はありません。

Debezium Db2 コネクタを使用するための情報および手順は、以下のように設定されています。

- [「Debezium Db2 コネクタの概要」](#)
- [「Debezium Db2 コネクタの仕組み」](#)
- [「Debezium Db2 コネクタのデータ変更イベントの説明」](#)
- [「Debezium Db2 コネクタによるデータ型のマッピング方法」](#)
- [「Debezium コネクタを実行するための Db2 の設定」](#)
- [「Debezium Db2 コネクタのデプロイ」](#)
- [「Debezium Db2 コネクタのパフォーマンスの監視」](#)
- [「Debezium Db2 コネクタの管理」](#)
- [「Debezium コネクタでのキャプチャーモードの Db2 テーブルのスキーマの更新」](#)

3.1. DEBEZIUM DB2 コネクタの概要

Debezium Db2 コネクタは、Db2 で SQL レプリケーションを有効にする [ASN Capture/Apply エージェント](#) をベースにしています。キャプチャーエージェントは以下を行います。

- キャプチャーモードであるテーブルの変更データテーブルを生成します。
- キャプチャーモードのテーブルを監視し、更新の変更イベントを対応する変更データテーブルのテーブルに格納します。

Debezium コネクタは SQL インターフェイスを使用して変更イベントの変更データテーブルに対してクエリを実行します。

データベース管理者は、変更をキャプチャーするテーブルをキャプチャーモードにする必要があります。便宜上およびテストを自動化するために、以下の管理タスクをコンパイルし、実行できる [Debezium 管理ユーザー定義機能 \(UDF\)](#) が C にあります。

- ASN エージェントの開始、停止、および再初期化。
- テーブルをキャプチャーモードにする。
- レプリケーション (ASN) スキーマと変更データテーブルの作成。
- キャプチャーモードからのテーブルの削除。

また、Db2 制御コマンドを使用してこれらのタスクを実行することもできます。

対象のテーブルがキャプチャーモードになった後、コネクタは対応する変更データテーブルを読み取り、テーブル更新の変更イベントを取得します。コネクタは、変更されたテーブルと同じ名前を持つ Kafka トピックに対して、行レベルの挿入、更新、および削除操作ごとに変更イベントを出力します。これは、変更可能なデフォルトの動作です。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、行レベルの各変更イベントに対応できます。

通常、データベース管理者はテーブルのライフサイクルの途中でテーブルをキャプチャーモードにします。つまり、コネクタにはテーブル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、Db2 コネクタが最初に特定の Db2 データベースに接続すると、キャプチャーモードである各テーブルで **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時点から変更イベントをストリーミングします。これにより、コネクタはキャプチャーモードのテーブルの整合性のあるビューで開始し、スナップショットの実行中に加えられた変更は破棄されません。

Debezium コネクタはフォールトトレラントです。コネクタは変更イベントを読み取りおよび生成すると、変更データテーブルエントリのログシーケンス番号 (LSN) を記録します。LSN はデータベースログの変更イベントの位置になります。コネクタ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)、コネクタは再起動後に最後に停止した場所の変更データ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。つまり、コネクタの停止時にスナップショットが完了しなかった場合、コネクタの再起動時に新しいスナップショットが開始されます。

3.2. DEBEZIUM DB2 コネクタの仕組み

Debezium Db2 コネクタを最適に設定および実行するには、コネクタによるスナップショットの実行方法、変更イベントのストリーム方法、Kafka トピック名の決定方法、およびスキーマ変更の処理方法を理解すると便利です。

詳細は以下を参照してください。

- [「Debezium Db2 コネクタによるデータベーススナップショットの実行方法」](#)

- 「Debezium Db2 コネクタによる変更データテーブルの読み取り方法」
- 「Debezium Db2 変更イベントレコードを受信する Kafka トピックのデフォルト名」
- 「Debezium Db2 コネクタのスキーマ変更トピック」
- 「トランザクション境界を表す Debezium Db2 コネクタによって生成されたイベント」

3.2.1. Debezium Db2 コネクタによるデータベーススナップショットの実行方法

Db2 のレプリケーション機能は、データベース変更の完全な履歴を保存するようには設計されていません。そのため、Debezium Db2 コネクタが初めてデータベースに接続すると、キャプチャーモードのテーブルの整合性スナップショットを作成し、この状態を Kafka にストリーミングします。これにより、テーブルの内容のベースラインが確立されます。

デフォルトでは、Db2 コネクタがスナップショットを実行すると、以下が実行されます。

1. キャプチャーモードになっているテーブルを判断するため、スナップショットに含まれないテーブルも判断されます。デフォルトでは、システム以外のテーブルはすべてキャプチャーモードになっています。**table.exclude.list** や **table.include.list** などのコネクタ設定プロパティを使用すると、キャプチャーモードである必要があるテーブルを指定できません。
2. キャプチャーモードの各テーブルでロックを取得します。これにより、スナップショットの実行中にこれらのテーブルでスキーマの変更が発生しないようにします。ロックのレベルは、**snapshot.isolation.mode** コネクタ設定プロパティ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で、最大 (最新) の LSN の位置を読み取ります。
4. キャプチャーモードになっているすべてのテーブルのスキーマをキャプチャーします。コネクタは、内部データベーススキーマ履歴トピックでこの情報を永続化します。
5. 必要に応じて、ステップ 2 で取得したロックを解放します。通常、これらのロックは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、コネクタはキャプチャーモードテーブルとそれらのスキーマをスキャンします。スキャン中、コネクタは以下を行います。
 - a. スナップショットの開始前に、テーブルが作成されたことを確認します。そうでない場合は、スナップショットはそのテーブルをスキップします。スナップショットが完了し、コネクタが変更イベントの出力を開始した後、コネクタはスナップショットの実行中に作成されたテーブルの変更イベントを生成します。
 - b. キャプチャーモードになっている各テーブルで、各行の **読み取り** イベントを生成します。すべての **読み取り** イベントには、LSN の位置が含まれ、これはステップ 3 で取得した LSN の位置と同じです。
 - c. テーブルと同じ名前を持つ Kafka トピックに各 **読み取り** イベントを出力します。
7. コネクタ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。

3.2.1.1. アドホックスナップショット

デフォルトでは、コネクタは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクタではスナップショットプロセスは繰り返し処理されません。コネクタがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクタを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。テーブルデータを再キャプチャーするメカニズムを提供するため、Debezium にはアドホックスナップショットを実行するオプションがあります。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクタ設定は、異なるテーブルセットをキャプチャーするように変更されます。
- Kafka トピックを削除して、再構築する必要があります。
- 設定エラーや他の問題が原因で、データの破損が発生します。

アドホックと呼ばれるスナップショットを開始することで、以前にスナップショットをキャプチャーしたテーブルのスナップショットを再実行できます。アドホックスナップショットには、[シグナルテーブル](#)を使用する必要があります。シグナルリクエストを Debezium シグナルテーブルに送信して、アドホックスナップショットを開始します。

既存のテーブルのアドホックスナップショットを開始すると、コネクタはテーブルにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、[トピックの自動作成](#)が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックのスナップショットシグナルは、スナップショットに追加するテーブルを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のテーブルのサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のテーブルの内容のサブセットをキャプチャできます。

execute-snapshot メッセージをシグナルテーブルに送信してキャプチャーするテーブルを指定します。以下の表で説明されているように、**run-snapshot** シグナルのタイプを **incremental** に設定し、スナップショットに追加するテーブルの名前を指定します。

表3.1 アドホックの **execute-snapshot** シグナルレコードの例

フィールド	デフォルト	値
type	incremental	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 incremental スナップショットのみです。
data-collections	該当なし	スナップショットされるテーブルの完全修飾名にマッチする正規表現を含む配列。名前の形式は signal.data.collection 設定オプションと同じです。
additional-condition	該当なし	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

execute-snapshot シグナルタイプのエントリーをシグナルテーブルに追加して、アドホックスナップショットを開始します。コネクタがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各テーブルの開始ポイントおよびエンドポイントとして使用します。テーブルのエントリー数と設定されたチャンクサイズに基づいて、Debezium はテーブルをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、**execute-snapshot** アクションタイプは **増分スナップショット** のみをトリガーします。詳細は、**スナップショットの増分**を参照してください。

3.2.1.2. 増分スナップショット

スナップショットを柔軟に管理するため、Debezium には **増分スナップショット** と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、**Debezium コネクターにシグナルを送信するための** Debezium メカニズムに依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各テーブルを段階的にキャプチャーします。スナップショットがキャプチャーするテーブルと、**各チャンクのサイズ**を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

増分スナップショットが進むと、Debezium はウォーターマークを使用して進捗を追跡し、キャプチャーする各テーブル行のレコードを管理します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクターはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセスが再開すると、スナップショットは最初からテーブルをキャプチャーするのではなく、停止した時点から開始します。
- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。たとえば、コネクター設定を変更してテーブルを **table.include.list** プロパティに追加した後スナップショットを再実行します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各テーブルをプライマリーキー別に分類して、**設定されたチャンクサイズ**に基づいてテーブルをチャンクに分割します。チャンクごとに作業し、テーブルの行ごとにチャンクでキャプチャーします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットの作成が進むにつれ、他のプロセスがデータベースへのアクセスを継続し、テーブルレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT、UPDATE、DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できます。つまり、ストリーミングプロセスは、スナップショットがその行の **READ** イベントが含まれるチャンクをキャプチャーする前に、テーブルの行を変更するイベントを生成する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリーミングされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて入ってきた **READ** イベントと、同じテーブルの行を変更するストリーミングイベント間の競合の解決を容易にするために、Debezium は **スナップショットウィンドウ** と呼ばれるものを使用します。スナップショットウィンドウは、増分スナップショットが指定のテーブルチャンクのデータをキャプチャしている途中に、間隔を決定します。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は 2 種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャするスナップショットレコードは、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをテーブルの Kafka トピックに出力します。

コネクタは各スナップショットチャンクにプロセスを繰り返します。



警告

Db2 の Debezium コネクタでは、増分スナップショットの実行中のスキーマの変更はサポートしません。

3.2.1.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、[アドホックスナップショットシグナル](#) をソースデータベースのシグナルテーブルに送信することです。

シグナルを SQL **INSERT** クエリーとしてシグナルテーブルに送信します。

Debezium がシグナルテーブルの変更を検出すると、シグナルを読み取り、要求されたスナップショット操作を実行します。

送信するクエリーはスナップショットに追加するテーブルを指定し、必要に応じてスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに追加するテーブルを指定するには、テーブルをリストする **data-collections** 配列またはテーブルの照合に使用する正規表現の配列を指定します。以下に例を示します。

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

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるテーブルの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのテーブルを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **My.Table** のテーブルを含めるには、**"public"."My.Table"** の形式を使用します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. SQL クエリーを送信し、アドホック増分スナップショット要求をシグナルテーブルに追加します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
'execute-snapshot', 3
'{"data-collections": ["schema1.table1", "schema2.table2"], 4
"type": "incremental"}, 5
"additional-condition": "color=blue"}'); 6
```

コマンドの **id**、**type**、および **data** パラメーターの値は、[シグナルテーブルのフィールド](#) に対応します。

以下の表では、この例のパラメーターについて説明しています。

表3.2 シグナルテーブルに増分スナップショットシグナルを送信する SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。

項目	値	説明
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリーへのログメッセージを特定します。Debezium はこの文字列を使用しません。代わりに、スナップショット作成中に、Debezium は独自の ID 文字列をウォーターマークシグナルとして生成します。
3	execute-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるテーブル名の配列またはテーブル名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。
5	incremental	実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 値を指定しない場合には、コネクターは増分スナップショットを実行します。
6	additional-condition	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。 additional-condition パラメーターの詳細は、 additional-condition 付きのアドホック増分スナップショット を参照してください。

additional-condition 付きのアドホック増分スナップショット

スナップショットに、テーブル内のコンテンツのサブセットのみを含める場合は、スナップショットシグナルシグナルに **additional-condition** パラメーターを追加してシグナル要求を変更できます。

一般的なスナップショットの SQL クエリーは、以下の形式を取ります。

```
SELECT * FROM <tableName> ....
```

additional-condition パラメーターを追加して、以下の例のように **WHERE** 条件を SQL クエリーに追加します。

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```

以下の例は、シグナルテーブルに追加の条件を含むアドホック増分スナップショット要求を送信する SQL クエリーを示しています。

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

たとえば、以下の列が含まれる **products** テーブルがあるとします。

- **id** (プライマリーキー)
- **color**
- **quantity**

products テーブルの増分スナップショットに **color=blue** のデータ項目のみを含める場合は、次の SQL ステートメントを使用してスナップショットをトリガーできます。

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

additional-condition パラメーターを使用すると、列が2つ以上となる条件を指定することもできます。たとえば、前述の例の **products** テーブルを使用して、**color=blue** および **quantity>10** だけに一致するアイテムのみのデータが含まれる増分スナップショットをトリガーするクエリーを送信できます。

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-snapshot',
{'data-collections': ["schema1.products"],"type":"incremental", "additional-condition":"color=blue AND
quantity>10"});
```

以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 シグナルテーブルに送信する SQL クエリーでの type 値の指定は任意です。 値を指定しない場合には、コネクターは増分スナップショットを実行します。

項目	フィールド名	説明
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

3.2.1.4. 増分スナップショットの停止

ソースデータベースのテーブルにシグナルを送信して、増分スナップショットを停止することもできます。SQL **INSERT** クエリーを送信して、停止スナップショットシグナルをテーブルに送信します。

Debezium はシグナルテーブルの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのテーブルを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

- SQL クエリーを送信して、シグナリングテーブルへのアドホックインクリメンタル スナップショットを停止します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
'stop-snapshot', 3
'{"data-collections": ["schema1.table1", "schema2.table2"], 4
"type": "incremental"}'); 5
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表3.3 シグナリングテーブルに増分スナップショット停止信号を送信するための SQL コマンドのフィールドの説明

項目	値	説明
----	---	----

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリへのログメッセージを特定します。Debezium はこの文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドのオプションコンポーネントで、スナップショットから削除するテーブル名の配列またはテーブル名とマッチする正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクタのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

3.2.2. Debezium Db2 コネクタによる変更データテーブルの読み取り方法

スナップショットの完了後、Debezium Db2 コネクタが初めて起動すると、キャプチャーモードである各ソーステーブルの変更データテーブルを識別します。コネクタは各変更データテーブルに対して以下を行います。

1. 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イベントを読み取りません。
2. 各イベントのコミット LSN および変更 LSN に従って、変更イベントを順序付けます。これにより、コネクタはテーブルが変更された順序で変更イベントを出力します。
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. コネクタが Kafka Connect に渡した最大 LSN を保存します。

再起動後、コネクタは停止した場所でオフセット (コミット LSN および変更 LSN) から変更イベントの出力を再開します。コネクタが稼働し、変更イベントを出力している間、キャプチャーモードからテーブルを削除したり、テーブルをキャプチャーモードに追加したりすると、コネクタは変更を検出して、それに合わせて動作を変更します。

3.2.3. Debezium Db2 変更イベントレコードを受信する Kafka トピックのデフォルト名

デフォルトでは、Db2 コネクタは、テーブルで発生するすべての **INSERT**、**UPDATE**、**DELETE** 操作の変更イベントを、そのテーブルに固有の単一の Apache Kafka トピックに書き込みます。コネクタは以下の規則を使用して変更イベントトピックに名前を付けます。

`topicPrefix.schemaName.tableName`

以下のリストは、デフォルト名のコンポーネントの定義を示しています。

topicPrefix

topic.prefix コネクタ設定プロパティで指定されたトピック接頭辞。

schemaName

操作が発生したスキーマの名前。

tableName

操作が発生したテーブルの名前。

たとえば、**MYSCHEMA** スキーマに 4 つのテーブル (**PRODUCTS**、**PRODUCTS_ON_HAND**、**CUSTOMERS**、**ORDERS**) を含む **mydatabase** データベースを使用した Db2 インストールについて考えてみます。コネクタはイベントを以下の 4 つの Kafka トピックに出力します。

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

コネクタは同様の命名規則を適用して、内部データベーススキーマの履歴トピック (**スキーマ変更トピック** と **トランザクションメタデータトピック**) にラベルを付けます。

デフォルトのトピック名が要件を満たさない場合は、カスタムトピック名を設定できます。カスタムトピック名を設定するには、論理トピックルーティング SMT に正規表現を指定します。論理トピックルーティング SMT を使用してトピックの命名をカスタマイズする方法は、**トピックルーティング** を参照してください。

3.2.4. Debezium Db2 コネクタのスキーマ変更トピック

Debezium Db2 コネクタを設定すると、データベースのキャプチャーされたテーブルに適用されるスキーマの変更を記述するスキーマ変更イベントを生成できます。

Debezium は、以下の場合にスキーマ変更トピックにメッセージを出力します。

- 新しいテーブルがキャプチャーモードになる。
- テーブルがキャプチャーモードから削除される。
- **データベーススキーマの更新** 中に、キャプチャーモードであるテーブルのスキーマが変更される。

コネクタはスキーマ変更イベントを、**<topicPrefix>** という名前の Kafka スキーマ変更トピックに書き込みます。ここで **<topicPrefix>** は、**topic.prefix** コネクタ設定プロパティで指定されたトピック接頭辞です。コネクタがスキーマ変更トピックに送信するメッセージには以下の要素などのペイロードが含まれます。

databaseName

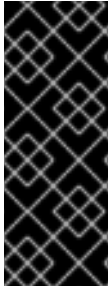
ステートメントが適用されるデータベースの名前。**databaseName** の値は、メッセージキーとして機能します。

pos

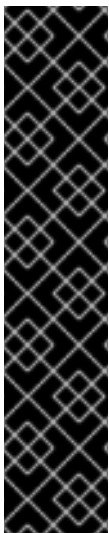
ステートメントが表示される binlog の位置。

tableChanges

スキーマの変更後のテーブルスキーマ全体の構造化表現。**tableChanges** フィールドには、テーブルの各列のエントリなどのアレイが含まれます。構造化された表現は JSON または Avro 形式でデータを表示するため、コンシューマーは DDL パーサーを介して最初にメッセージを処理しなくてもメッセージを簡単に読み取りできます。

**重要**

キャプチャーモードであるテーブルでは、コネクターはスキーマ変更トピックにスキーマ変更の履歴だけでなく、内部データベーススキーマ履歴トピックにも格納します。内部データベーススキーマ履歴トピックはコネクターのみの使用を対象としており、使用するアプリケーションによる直接使用を目的としていません。スキーマ変更に関する通知が必要なアプリケーションが、スキーマ変更トピックからの情報のみを使用するようにしてください。

**重要**

データベーススキーマ履歴トピックをパーティションに分割しないでください。データベーススキーマ履歴トピックが正しく機能するには、コネクターが出力するイベントレコードの一貫したグローバル順序を維持する必要があります。

トピックがパーティション間で分割されないようにするには、以下のいずれかの方法を使用してトピックのパーティション数を設定します。

- データベーススキーマ履歴トピックを手動で作成する場合は、パーティション数を **1** に指定します。
- Apache Kafka ブローカーを使用してデータベーススキーマ履歴トピックを自動的に作成する場合に、トピックが作成されるので、**Kafka num.partitions** 設定オプションの値を **1** に設定します。

**警告**

コネクターがスキーマ変更トピックに出力するメッセージの形式は、初期の状態であり、通知なしに変更される可能性があります。

例: Db2 コネクターのスキーマ変更トピックに出力されるメッセージ

以下の例は、スキーマ変更トピックのメッセージを示しています。メッセージには、テーブルスキーマの論理表現が含まれます。

```
{
  "schema": {
    ...
```

```
},
"payload": {
  "source": {
    "version": "2.1.4.Final",
    "connector": "db2",
    "name": "db2",
    "ts_ms": 0,
    "snapshot": "true",
    "db": "testdb",
    "schema": "DB2INST1",
    "table": "CUSTOMERS",
    "change_isn": null,
    "commit_isn": "00000025:00000d98:00a2",
    "event_serial_no": null
  },
  "ts_ms": 1588252618953, ①
  "databaseName": "TESTDB", ②
  "schemaName": "DB2INST1",
  "ddl": null, ③
  "tableChanges": [ ④
    {
      "type": "CREATE", ⑤
      "id": "\"DB2INST1\".\"CUSTOMERS\"", ⑥
      "table": { ⑦
        "defaultCharsetName": null,
        "primaryKeyColumnNames": [ ⑧
          "ID"
        ],
        "columns": [ ⑨
          {
            "name": "ID",
            "jdbcType": 4,
            "nativeType": null,
            "typeName": "int identity",
            "typeExpression": "int identity",
            "charsetName": null,
            "length": 10,
            "scale": 0,
            "position": 1,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "FIRST_NAME",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "varchar",
            "typeExpression": "varchar",
            "charsetName": null,
            "length": 255,
            "scale": null,
            "position": 2,
            "optional": false,
            "autoIncremented": false,
```

```
    "generated": false
  },
  {
    "name": "LAST_NAME",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "varchar",
    "typeExpression": "varchar",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 3,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "EMAIL",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "varchar",
    "typeExpression": "varchar",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 4,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  }
],
"attributes": [ 10
  {
    "customAttribute": "attributeValue"
  }
]
}
]
```

表3.4 スキーマ変更トピックに出力されたメッセージのフィールドの説明

項目	フィールド名	説明
1	ts_ms	<p>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。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。</p> <p>ソースオブジェクトの <code>ts_ms</code> は、データベースで変更が行われた時刻を示す。 <code>payload.source.ts_ms</code> の値を <code>payload.ts_ms</code> の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

項目	フィールド名	説明
2	databaseName schemaName	変更が含まれるデータベースとスキーマを識別します。
3	ddl	Db2 コネクタの場合は常に null です。その他のコネクタでは、このフィールドにスキーマの変更を行う DDL が含まれます。この DDL は Db2 コネクタでは使用できません。
4	tableChanges	DDL コマンドによって生成されるスキーマの変更が含まれる 1 つ以上の項目の配列。
5	type	変更の種類を説明します。値は以下のいずれかになります。 <ul style="list-style-type: none"> ● CREATE - テーブルの作成 ● ALTER - テーブルの変更 ● DROP - テーブルの削除
6	id	作成、変更、または破棄されたテーブルの完全な識別子。
7	table	適用された変更後のテーブルメタデータを表します。
8	primaryKeyColumnNames	テーブルのプライマリーキーを設定する列のリスト。
9	列	変更されたテーブルの各列のメタデータ。
10	attributes	各テーブル変更のカスタム属性メタデータ。

コネクタがスキーマ変更トピックに送信するメッセージでは、メッセージキーはスキーマの変更が含まれるデータベースの名前です。以下の例では、**payload** フィールドにキーが含まれます。

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

```

"databaseName": "TESTDB"
}
}

```

3.2.5. トランザクション境界を表す Debezium Db2 コネクターによって生成されたイベント

Debezium は、トランザクション境界を表し、変更データイベントメッセージをエンリッチするイベントを生成できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクターのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクターをデプロイする前に発生するトランザクションのメタデータは利用できません。

Debezium は、すべてのトランザクションで **BEGIN** および **END** 区切り文字のトランザクション境界イベントを生成します。トランザクション境界イベントには以下のフィールドが含まれます。

status

BEGIN または **END**

id

一意のトランザクション識別子の文字列表現。

ts_ms

データソースでのトランザクション境界イベント (**BEGIN** または **END** イベント) の時間。もしデータソースが Debezium にイベント時間を提供しないなら、このフィールドは代わりに Debezium がイベントを処理する時間を表します。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

data_collection と **event_count** 要素のペアの配列。これは、コネクターがデータコレクションから発信された変更に対して出力するイベントの数を示します。

例

```

{
  "status": "BEGIN",
  "id": "00000025:00000d08:0025",
  "ts_ms": 1486500577125,
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "00000025:00000d08:0025",
  "ts_ms": 1486500577691,
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "testDB.dbo.tablea",

```

```

    "event_count": 1
  },
  {
    "data_collection": "testDB.dbo.tableb",
    "event_count": 1
  }
]
}

```

topic.transaction オプションで上書きされない限り、コネクタはトランザクションイベントを **<topic.prefix>.transaction** トピックに出力します。

データ変更イベントのエンリッチメント

トランザクションメタデータを有効にすると、コネクタは変更イベント **Envelope** を新しい **transaction** フィールドでエンリッチします。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下は、メッセージの例になります。

```

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

```

3.3. DEBEZIUM DB2 コネクタのデータ変更イベントの説明

Debezium Db2 コネクタは、行レベルの **INSERT**、**UPDATE**、および **DELETE** 操作ごとにデータ変更イベントを生成します。各イベントにはキーと値が含まれます。キーと値の構造は、変更されたテーブルによって異なります。

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** を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があり、コンシュー

マーによる処理が困難になることがあります。これに対応するために、各イベントにはコンテンツのスキーマが含まれます。スキーマレジストリーを使用している場合は、コンシューマーがレジストリーからスキーマを取得するために使用できるスキーマ ID が含まれます。これにより、各イベントが自己完結型になります。

以下のスケルトン JSON は、変更イベントの基本となる 4 つの部分を示しています。ただし、アプリケーションで使用するために選択した Kafka Connect コンバーターの設定方法によって、変更イベントのこれら 4 部分の表現が決定されます。**schema** フィールドは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。同様に、イベントキーおよびイベントペイロードは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。JSON コンバーターを使用し、変更イベントの基本となる 4 つの部分すべてを生成するように設定すると、変更イベントの構造は次のようになります。

```
{
  "schema": { ❶
    ...
  },
  "payload": { ❷
    ...
  },
  "schema": { ❸
    ...
  },
  "payload": { ❹
    ...
  },
}
```

表3.5 変更イベントの基本内容の概要

項目	フィールド名	説明
1	schema	最初の schema フィールドはイベントキーの一部です。イベントキーの payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、最初の schema フィールドは、変更されたテーブルのプライマリーキーの構造、またはテーブルにプライマリーキーがない場合は変更されたテーブルの一意キーの構造を記述します。 message.key.columns コネクター設定プロパティを設定すると、テーブルのプライマリーキーをオーバーライドできます。この場合、最初の schema フィールドはそのプロパティによって識別されるキーの構造を記述します。
2	payload	最初の payload フィールドはイベントキーの一部です。前述の schema フィールドによって記述された構造を持ち、変更された行のキーが含まれます。
3	schema	2 つ目の schema フィールドはイベント値の一部です。イベント値の payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、2 つ目の schema は変更された行の構造を記述します。通常、このスキーマには入れ子になったスキーマが含まれます。

項目	フィールド名	説明
4	payload	2つ目の payload フィールドはイベント値の一部です。前述の schema フィールドによって記述された構造を持ち、変更された行の実際のデータが含まれます。

デフォルトでは、コネクタによって、変更イベントレコードがイベントの元のテーブルと同じ名前を持つトピックにストリーミングされます。詳細は、[トピック名](#) を参照してください。



警告

Debezium Db2 コネクタは、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。つまり、論理サーバー名はアルファベットまたはアンダースコア (a-z、A-Z、または `_`) で始まる必要があります。論理サーバー名の残りの各文字と、データベース名とテーブル名の各文字は、アルファベット、数字、またはアンダースコア (a-z、A-Z、0-9、または `_`) でなければなりません。無効な文字がある場合は、アンダースコアに置き換えられます。

論理サーバー名、データベース名、またはテーブル名に無効な文字が含まれ、名前を区別する唯一の文字が無効であると、無効な文字はすべてアンダースコアに置き換えられるため、予期せぬ競合が発生する可能性があります。

また、データベース、スキーマ、およびテーブルの Db2 名では、大文字と小文字を区別することができます。つまり、コネクタは同じ Kafka トピックに複数のテーブルのイベントレコードを出力できます。

詳細は以下を参照してください。

- [「Debezium db2 変更イベントのキー」](#)
- [「Debezium Db2 変更イベントの値」](#)

3.3.1. Debezium db2 変更イベントのキー

変更イベントのキーには、変更されたテーブルのキーのスキーマと、変更された行の実際のキーのスキーマが含まれます。スキーマとそれに対応するペイロードの両方には、コネクタによってイベントが作成された時点において、変更されたテーブルの **PRIMARY KEY** (または一意の制約) に存在した各列のフィールドが含まれます。

以下の **customers** テーブルについて考えてみましょう。この後に、このテーブルの変更イベントキーの例を示します。

テーブルの例

```
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
);

```

変更イベントキーの例

customers テーブルへの変更をキャプチャーする変更イベントのすべてに、イベントキースキーマがあります。**customers** テーブルに前述の定義がある限り、**customers** テーブルへの変更をキャプチャーする変更イベントのキー構造はすべて以下ようになります。JSON では、以下ようになります。

```

{
  "schema": { ❶
    "type": "struct",
    "fields": [ ❷
      {
        "type": "int32",
        "optional": false,
        "field": "ID"
      }
    ],
    "optional": false, ❸
    "name": "mydatabase.MYSCHEMA.CUSTOMERS.Key" ❹
  },
  "payload": { ❺
    "ID": 1004
  }
}

```

表3.6 変更イベントキーの説明

項目	フィールド名	説明
1	schema	キーのスキーマ部分は、キーの payload 部分の内容を記述する Kafka Connect スキーマを指定します。
2	fields	各フィールドの名前、型、および必要かどうかなど、 payload で想定される各フィールドを指定します。
3	任意	イベントキーの payload フィールドに値が含まれる必要があるかどうかを示します。この例では、キーのペイロードに値が必要です。テーブルにプライマリーキーがない場合は、キーの payload フィールドの値は任意です。

項目	フィールド名	説明
4	mydatabase.MY SCHEMA.CUSTOMERS.Key	<p>キーのペイロードの構造を定義するスキーマの名前。このスキーマは、変更されたテーブルのプライマリーキーの構造を記述します。キースキーマ名の形式は connector-name.database-name.table-name.Key です。この例では、以下のようになります。</p> <ul style="list-style-type: none"> ● mydatabase はこのイベントを生成したコネクタの名前です。 ● MYSHEMA は変更されたテーブルが含まれるデータベーススキーマです。 ● CUSTOMERS は更新されたテーブルです。
5	payload	<p>この変更イベントが生成された行のキーが含まれます。この例では、キーには値が 1004 の1つの ID フィールドが含まれます。</p>

3.3.2. Debezium Db2 変更イベントの値

変更イベントの値はキーよりも若干複雑です。キーと同様に、値には **schema** セクションと **payload** セクションがあります。**schema** セクションには、入れ子のフィールドを含む、**Envelope** セクションの **payload** 構造を記述するスキーマが含まれています。データを作成、更新、または削除する操作のすべての変更イベントには、Envelope 構造を持つ値 **payload** があります。

変更イベントキーの例を紹介するために使用した、同じサンプルテーブルについて考えてみましょう。

テーブルの例

```
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
);
```

customers テーブルのすべての変更イベントのイベント値部分は同じスキーマを指定します。イベント値のペイロードは、イベント型によって異なります。

- **作成イベント**
- **更新イベント**
- **削除イベント**

作成 イベント

以下の例は、**customers** テーブルにデータを作成する操作に対して、コネクタによって生成される変更イベントの値の部分を示しています。

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

```
{
  "type": "struct",
  "fields": [
    {
      "type": "string",
      "optional": false,
      "field": "version"
    },
    {
      "type": "string",
      "optional": false,
      "field": "connector"
    },
    {
      "type": "string",
      "optional": false,
      "field": "name"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "ts_ms"
    },
    {
      "type": "boolean",
      "optional": true,
      "default": false,
      "field": "snapshot"
    },
    {
      "type": "string",
      "optional": false,
      "field": "db"
    },
    {
      "type": "string",
      "optional": false,
      "field": "schema"
    },
    {
      "type": "string",
      "optional": false,
      "field": "table"
    },
    {
      "type": "string",
      "optional": true,
      "field": "change_lsn"
    },
    {
      "type": "string",
      "optional": true,
      "field": "commit_lsn"
    }
  ],
  "optional": false,
}
```

```

    "name": "io.debezium.connector.db2.Source", ❸
    "field": "source"
  },
  {
    "type": "string",
    "optional": false,
    "field": "op"
  },
  {
    "type": "int64",
    "optional": true,
    "field": "ts_ms"
  }
],
"optional": false,
"name": "mydatabase.MYSCHEMA.CUSTOMERS.Envelope" ❹
},
"payload": { ❺
  "before": null, ❻
  "after": { ❼
    "ID": 1005,
    "FIRST_NAME": "john",
    "LAST_NAME": "doe",
    "EMAIL": "john.doe@example.org"
  },
  "source": { ❽
    "version": "2.1.4.Final",
    "connector": "db2",
    "name": "myconnector",
    "ts_ms": 1559729468470,
    "snapshot": false,
    "db": "mydatabase",
    "schema": "MYSCHEMA",
    "table": "CUSTOMERS",
    "change_lsn": "00000027:00000758:0003",
    "commit_lsn": "00000027:00000758:0005",
  },
  "op": "c", ❾
  "ts_ms": 1559729471739 ❿
}
}

```

表3.7 作成 イベント値フィールドの説明

項目	フィールド名	説明
1	schema	値のペイロードの構造を記述する、値のスキーマ。変更イベントの値スキーマは、コネクターが特定のテーブルに生成するすべての変更イベントで同じになります。

項目	フィールド名	説明
2	name	<p>スキーマ セクションで、各 name フィールドは、値のペイロードのフィールドのスキーマを指定します。</p> <p>mydatabase.MYSCHEMA.CUSTOMERS.Value はペイロードの before および after フィールドのスキーマです。このスキーマは customers テーブルに固有です。コネクタは、MYSCHEMA.CUSTOMERS テーブルのすべての行に対してこのスキーマを使用します。</p> <p>before フィールドおよび after フィールドのスキーマ名は logicalName.schemaName.tableName.Value の形式を取ることで、スキーマ名がデータベースで一意になるようにします。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 スキーマは、それぞれ独自に進化し、独自の履歴を持つことになります。</p>
3	name	<p>io.debezium.connector.db2.Source は、ペイロードの source フィールドのスキーマです。このスキーマは Db2 コネクタに固有です。コネクタは生成するすべてのイベントにこれを使用します。</p>
4	name	<p>mydatabase.MYSCHEMA.CUSTOMERS.Envelope は、ペイロードの全体的な構造のスキーマです。mydatabase はデータベース、MYSCHEMA はスキーマ、CUSTOMERS はテーブルです。</p>
5	payload	<p>値の実際のデータ。これは、変更イベントが提供する情報です。</p> <p>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。これは、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です。しかし、Avro コンバーター を使用すると、コネクタが Kafka トピックにストリーミングするメッセージのサイズを大幅に小さくすることができます。</p>
6	before	<p>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。この例のように、op フィールドが create (作成) の c である場合、この変更イベントは新しい内容に対するものであるため、before は null になります。</p>
7	after	<p>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。この例では、after フィールドには、新しい行の ID、FIRST_NAME、LAST_NAME、および EMAIL 列の値が含まれます。</p>

項目	フィールド名	説明
8	source	<p>イベントのソースメタデータを記述する必須のフィールド。source 構造には、この変更に関する Db2 の情報が示され、トレーサビリティが提供されます。また、同じトピックや他のトピックの他のイベントと比較する情報もあり、このイベントが他のイベントの前または後に発生したか、あるいはこのイベントが他のイベントと同じコミットの一部であることを認識できます。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベースに変更が加えられた時点のタイムスタンプ ● 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 ● 新しい行が含まれるデータベース、スキーマ、およびテーブルの名前 ● 変更 LSN ● コミット LSN (このイベントがスナップショットの一部である場合は省略)
9	op	<p>コネクターによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、c は操作によって行が作成されたことを示しています。有効な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (読み取り、スナップショットのみに適用)
10	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

更新イベント

サンプル **customers** テーブルにある更新の変更イベントの値には、そのテーブルの **作成** イベントと同じスキーマがあります。同様に、**更新** イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。以下は、コネクターによって **customers** テーブルでの更新に生成されるイベントの変更イベント値の例になります。

```
{
```

```

"schema": { ... },
"payload": {
  "before": { ❶
    "ID": 1005,
    "FIRST_NAME": "john",
    "LAST_NAME": "doe",
    "EMAIL": "john.doe@example.org"
  },
  "after": { ❷
    "ID": 1005,
    "FIRST_NAME": "john",
    "LAST_NAME": "doe",
    "EMAIL": "noreply@example.org"
  },
  "source": { ❸
    "version": "2.1.4.Final",
    "connector": "db2",
    "name": "myconnector",
    "ts_ms": 1559729995937,
    "snapshot": false,
    "db": "mydatabase",
    "schema": "MYSCHEMA",
    "table": "CUSTOMERS",
    "change_lsn": "00000027:00000ac0:0002",
    "commit_lsn": "00000027:00000ac0:0007",
  },
  "op": "u", ❹
  "ts_ms": 1559729998706 ❺
}
}

```

表3.8 更新 イベント値フィールドの説明

項目	フィールド名	説明
1	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。 更新 イベント値の before フィールドには、各テーブル列のフィールドと、データベースのコミット前にその列にあった値が含まれます。この例では、EMAIL の値が EMAIL value is john.doe@example.com であることに注意してください。
2	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。 before と after の構造を比較すると、この行への更新内容を判断できます。この例では、 EMAIL の値が noreply@example.com となっています。

項目	フィールド名	説明
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。source フィールド構造には 作成 イベントと同じフィールドが含まれますが、一部の値が異なります。たとえば、更新 イベントサンプルの LSN は異なります。この情報を使用して、このイベントを他のイベントと比較し、このイベントが他のイベントの前または後に発生したか、あるいはこのイベントが他のイベントと同じコミットの一部であることを認識できます。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベースに変更が加えられた時点のタイムスタンプ ● 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 ● 新しい行が含まれるデータベース、スキーマ、およびテーブルの名前 ● 変更 LSN ● コミット LSN (このイベントがスナップショットの一部である場合は省略)
4	op	<p>操作の型を記述する必須の文字列。更新 イベントの値では、op フィールドの値は u で、更新によってこの行が変更したことを示します。</p>
5	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>



注記

行のプライマリーキー/一意キーの列を更新すると、行のキーの値が変更されます。キーが変更されると、3つのイベントが Debezium によって出力されます。3つのイベントとは、**DELETE** イベント、行の古いキーを持つ **廃棄 (tombstone)**、およびそれに続く行の新しいキーを持つイベントです。

削除 イベント

削除 変更イベントの値は、同じテーブルの **作成** および **更新** イベントと同じ **schema** の部分になります。サンプル **customers** テーブルの **削除** イベントのイベント値 **payload** は以下のようになります。

```
{
  "schema": { ... },
},
"payload": {
```

```

"before": { ❶
  "ID": 1005,
  "FIRST_NAME": "john",
  "LAST_NAME": "doe",
  "EMAIL": "noreply@example.org"
},
"after": null, ❷
"source": { ❸
  "version": "2.1.4.Final",
  "connector": "db2",
  "name": "myconnector",
  "ts_ms": 1559730445243,
  "snapshot": false,
  "db": "mydatabase",
  "schema": "MYSCHEMA",
  "table": "CUSTOMERS",
  "change_lsn": "00000027:00000db0:0005",
  "commit_lsn": "00000027:00000db0:0007"
},
"op": "d", ❹
"ts_ms": 1559730450205 ❺
}
}

```

表3.9 削除 イベント値フィールドの説明

項目	フィールド名	説明
1	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。 削除 イベント値の before フィールドには、データベースのコミットで削除される前に行にあった値が含まれます。
2	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。 削除 イベント値の after フィールドは null で、行が存在しないことを示します。

項目	フィールド名	説明
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。削除 イベント値の source フィールド構造は、同じテーブルの作成 および更新 イベントと同じになります。多くの source フィールドの値も同じです。削除 イベント値では、ts_ms および LSN フィールドの値や、その他の値が変更された可能性があります。ただし、削除 イベント値の source フィールドは、同じメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベースに変更が加えられた時点のタイムスタンプ ● 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 ● 新しい行が含まれるデータベース、スキーマ、およびテーブルの名前 ● 変更 LSN ● コミット LSN (このイベントがスナップショットの一部である場合は省略)
4	op	<p>操作の型を記述する必須の文字列。op フィールドの値は d で、行が削除されたことを示します。</p>
5	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

削除 変更イベントレコードは、この行の削除を処理するために必要な情報を持つコンシューマーを提供します。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。

Db2 コネクターイベントは、[Kafka の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。ログコンパクションにより、少なくとも各キーの最新のメッセージが保持される限り、一部の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、**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、Kafka が同じキーを持つすべてのメッセージを削除するには、メッセージの値が **null** である必要があります。これを可能にするために、Debezium の Db2 コネクターは **削除** イベントを出力した後に、**null** 値以外で同じキーを持つ特別な廃棄 (tombstone) イベントを出力します。

3.4. DEBEZIUM DB2 コネクターによるデータ型のマッピング方法

Db2 のデータ型の説明は [Db2 SQL Data Types](#) を参照してください。

Db2 コネクタは、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。その値がどのようにイベントで示されるかは、列の Db2 のデータ型によって異なります。ここでは、これらのマッピングについて説明します。デフォルトのデータ型変換が要件に合わない場合は、コネクタ用の [カスタムコンバータの作成](#) が可能です。

詳細は以下を参照してください。

- [基本型](#)
- [時間型](#)
- [タイムスタンプ型](#)
- [表3.13 「10 進数型」](#)

基本型

以下の表では、各 Db2 データ型をイベントフィールドの [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説明します。

- **literal type** 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、値がどのように表現されるかを記述します。
- **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。

表3.10 Db2 の基本データ型のマッピング

DB2 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) および注記
BOOLEAN	BOOLEAN	BOOLEAN 型の列のあるテーブルからのみスナップショットを作成できます。現在、Db2 での SQL レプリケーションは BOOLEAN をサポートしないため、Debezium はこれらのテーブルで CDC を実行できません。別の型の使用を検討してください。
BIGINT	INT64	該当なし
BINARY	BYTES	該当なし
BLOB	BYTES	該当なし
CHAR[(N)]	STRING	該当なし
CLOB	STRING	該当なし
DATE	INT32	io.debezium.time.Date タイムゾーン情報のないタイムスタンプの文字列表現

DB2 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) および注記
DECFLOAT	BYTES	org.apache.kafka.connect.data.Decimal
DECIMAL	BYTES	org.apache.kafka.connect.data.Decimal
DBCLOB	STRING	該当なし
DOUBLE	FLOAT64	該当なし
INTEGER	INT32	該当なし
REAL	FLOAT32	該当なし
SMALLINT	INT16	該当なし
TIME	INT32	io.debezium.time.Time タイムゾーン情報のない時刻の文字列表現
TIMESTAMP	INT64	io.debezium.time.MicroTimestamp タイムゾーン情報のないタイムスタンプの文字列表現
VARBINARY	BYTES	該当なし
VARCHAR[(N)]	STRING	該当なし
VARGRAPHIC	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。

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。明示的な列値が指定されない限り、変更イベントにはフィールドのデフォルト値が含まれます。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。

時間型

タイムゾーン情報が含まれる Db2 の **DATETIMEOFFSET** データ型以外に、時間型がマッピングされる仕組みは **time.precision.mode** コネクター設定プロパティの値によって異なります。ここでは、以下のマッピングについて説明します。

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

time.precision.mode=adaptive

time.precision.mode 設定プロパティがデフォルトの **adaptive** に設定された場合、コネクタは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。これにより、イベントがデータベースの値を **正確** に表すようになります。

表3.11 **time.precision.mode** が **adaptive** の場合のマッピング

DB2 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) および注記
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。
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。

time.precision.mode=connect

time.precision.mode 設定プロパティが **connect** に設定された場合、コネクタは 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 の論理型のみを処理でき、可変精度の時間値を処理できない場合に便利です。ただし、Db2 はマイクロ秒の 10 分の 1 の精度をサポートするため、**connect** 時間精度を指定してコネクタ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**。

表3.12 **time.precision.mode** が **connect** の場合のマッピング

DB2 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) および注記
DATE	INT32	org.apache.kafka.connect.data.Date エポックからの日数を表します。

DB2 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) および注記
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。Db2 では、範囲が 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 が 3 よりも大きい場合は、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。

タイムスタンプ型

DATETIME 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 **DATETIME** 値は、1529507596000 という値の **io.debezium.time.Timestamp** 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ません。

表3.13 10 進数型

DB2 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) および注記
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。

3.5. DEBEZIUM コネクターを実行するための DB2 の設定

Db2 テーブルにコミットされた変更イベントを Debezium がキャプチャーするには、必要な権限を持つ Db2 データベース管理者が、変更データキャプチャー (CDC) のデータベースでテーブルを設定する必要があります。Debezium の実行を開始した後、キャプチャーエージェントの設定を調整してパフォーマンスを最適化できます。

Debezium コネクターと使用するために Db2 を設定する場合の詳細は、以下を参照してください。

- 「変更データキャプチャーの Db2 テーブルの設定」
- 「Db2 キャプチャーエージェント設定のサーバー負荷およびレイテンシーへの影響」
- 「DB2 キャプチャーエージェントの設定パラメーター」

3.5.1. 変更データキャプチャーの Db2 テーブルの設定

テーブルをキャプチャーモードにするために、Debezium ではユーザー定義関数 (UDF) のセットが提供されます。ここでは、これらの管理 UDF をインストールおよび実行する手順を説明します。また、Db2 制御コマンドを実行してテーブルをキャプチャーモードにすることもできます。その後、管理者は Debezium がキャプチャーする各テーブルに対して、CDC を有効にする必要があります。

前提条件

- **db2inst1** ユーザーとして Db2 にログインしている。
- Db2 ホストの `$HOME/asncdctools/src` ディレクトリーで Debezium 管理 UDF を使用できる。UDF は [Debezium サンプルリポジトリー](#) から入手できます。

手順

1. Db2 で提供される **bldrtn** コマンドを使用して、Db2 サーバーホストで Debezium 管理 UDF をコンパイルします。

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

```
./bldrtn asncdc
```

2. データベースが稼働していない場合は起動します。**DB_NAME** は、Debezium が接続するデータベースの名前に置き換えます。

```
db2 start db DB_NAME
```

3. JDBC が Db2 メタデータカタログを読み取りできるようにします。

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

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

4. データベースが最近バックアップされたことを確認します。ASN エージェントには、読み取りを始める最新の開始点が必要です。バックアップを実行する必要がある場合は、以下のコマンドを実行して、最新のバージョンのみを利用できるようにデータをプルーニングします。古いバージョンのデータを保持する必要がない場合は、バックアップの場所に **dev/null** を指定します。

- a. データベースをバックアップします。**DB_NAME** および **BACK_UP_LOCATION** を適切な値に置き換えます。

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

- b. データベースを再起動します。


```
db2 restart db DB_NAME
```

- データベースに接続して、Debezium 管理 UDF をインストールします。**db2inst1** ユーザーとしてログインしていることを前提とするため、UDF が **db2inst1** ユーザーにインストールされている必要があります。

```
db2 connect to DB_NAME
```

- Debezium 管理 UDF をコピーし、その権限を設定します。

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

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

- ASN キャプチャーエージェントを開始および停止する Debezium UDF を有効にします。

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

- ASN 制御テーブルを作成します。

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

- テーブルをキャプチャーモードに追加し、キャプチャーモードからテーブルを削除する Debezium UDF を有効にします。

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

Db2 サーバーを設定したら、UDF を使用して SQL コマンドで Db2 レプリケーション (ASN) を制御します。UDF によっては戻り値が必要な場合があります。この場合、SQL の **VALUE** ステートメントを使用して呼び出します。その他の UDF には、SQL の **CALL** ステートメントを使用します。

- ASN エージェントを起動します。

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

前述のステートメントは、以下のいずれかの結果を返します。

- **asncap is already running**

- **start --> <COMMAND>**

この場合は、以下の例のように、ターミナルウィンドウに指定の **<COMMAND>** を入力します。

```
/database/config/db2inst1/sqllib/bin/asncap capture_schema=asncdc
capture_server=SAMPLE &
```

- テーブルをキャプチャーモードにします。キャプチャーする各テーブルに対して、以下のステートメントを呼び出します。**MYSHEMA** は、キャプチャーモードにするテーブルが含まれるスキーマの名前に置き換えます。同様に、**MYTABLE** は、キャプチャーモードにするテーブルの名前に置き換えます。

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

12. ASN サービスを再初期化します。

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

関連情報

[Debezium Db2 管理 UDF の参照テーブル](#)

3.5.2. Db2 キャプチャーエージェント設定のサーバー負荷およびレイテンシーへの影響

データベース管理者がソーステーブルに対して変更データキャプチャーを有効にすると、キャプチャーエージェントの実行が開始されます。エージェントは新しい変更イベントレコードをトランザクションログから読み取り、イベントレコードをキャプチャーテーブルに複製します。変更がソーステーブルにコミットされてから、対応する変更テーブルに変更が反映される間、常に短いレイテンシーが間隔で発生します。この遅延間隔は、ソーステーブルで変更が発生したときから、Debezium がその変更を Apache Kafka にストリーミングできるようになるまでの差を表します。

データの変更に素早く対応する必要があるアプリケーションについては、ソースとキャプチャーテーブル間で密接に同期を維持するのが理想的です。キャプチャーエージェントを実行してできるだけ迅速に変更イベントを継続的に処理すると、スループットが増加し、レイテンシーが減少するため、イベントの発生後にほぼリアルタイムで新しいイベントレコードが変更テーブルに入力されることを想像するかもしれません。しかし、これは必ずしもそうであるとは限りません。同期を即時に行うとパフォーマンスに影響します。変更エージェントが新しいイベントレコードについてデータベースにクエリーを実行するたびに、データベースホストの CPU 負荷が増加します。サーバーへの負荷が増えると、データベース全体のパフォーマンスに悪影響を及ぼす可能性があり、特にデータベースの使用がピークに達するときにトランザクションの効率が低下する可能性があります。

データベースメトリクスを監視して、サーバーがキャプチャーエージェントのアクティビティをサポートできなくなるレベルにデータベースが達した場合に認識できるようにすることが重要となります。キャプチャーエージェントの実行中にパフォーマンスの問題が発生した場合は、キャプチャーエージェント設定を調整して CPU の負荷を減らします。

3.5.3. DB2 キャプチャーエージェントの設定パラメーター

Db2 では、**IBMSNAP_CAPPARMS** テーブルにはキャプチャーエージェントの動作を制御するパラメーターが含まれています。これらのパラメーターの値を調整して、キャプチャープロセスの設定を調整すると、CPU の負荷を減らしながら許容レベルのレイテンシーを維持することができます。



注記

Db2 のキャプチャーエージェントパラメーターの設定方法に関する具体的なガイダンスは、本書の範囲外となります。

IBMSNAP_CAPPARMS テーブルでは、CPU 負荷の削減に最も影響を与えるパラメーターは以下のとおりです。

COMMIT_INTERVAL

- キャプチャーエージェントがデータを変更データテーブルにコミットするまで待つ期間を秒単位で指定します。
- 値が大きいほど、データベースホストの負荷が減少し、レイテンシーが増加します。

- デフォルト値は **30** です。

SLEEP_INTERVAL

- キャプチャーエージェントがアクティブなトランザクションログの最後に到達した後に、新しいコミットサイクルの開始まで待つ期間を秒単位で指定します。
- 値が大きいほど、サーバーの負荷が減少し、レイテンシーが増加します。
- デフォルト値は **5** です。

関連情報

- キャプチャーエージェントパラメーターの詳細は、Db2 のドキュメントを参照してください。

3.6. DEBEZIUM DB2 コネクターのデプロイ

以下の方法のいずれかを使用して Debezium Db2 コネクターをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタープラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。



重要

ライセンス要件のため、Debezium Db2 コネクターアーカイブには、Debezium が Db2 データベースに接続するために必要な Db2 JDBC ドライバーは含まれていません。コネクターがデータベースにアクセスできるようにするには、コネクター環境にドライバーを追加する必要があります。ドライバーの入手方法については、[Db2JDBC ドライバーの入手](#)を参照してください。

関連情報

- [「Debezium Db2 コネクター設定プロパティの説明」](#)

3.6.1. Db2 JDBC ドライバーの取得

Debezium が Db2 データベースに接続するために必要な Db2 JDBC ドライバーファイルは、ライセンスの関係で Debezium Db2 コネクターアーカイブに含まれていません。ドライバーは、Maven Central からダウンロード可能です。使用するデプロイメント方法に応じて、Kafka Connect カスタムリソースまたはコネクターイメージの構築に使用する Dockerfile にコマンドを追加して、ドライバーを取得することができます。

- AMQ Streams を使用して Kafka Connect イメージにコネクターを追加する場合は、[「AMQ Streams を使用した Debezium Db2 コネクターのデプロイ」](#) に示すように、**KafkaConnect** カスタムリソースの **builds.plugins.artifact.url** にドライバーの Maven Central の場所を追加してください。
- Dockerfile を使用してコネクター用のコンテナイメージを構築する場合、Dockerfile に **curl** コマンドを挿入して、Maven Central から必要なドライバーファイルをダウンロードするための URL を指定します。詳細は、[「Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium Db2 コネクターのデプロイ」](#) を参照してください。

3.6.2. AMQ Streams を使用した Db2 コネクターステプロイメント

Debezium 1.7 以降、Debezium コネクターステプロイに推奨される方法は、AMQ Streams を使用してコネクターステプラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

ステプロイメントステプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクターステアーツィファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクターステがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクターステを開始します。

Kafka Connect イメージのビルド仕様では、ステプロイ可能なコネクターステを指定できます。各コネクターステプラグインに対して、ステプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーツィファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーツィファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をステプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクターステを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。
- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

3.6.3. AMQ Streams を使用した Debezium Db2 コネクターステステプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクターステをステプロイするには、最初にコネクターステ用の Kafka Connect イメージをビルドする必要がありました。コネクターステを OpenShift にステプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクターステプラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドステプロセス中、AMQ Streams Operator は Debezium コネクターステ定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリーまたは別の設定済みの HTTP サーバーから必要なアーツィファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのステプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクターステを起動します。

前提条件

- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスターが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合

- [ImageStream](#) リソースを新規コンテナイメージを保存するためにクラスターにデプロイします。クラスターの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスターにログインします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例3.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium Db2 コネクタアーカイブ。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクタで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプト SMT アーカイブと Debezium コネクタで使用する関連言語の依存関係。SMT アーカイブおよび言語の依存関係は任意のコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場合にのみ、これらのコンポーネントを追加します。

- Db2 JDBC ドライバー。Db2 データベースに接続するために必要ですが、コネクタアーカイブには含まれていません。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
    type: imagestream ❹
    image: debezium-streams-connect:latest
  plugins: ❺
    - name: debezium-connector-db2
      artifacts:
        - type: zip ❻
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-db2/2.1.4.Final-redhat-00001/debezium-connector-db2-2.1.4.Final-redhat-00001-plugin.zip ❼
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3.0.Final-redhat-❸/build-number/apicurio-registry-distro-connect-converter-2.3.0.Final-redhat-❸.zip ❽
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip ❾
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar ❿
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json/3.0.11/groovy-json-3.0.11.jar
        - type: jar ❶❶
          url: https://repo1.maven.org/maven2/com/ibm/db2/jcc/11.5.0.0/jcc-11.5.0.0.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

  ...

```

表3.14 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。

項目	説明
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクターを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクターのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタープラグインに対して、コネクターと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクターアーカイブは、 .zip ファイル形式で提供されます。JDBC ドライバーファイルは .jar 形式です。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。
7	artifacts.url の値は、コネクターアーティファクトのファイルを格納する Maven リポジトリーなどの HTTP サーバーのアドレスを指定します。OpenShift クラスターは指定されたサーバーにアクセスできる必要があります。
8	(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクターが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。
9	(オプション) Debezium コネクターで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベーススルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。

項目	説明
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 40px; height: 40px; background-color: black; margin-right: 10px;"></div> <div> <p>重要</p> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタプラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクタが失敗します。</p> </div> </div> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>
11	<p>Maven Central にある Db2 JDBC ドライバーの場所を指定します。必要なドライバーが Debezium Db2 コネクタアーカイブに含まれていない。</p>

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスタを起動します。設定に一覧表示されているコネクタアーティファクトはクラスタで利用できます。

- KafkaConnector** リソースを作成し、デプロイする各コネクタのインスタンスを定義します。
たとえば、以下の **KafkaConnector** CR を作成し、**db2-inventory-connector.yaml** として保存します。

例3.2 Debezium コネクタの **KafkaConnector** カスタムリソースを定義する **db2-inventory-connector.yaml** ファイル

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
name: inventory-connector-db2 1
```



```

spec:
  class: io.debezium.connector.db2.Db2ConnectorConnector 2
  tasksMax: 1 3
  config: 4
    schema.history.internal.kafka.bootstrap.servers: debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092
    schema.history.internal.kafka.topic: schema-changes.inventory
    database.hostname: db2.debezium-db2.svc.cluster.local 5
    database.port: 50000 6
    database.user: debezium 7
    database.password: dbz 8
    database.dbname: mydatabase 9
    topic.prefix: inventory-connector-db2 10
    table.include.list: public.inventory 11
  ...

```

表3.15 コネクタ設定の説明

項目	説明
1	Kafka Connect クラスタに登録するコネクタの名前。
2	コネクタクラスの名称。
3	同時に動作できるタスクの数。
4	コネクタの設定。
5	ホストデータベースインスタンスのアドレス。
6	データベースインスタンスのポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名称。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。
9	変更をキャプチャするデータベースの名称。
10	データベースインスタンスまたはクラスタのトピック接頭辞。 指定する名称は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクタから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名称はクラスタのコネクタ間で一意である必要があります。 コネクタを Avro コネクタ と統合する場合、この名前空間は関連する Kafka Connect スキーマの名称や、対応する Avro スキーマの名称空間でも使用されます。
11	コネクタが変更イベントをキャプチャするテーブルの一覧。

5. 以下のコマンドを実行してコネクタリソースを作成します。

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium Db2 のデプロイメントを確認](#)する準備が整いました。

3.6.4. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium Db2 コネクタのデプロイ

Debezium Db2 コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、以下のカスタムリソース (CR) を作成する必要があります。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium Db2 コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用したのと同じ OpenShift インスタンスに適用します。

前提条件

- Db2 が実行中で、[Db2 を設定して Debezium コネクタと連携する](#)手順が完了済みである必要があります。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。
- Podman または Docker がインストールされている。
- Kafka Connect サーバーは、Db2 用の必要な JDBC ドライバーをダウンロードするために、Maven Central にアクセスすることができます。また、ドライバーのローカルコピー、またはローカルの Maven リポジトリーや他の HTTP サーバーから利用可能なものを使用することもできます。
- Debezium コネクタを実行するコンテナを追加する予定のコンテナレジストリー ([quay.io](#) や [docker.io](#) など) でコンテナを作成および管理するアカウントとパーミッションを持っている。

手順

1. Kafka Connect の Debezium Db2 コンテナを作成します。

- a. **registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12** をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-db2.yaml ❶
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium ❷
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
db2/2.1.4.Final-redhat-00001/debezium-connector-db2-2.1.4.Final-redhat-00001-
plugin.zip \
&& unzip debezium-connector-db2-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-db2-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://repo1.maven.org/maven2/com/ibm/db2/jcc/11.5.0.0/jcc-11.5.0.0.jar
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-db2.yaml** という名前の Docker ファイルを作成します。

- b. 前のステップで作成した **debezium-container-for-db2.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

```
podman build -t debezium-container-for-db2:latest .
```

```
docker build -t debezium-container-for-db2:latest .
```

上記のコマンドは、**debezium-container-for-db2** という名前のコンテナイメージを構築します。

- c. カスタムイメージを quay.io などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

```
podman push <myregistry.io>/debezium-container-for-db2:latest
```

```
docker push <myregistry.io>/debezium-container-for-db2:latest
```

- d. 新しい Debezium Db2 **KafkaConnect** カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の

KafkaConnect CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  #...
  image: debezium-container-for-db2 ❷
...

```

項目	説明
1	KafkaConnector リソースはこの Kafka Connect クラスタでコネクタを設定するために使用されることを、 metadata.annotations は Cluster Operator に示します。
2	spec.image は Debezium コネクタを実行するために作成したイメージの名前を指定します。設定された場合、このプロパティによって Cluster Operator の STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE 変数がオーバーライドされます。

- e. 以下のコマンドを入力して、**KafkaConnect** CR を OpenShift Kafka Connect 環境に適用します。

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

このコマンドは、Debezium コネクタを実行するために作成したイメージの名前を指定する Kafka Connect インスタンスを追加します。

- Debezium Db2 コネクタインスタンスを設定する **KafkaConnector** カスタムリソースを作成します。
通常、コネクタに使用できる設定プロパティを使用して、**.yaml** ファイルに Debezium Db2 コネクタを設定します。コネクタ設定は、Debezium に対して、スキーマおよびテーブルのサブセットにイベントを生成するよう指示する可能性があり、または機密性の高い、大きすぎる、または不必要な指定の列で Debezium が値を無視、マスク、または切り捨てするようにプロパティを設定する可能性もあります。

以下の例では、ポート **50000** で Db2 サーバーホスト **192.168.99.100** に接続する Debezium コネクタを設定します。このホストには、データベース名が **mydatabase**、テーブル名が **inventory**、サーバーの論理名が **inventory-connector-db2** があります。

Db2 inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector-db2 ❶

```

```

labels:
  strimzi.io/cluster: my-connect-cluster
annotations:
  strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.db2.Db2Connector ②
  tasksMax: 1 ③
  config: ④
    database.hostname: 192.168.99.100 ⑤
    database.port: 50000 ⑥
    database.user: db2inst1 ⑦
    database.password: Password! ⑧
    database.dbname: mydatabase ⑨
    topic.prefix: inventory-connector-db2 ⑩
    table.include.list: public.inventory ⑪
  ...

```

表3.16 コネクター設定の説明

項目	説明
1	Kafka Connect クラスタに登録する場合のコネクターの名前。
2	この Db2 コネクタークラスの名前。
3	1度に1つのタスクのみが動作する必要があります。
4	コネクターの設定。
5	Db2 インスタンスのアドレスであるデータベースホスト。
6	Db2 インスタンスのポート番号。
7	Db2 ユーザーの名前。
8	Db2 ユーザーのパスワード。
9	変更をキャプチャーするデータベースの名前。
10	名前空間を形成する Db2 インスタンス/クラスタの論理名で、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 Arvo コネクター が使用される場合の対応する Avro スキーマの名前空間で使用されます。
11	コネクターは public.inventory テーブルからのみ変更をキャプチャーします。

- Kafka Connect でコネクターインスタンスを作成します。たとえば、**KafkaConnector** リソースを **inventory-connector.yaml** ファイルに保存した場合は、以下のコマンドを実行します。

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

上記のコマンドは **inventory-connector** を登録し、コネクタは **KafkaConnector** CR に定義されている **mydatabase** データベースに対して実行を開始します。

Debezium Db2 コネクタに設定できる設定プロパティの完全リストは、[Db2 コネクタプロパティ](#) を参照してください。

結果

コネクタが起動すると、コネクタが変更をキャプチャーするように設定された Db2 データベーステーブルの [整合性スナップショット](#) が実行されます。その後、コネクタは行レベルの操作のデータ変更イベントの生成を開始し、変更イベントレコードを Kafka トピックにストリーミングします。

3.6.5. Debezium Db2 コネクタが実行していることの確認

コネクタがエラーなしで正常に起動すると、コネクタがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウンストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクタが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクタのステータスを確認します。
- コネクタがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクタが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクタが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。
 - OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクタの名前をクリックします (例: **inventory-connector-db2**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
 - ターミナルウィンドウから以下を実行します。

- a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-db2 -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例3.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-db2
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector

...

Status:
Conditions:
  Last Transition Time: 2021-12-08T17:41:34.897153Z
  Status:              True
  Type:                Ready
Connector Status:
Connector:
  State:  RUNNING
  worker_id: 10.131.1.124:8083
Name:    inventory-connector-db2
Tasks:
  Id:    0
  State:  RUNNING
  worker_id: 10.131.1.124:8083
  Type:  source
Observed Generation: 1
Tasks Max: 1
Topics:
  inventory-connector-db2.inventory
  inventory-connector-db2.inventory.addresses
  inventory-connector-db2.inventory.customers
  inventory-connector-db2.inventory.geom
  inventory-connector-db2.inventory.orders
  inventory-connector-db2.inventory.products
  inventory-connector-db2.inventory.products_on_hand
Events: <none>
```

2. コネクターによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. Home → Search に移動します。

- b. Search ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. KafkaTopics リストから確認するトピックの名前をクリックします (例: **inventory-connector-db2.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例3.4 KafkaTopic リソースのステータス

```

NAME                                     CLUSTER
PARTITIONS REPLICATION FACTOR  READY
connect-cluster-configs                debezium-kafka-cluster  1
1                                     True
connect-cluster-offsets                debezium-kafka-cluster  25
1                                     True
connect-cluster-status                 debezium-kafka-cluster  5
1                                     True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fad1c6a
debezium-kafka-cluster 50             1             True
inventory-connector-db2--a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster 1             1             True
inventory-connector-db2.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480    debezium-kafka-cluster
1             1             True
inventory-connector-db2.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b    debezium-kafka-cluster  1
1             True
inventory-connector-db2.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5    debezium-kafka-cluster
1             1             True
inventory-connector-db2.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d    debezium-kafka-cluster
1             1             True
inventory-connector-db2.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef    debezium-kafka-cluster  1
1             True
inventory-connector-db2.inventory.products_on_hand---
8649e0f17fcc9212e266e31a7aeaa4585e5c6b5    debezium-kafka-cluster  1
1             True
schema-changes.inventory                debezium-kafka-cluster
1             1             True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
kafka-cluster 1             1             True

```



```
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b debezium-kafka-cluster 1 1
True
```

3. トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。

```
oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>
```

以下に例を示します。

```
oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-db2.inventory.products_on_hand
```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory_connector_db2.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例3.5 Debezium 変更イベントの内容

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},"optional":false,"name":"inventory-
connector-db2.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-db2.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-db2.inventory.products_on_hand.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":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}]},"optional":false,"name":"io.debezium.connecto
```

```
r.db2.Source", "field": "source"}, {"type": "string", "optional": false, "field": "op"},
{"type": "int64", "optional": true, "field": "ts_ms"}, {"type": "struct", "fields":
[{"type": "string", "optional": false, "field": "id"},
{"type": "int64", "optional": false, "field": "total_order"},
{"type": "int64", "optional": false, "field": "data_collection_order"}], "optional": true, "field": "transacti
on"}], "optional": false, "name": "inventory-connector-
db2.inventory.products_on_hand.Envelope"}, "payload": {"before": null, "after":
{"product_id": 101, "quantity": 3, "source": {"version": "2.1.4.Final-redhat-
00001", "connector": "db2", "name": "inventory-connector-
db2", "ts_ms": 1638985247805, "snapshot": "true", "db": "inventory", "sequence": null, "table": "pro
ducts_on_hand", "server_id": 0, "gtid": null, "file": "db2-
bin.000003", "pos": 156, "row": 0, "thread": null, "query": null}, "op": "r", "ts_ms": 1638985247805, "t
ransaction": null}}
```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

3.6.6. Debezium Db2 コネクタースettingsプロパティの説明

Debezium Db2 コネクタースには、アプリケーションに適したコネクタース動作を実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。プロパティに関する情報は、以下のように設定されています。

- [必要な設定プロパティ](#)
- [高度な設定プロパティ](#)
- Debezium がデータベース履歴トピックから読み取るイベントを処理する方法を制御する [データベース履歴コネクタースettingsプロパティ](#)。
 - [パススルーデータベーススキーマ履歴プロパティ](#)
- データベースドライバーの動作を制御する [パススルーデータベースドライバープロパティ](#)。

必要な Debezium Db2 コネクタースettingsプロパティ

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

プロパティ	デフォルト	説明
name	デフォルトなし	コネクタースの一意名。同じ名前でも再登録を試みると失敗します。このプロパティはすべての Kafka Connect コネクタースに必要です。
connector.class	デフォルトなし	コネクタースの Java クラスの名前。Db2 コネクタースには、常に io.debezium.connector.db2.Db2Connector の値を使用します。

プロパティ	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があるタスクの最大数。Db2 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	デフォルトなし	Db2 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	50000	Db2 データベースサーバーの整数のポート番号。
database.user	デフォルトなし	Db2 データベースサーバーに接続するための Db2 データベースユーザーの名前。
database.password	デフォルトなし	Db2 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	デフォルトなし	変更をストリーミングする Db2 データベースの名前
topic.prefix	デフォルトなし	<p>Debezium が変更をキャプチャーするデータベースをホストする特定の Db2 データベースサーバーの namespace を提供するトピック接頭辞。トピックの接頭辞名には、英数字、ハイフン、ドット、およびアンダースコアのみを使用する必要があります。このトピック 接頭辞は、このコネクターからレコードを受け取るすべての Kafka トピックに使用されるため、トピック 接頭辞は他のすべてのコネクターで一貫である必要があります。</p> <div data-bbox="884 1469 1426 2063" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <div style="display: flex; align-items: center;">  <div> <p>警告</p> <p>このプロパティの値を変更しないでください。名前を変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。また、コネクターはデータベーススキーマ履歴トピックを復元できません。</p> </div> </div> </div>

プロパティ	デフォルト	説明
table.include.list	デフォルトなし	<p>コネクターで変更をキャプチャー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)。このプロパティが設定されている場合、コネクターは指定されたテーブルからのみ変更をキャプチャします。各識別子の形式は schemaName.tableName です。デフォルトでは、コネクターはシステム以外のテーブルすべての変更をキャプチャーします。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.exclude.list プロパティも設定しないでください。</p>
table.exclude.list	デフォルトなし	<p>コネクターで変更をキャプチャーしない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)。コネクターは exclude リストに含まれていないシステム以外のテーブルごとに変更をキャプチャーします。各識別子の形式は schemaName.tableName です。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.include.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
column.include.list	空の文字列	<p>変更イベントレコード値に含め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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。このプロパティを設定に含める場合は、column.exclude.list プロパティを設定しないでください。</p>
column.exclude.list	空の文字列	<p>変更イベント値から除外す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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。プライマリーキー列は、値から除外された場合でも、イベントのキーに常に含まれます。このプロパティを設定に含める場合は、column.include.list プロパティを設定しないでください。</p>

プロパティ	デフォルト	説明
<p><code>column.mask.hash.hashAlgorithm.with.salt.salt</code></p>	<p>該当なし</p>	<p>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)。列の完全修飾名の形式は <code>schemaName.tableName.columnName</code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。作成された変更イベントレコードでは、指定された列の値は仮名に置き換えられます。</p> <p>仮名は、指定された <code>hashAlgorithm</code> と <code>salt</code> を適用すると得られるハッシュ化された値で設定されます。使用されるハッシュ関数に基づいて、参照整合性は保持され、列値は仮名に置き換えられます。サポートされるハッシュ関数は、Java Cryptography Architecture Standard Algorithm Name Documentation の MessageDigest セクション に説明されています。</p> <p>以下の例では、CzQMA0cB5K が無作為に選択された salt になります。</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>必要な場合は、仮名は自動的に列の長さに短縮されます。コネクタ設定には、異なるハッシュアルゴリズムと salt を指定する複数のプロパティを含めることができます。</p> <p>使用される <code>hashAlgorithm</code>、選択された <code>salt</code>、および実際のデータセットによっては、結果として得られるデータセットが完全にマスクされないことがあります。</p>

プロパティ	デフォルト	説明
time.precision.mode	adaptive	<p>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。</p> <p>adaptive 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。</p> <p>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。詳細は 一時的な型 を参照してください。</p>
tombstones.on.delete	true	<p>削除 イベントの後に廃棄 (tombstone) イベントが続くかどうかを制御します。</p> <p>true: 削除操作は、削除 イベントと後続の破棄 (tombstone) イベントで表されます。</p> <p>false - delete イベントのみが出力されます。</p> <p>log compaction がトピックで有効になっている場合には、ソースレコードの削除後に廃棄 (tombstone) イベントを出力すると (デフォルト動作)、Kafka は削除された行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。</p>
include.schema.changes	true	<p>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と、スキーマ更新を記述する JSON 構造である値で記録されます。これは、コネクターがデータベーススキーマ履歴を内部で記録する方法には依存しません。</p>

プロパティ	デフォルト	説明
<code>column.truncate.to.length.chars</code>	該当なし	<p>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)。プロパティ名の 長さ で指定された文字数を超えた場合に、一連の列のデータを切り捨てる場合は、このプロパティを設定します。 length を正の整数値に設定します (例: column.truncate.to.20.chars)。</p> <p>列の完全修飾名は、次の形式に従います: schemaName.tableName.columnName列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>
<code>column.mask.with.length.chars</code>	該当なし	<p>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)。一連の列の値をコネクタでマスクする場合 (たとえば、列に機密データが含まれている場合) は、このプロパティを設定します。 length を正の整数に設定して、指定された列のデータをプロパティ名の 長さ で指定されたアスタリスク (*) 文字数で置き換えます。指定した列のデータを空の文字列に置き換えるには、 長さ を 0 (ゼロ) に設定します。</p> <p>列の完全修飾名は、次の形式に従います: schemaName.tableName.columnName。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>

プロパティ	デフォルト	説明
column.propagate.source.type	該当なし	<p>列のメタデータを表す追加パラメーターをコネクターに発行させたい列の完全修飾名に一致する、オプションのコンマ区切りの正規表現のリスト。このプロパティが設定されている場合、コネクターは次のフィールドをイベントレコードのスキーマに追加します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクターがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、次のいずれかの形式に従います: databaseName.tableName.columnName、または databaseName.schemaName.tableName.columnName.</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p>

プロパティ	デフォルト	説明
<p>datatype.propagate.source.type</p>	<p>該当なし</p>	<p>データベース内の列に対して定義されているデータ型の完全修飾名を指定する正規表現のオプションのコンマ区切りリスト。このプロパティが設定されている場合、データ型が一致する列に対して、コネクタはスキーマに次の追加フィールドを含むイベントレコードを発行します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクタがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名の形式は、<code>databaseName.tableName.typeName</code>、または <code>databaseName.schemaName.tableName.typeName</code> のいずれかになります。</p> <p>データ型の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データ型の名前文字列全体に対して照合されます。式は、型名に存在する可能性のある部分文字列と一致しません。</p> <p>Db2 固有のデータ型名の一覧は、Db2 データ型マッピング を参照してください。</p>

プロパティ	デフォルト	説明
<p>message.key.columns</p>	<p>空の文字列</p>	<p>指定のテーブルの Kafka トピックに公開する変更イベントレコードのカスタムメッセージキーを形成するためにコネクターが使用する列を指定する式のリスト。</p> <p>デフォルトでは、Debezium はテーブルのプライマリーキー列を、出力するレコードのメッセージキーとして使用します。デフォルトの代わりに、またはプライマリーキーのないテーブルのキーを指定するには、1つ以上の列をもとにカスタムメッセージキーを設定できます。</p> <p>テーブルにカスタムメッセージキーを設定するには、テーブルを列挙した後、メッセージキーとして使用する列を列挙します。各リストエントリは以下の形式を取ります。</p> <p><fully-qualified_tableName>:<keyColumn>,<keyColumn></p> <p>複数の列名をベースにテーブルキーを作成するには、列名の間にはコンマを挿入します。各完全修飾テーブル名は、以下の形式の正規表現です。</p> <p><schemaName>.<tableName></p> <p>プロパティは複数のテーブルのエントリをリストできます。リスト内の異なるテーブルのエントリは、セミコロンを使用して、区切ります。</p> <p>以下の例は、テーブル inventory.customers and purchaseorders:</p> <p>inventory.customers:pk1,pk2;(*).purchaseorders:pk3,pk4</p> <p>のメッセージキーを設定します。上記の例では、列 pk1 と pk2 がテーブル inventory.customer のメッセージキーとして指定されます。スキーマで purchaseorders を解決する場合には、列 pk3 と pk4 はメッセージキーとして機能します。</p>

プロパティ	デフォルト	説明
<code>schema.name.adjustment.mode</code>	none	<p>コネクターで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能:</p> <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。

高度なコネクター設定プロパティ

以下の **高度な** 設定プロパティには、ほとんどの状況で機能する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	デフォルト	説明
<code>converters</code>	デフォルトなし	<p>コネクターが使用できる カスタムコンバーター インスタンスのシンボリック名のコマンド区切りリストを列挙します。以下に例を示します。</p> <p>isbn</p> <p>コネクターがカスタムコンバータを使用できるようにするには、converters タプロパティを設定する必要があります。</p> <p>コネクターに設定するコンバーターごとに、コンバーターインターフェイスを実装するクラスの完全修飾名を指定する .type プロパティも追加する必要があります。.type プロパティでは、以下の形式を使用します。</p> <p><converterSymbolicName>.type</p> <p>以下に例を示します。</p> <pre>isbn.type: io.debezium.test.IsbnConverter</pre> <p>設定されたコンバータの動作をさらに制御したい場合は、1つ以上の設定パラメーターを追加して、コンバータに値を渡すことができます。追加の設定パラメーターとコンバーターを関連付けるには、パラメーター名の前にコンバーターのシンボリック名を付けます。以下に例を示します。</p> <pre>isbn.schema.name: io.debezium.db2.type.Isbn</pre>

プロパティ	デフォルト	説明
<code>snapshot.mode</code>	<code>Initial</code>	<p>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。</p> <p>initial - キャプチャーモードのテーブルの場合、コネクターはテーブルとそのテーブルのデータのスナップショットを作成します。これは、Kafka トピックにデータの完全な表現を入力するのに役立ちます。</p> <p>initial_only - 初期のように構造とデータのスナップショットを取得しますが、スナップショットが完了するとストリーミングの変更に移行しません。</p> <p>schema_only - キャプチャーモードのテーブルの場合には、コネクターはテーブルのスキーマのみのスナップショットを作成します。これは、現時点以降に発生する変更のみを Kafka トピックに出力する必要がある場合に便利です。スナップショットの完了後、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。</p>

プロパティ	デフォルト	説明
<code>snapshot.isolation.mode</code>	<code>repeatable_read</code>	<p>スナップショットの実行中に、トランザクション分離レベルとキャプチャーモードのテーブルをロックする期間を制御します。使用できる値は次のとおりです。</p> <p>read_uncommitted - 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。このモードでは、データの整合性は保証されず、一部のデータが損失または破損する可能性があります。</p> <p>read_committed - 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。新しい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回発生す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に適しています。</p> <p>repeatable_read - 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がテーブル行を更新しないようにします。新しい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回発生す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に適しています。</p> <p>exclusive - 繰り返し可能な読み取り分離レベルを使用しますが、すべてのテーブルを読み取るために排他的ロックを使用します。このモードは、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がテーブル行を更新しないようにします。exclusive モードのみが完全な整合性を保証し、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設定します。</p>
<code>event.processing.failure.handling.mode</code>	<code>fail</code>	<p>イベントの処理中にコネクタが例外を処理する方法を指定します。使用できる値は次のとおりです。</p> <p>fail - コネクタは問題のあるイベントのオフセットをログに記録し、処理を停止します。</p> <p>warn - コネクタは問題のあるイベントのオフセットをログに記録し、次のイベントの処理を続行します。</p> <p>skip - コネクタは問題のあるイベントをスキップし、次のイベントの処理を続行します。</p>

プロパティ	デフォルト	説明
<code>poll.interval.ms</code>	500	コネクターがイベントのバッチの処理を開始する前に、新しい変更イベントの発生を待つ期間をミリ秒単位で指定する正の整数値。デフォルトは 500 ミリ秒 (0.5 秒) です。
<code>max.batch.size</code>	2048	コネクターが処理す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。
<code>max.queue.size</code>	8192	ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクターが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクターがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。 max.queue.size の値を、 max.batch.size の値よりも大きくなるように設定します。
<code>max.queue.size.in.bytes</code>	0	ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。 max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、 max.queue.size=1000 、 max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。

プロパティ	デフォルト	説明
heartbeat.interval.ms	0	<p>コネクターがハートビートメッセージを Kafka トピックに送信する頻度を制御します。デフォルトの動作では、コネクターはハートビートメッセージを送信しません。</p> <p>ハートビートメッセージは、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のに便利です。ハートビートメッセージは、コネクターの再起動時に再送信する必要がある変更イベントの数を減らすのに役立つ可能性があります。ハートビートメッセージを送信するには、このプロパティを、ハートビートメッセージの間隔をミリ秒単位で示す正の整数に設定します。</p> <p>ハートビートメッセージは、追跡されているデータベースには多くの更新があるにも関わらず、キャプチャーモードのテーブルにある更新はわずかである場合に便利です。この場合、コネクターは通常どおりにデータベーストランザクションログから読み取りしますが、変更レコードを Kafka に出力することはほとんどありません。そのため、コネクターが最新のオフセットを Kafka に送信することはほとんどありません。ハートビートメッセージを送信すると、コネクターは最新のオフセットを Kafka に送信できます。</p>
snapshot.delay.ms	デフォルトなし	<p>コネクターの起動時にスナップショットを実行するまでコネクターが待つ必要がある間隔 (ミリ秒単位)。クラスターで複数のコネクターを起動する場合、このプロパティは、コネクターのリバランスが行われる原因となるスナップショットの中断を防ぐのに役立ちます。</p>

プロパティ	デフォルト	説明
snapshot.include.collecton.list	<code>table.include.list</code> に指定したすべてのテーブル	<p>スナップショットに含めるテーブルの完全修飾名 (<schemaName>.<tableName>) と一致する正規表現のコンマ区切りリスト (オプション) です。指定する項目は、コネクターの table.include.list プロパティで名前を付ける必要があります。このプロパティは、コネクターの snapshot.mode プロパティが never 以外の値に設定されている場合にのみ有効です。</p> <p>このプロパティは増分スナップショットの動作には影響しません。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p>
snapshot.fetch.size	2000	スナップショットの実行中、コネクターは行のバッチでテーブルの内容を読み取ります。このプロパティは、バッチの行の最大数を指定します。
snapshot.lock.timeout.ms	10000	<p>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。コネクターがこの間隔でテーブルロックを取得できないと、スナップショットは失敗します。詳細は、コネクターによるスナップショットの実行方法 を参照してください。その他の可能な設定は次のとおりです。</p> <p>0 - ロックを取得できないとすぐに失敗します。</p> <p>-1 - コネクターは永久に待機します。</p>

プロパティ	デフォルト	説明
<p>snapshot.select.statement.overrides</p>	<p>デフォルトなし</p>	<p>スナップショットに追加するテーブル行を指定します。スナップショットにテーブルの行のサブセットのみを含める場合は、プロパティを使用します。このプロパティはスナップショットにのみ影響します。コネクタがログから読み取るイベントには影響しません。</p> <p>プロパティには、<schemaName>、<tableName> の形式で完全修飾テーブル名のコンマ区切りリストが含まれます。たとえば、</p> <pre>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</pre> <p>をリスト内の各テーブルに対して、スナップショットを作成する場合には、その他の設定プロパティを追加して、コネクタがテーブルで実行するように SELECT ステートメントを指定します。指定した SELECT ステートメントは、スナップショットに追加するテーブル行のサブセットを決定します。以下の形式を使用して、この SELECT ステートメントプロパティの名前 (</p> <pre>snapshot.select.statement.overrides.<schemaName>.<tableName></pre> <p>を指定します。例:</p> <pre>snapshot.select.statement.overrides.customers.orders.</pre> <p>例:</p> <p>スナップショットにソフト削除以外のレコードのみを含める場合は、soft-delete 列 (delete_flag) を含む customers.orders テーブルから、以下のプロパティを追加します。</p> <pre>"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre> <p>作成されるスナップショットでは、コネクタには delete_flag = 0 のレコードのみが含まれます。</p>

プロパティ	デフォルト	説明
sanitize.field.names	コネクターが key.converter または value.converter プロパティを Avro コンバーターに設定する場合は true に設定します。 そうでない場合は false に設定します。	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を示します。
provide.transaction.meta data	false	コネクターがトランザクション境界でイベントを生成し、トランザクションメタデータで変更イベントエンベロープを強化するかどうかを決定します。コネクターにこれを実行させる場合は true を指定します。詳細は、 トランザクションメタデータ を参照してください。
skipped.operations	t	ストリーミング中にスキップされる oplog 操作のコンマ区切りリスト。挿入/作成は c 、更新は u 、削除は d 、切り捨ては t 、操作をスキップしない場合は none となります。デフォルトでは、省略操作はスキップされます (このコネクターによる出力ではない)。
signal.data.collection	デフォルトなし	シグナルをコネクターへの送信に使用されるデータコレクションの完全修飾名。コレクション名の指定には <schemaName>.<tableName> の形式を使用します。
incremental.snapshot.chunk.size	1024	増分スナップショットのチャンクの実行中にコネクターがメモリーを取得して読み取る行の最大数。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。
topic.naming.strategy	io.debezium.schema.SchemaTopicNamingStrategy	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する TopicNamingStrategy クラスの名前。デフォルトは SchemaTopicNamingStrategy 。
topic.delimiter	.	トピック名の区切り文字を指定します。デフォルトは . です。

プロパティ	デフォルト	説明
topic.cache.size	10000	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。
topic.heartbeat.prefix	<code>__debezium-heartbeat</code>	コネクタがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は <code>__debezium-heartbeat.fulfillment</code> になります。
topic.transaction	<code>transaction</code>	コネクタがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は <code>fulfillment.transaction</code> になります。

Debezium コネクタデータベーススキーマ履歴設定プロパティ

Debezium には、コネクタがスキーマ履歴トピックと対話する方法を制御する `schema.history.internal.*` プロパティのセットが含まれています。

以下の表は、Debezium コネクタを設定するための `schema.history.internal` プロパティについて説明しています。

表3.17 コネクタデータベーススキーマ履歴の設定プロパティ

プロパティ	デフォルト	説明
schema.history.internal.kafka.topic	デフォルトなし	コネクタ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。
schema.history.internal.kafka.bootstrap.servers	デフォルトなし	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。各ペアは、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	デフォルト	説明
<code>schema.history.internal.kafka.recovery.poll.interval.ms</code>	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。
<code>schema.history.internal.kafka.query.timeout.ms</code>	3000	Kafka 管理クライアントを使用してクラスター情報を取得する際に、コネクターが待機すべき最大ミリ秒数を指定する整数値です。
<code>schema.history.internal.kafka.create.timeout.ms</code>	30000	Kafka 管理クライアントを使用して kafka 履歴トピックを作成する間、コネクターが待機する最大ミリ秒数を指定する整数値。
<code>schema.history.internal.kafka.recovery.attempts</code>	100	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 × recovery.poll.interval.ms です。
<code>schema.history.internal.skip.unparseable.ddl</code>	false	コネクターが不正または不明なデータベースのステートメントを無視するかどうか、または人が問題を修正するために処理を停止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 の処理中にデータの損失や分割を引き起こす可能性があるため、必ず注意して使用する必要があります。
<code>schema.history.internal.store.only.captured.tables.ddl</code>	false	コネクターがすべての DDL ステートメントを記録するかどうかを指定するブール値 true は、変更が Debezium によってキャプチャーされるテーブルに関連する DDL ステートメントのみを記録します。変更がキャプチャーされたテーブルを変更すると、不足しているデータが必要になる可能性があるため、 true に設定するには注意が必要です。 安全なデフォルトは false です。

プロデューサーおよびコンシューマクライアントを設定するためのパススルーデータベーススキーマ履歴プロパティ

Debezium は、Kafka プロデューサーを使用して、データベーススキーマ履歴トピックにスキーマの変更を書き込みます。同様に、コネクターが起動すると、データベーススキーマ履歴トピックから読み取る Kafka コンシューマーに依存します。 **schema.history.internal.producer.*** および **schema.history.internal.consumer.*** 接頭辞で始まるパススルー設定プロパティのセットに値を割り当てて、Kafka プロデューサーおよびコンシューマクライアントの設定を定義します。パススルー

ロデューサーおよびコンシューマーデータベーススキーマ履歴プロパティは、以下の例のように Kafka ブローカーとのこれらのクライアントの接続をセキュアにする方法など、さまざまな動作を制御します。

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

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

Debezium は、プロパティを Kafka クライアントに渡す前に、プロパティ名から接頭辞を削除します。

[Kafka プロデューサー設定プロパティ](#) および [Kafka コンシューマー設定プロパティ](#) の詳細は、Kafka のドキュメントを参照してください。

Debezium コネクターのパススルーデータベースドライバー設定プロパティ

Debezium コネクターでは、データベースドライバーのパススルー設定が可能です。パススルーデータベースプロパティは接頭辞 **driver.*** で始まります。たとえば、コネクターは **driver.foofoo=false** などのプロパティを JDBC URL に渡します。

[データベーススキーマ履歴クライアントのパススループロパティ](#) の場合のように、Debezium はプロパティから接頭辞を削除してからデータベースドライバーに渡します。

3.7. DEBEZIUM DB2 コネクターのパフォーマンスの監視

Debezium Db2 コネクターは、Apache Zookeeper、Apache Kafka、および Kafka Connect によって提供される JMX メトリクスの組み込みサポートに加えて、3 種類のメトリクスを提供します。

- [スナップショットメトリクス](#) は、スナップショットの実行中にコネクター操作に関する情報を提供します。
- [メトリクスのストリーミング](#) は、コネクターが変更をキャプチャーし、変更イベントレコードをストリーミングする際のコネクター操作に関する情報を提供します。
- [スキーマ履歴メトリクス](#) は、コネクターのスキーマ履歴の状態に関する情報を提供します。

[Debezium モニタリングのドキュメント](#) では、JMX を使用してこれらのメトリクスを公開する方法の詳細を説明しています。

3.7.1. Db2 データベースのスナップショット作成時の Debezium の監視

MBean は **debezium.db2:type=connector-metrics,context=snapshot,server=<db2.server.name>** です。

スナップショット操作がアクティブでない場合や、最後のコネクターの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	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 include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	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。
QueueRemainingCapacity	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。
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
SnapshotAborted	boolean	スナップショットが中断されたかどうか。

属性	タイプ	説明
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

コネクタは、増分スナップショットの実行時に、以下の追加のスナップショットメトリクスも提供します。

属性	タイプ	説明
ChunkId	string	現在のスナップショットチャンクの識別子。
ChunkFrom	string	現在のチャンクを定義するプライマリーキーセットの下限。

属性	タイプ	説明
ChunkTo	string	現在のチャンクを定義するプライマリーキーセットの上限。
TableFrom	string	現在スナップショットされているテーブルのプライマリーキーセットの下限。
TableTo	string	現在スナップショットされているテーブルのプライマリーキーセットの上限。

3.7.2. Debezium Db2 コネクターレコードストリーミングの監視

MBean は `debezium.db2:type=connector-metrics,context=streaming,server=<db2.server.name>` です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。
TotalNumberOfCreateEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクターが見た更新イベントの合計数。

属性	タイプ	説明
TotalNumberOfDeleteEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルタリングルールによってフィルタされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。
Connected	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)。この値は、データベースサーバーとコネクターが稼働しているマシンのクロック間の差異に対応します。
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。

属性	タイプ	説明
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

3.7.3. Debezium Db2 コネクターのスキーマ履歴の監視

MBean は `debezium.db2:type=connector-metrics,context=schema-history,server=<db2.server.name>` です。

以下の表は、利用可能なスキーマ履歴メトリクスの一覧です。

属性	タイプ	説明
Status	string	データベーススキーマ履歴の状態を示す STOPPED 、 RECOVERING (ストレージから履歴を復元)、または 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。

3.8. DEBEZIUM DB2 コネクタの管理

Debezium Db2 コネクタをデプロイしたら、Debezium 管理 UDF を使用して、SQL コマンドで Db2 レプリケーション (ASN) を制御します。UDF によっては戻り値が必要な場合があります。この場合、SQL の **VALUE** ステートメントを使用して呼び出します。その他の UDF には、SQL の **CALL** ステートメントを使用します。

表3.18 Debezium 管理 UDF の説明

タスク	コマンドおよび注記
ASN エージェントを起動する	VALUES ASNCDC.ASNCDCSERVICES('start','asncdc');
ASN エージェントを停止する	VALUES ASNCDC.ASNCDCSERVICES('stop','asncdc');
Check ASN エージェントのステータスを確認する	VALUES ASNCDC.ASNCDCSERVICES('status','asncdc');
テーブルをキャプチャーモードにする	CALL ASNCDC.ADDTABLE('MYSCHEMA', 'MYTABLE'); MYSCHEMA は、キャプチャーモードにするテーブルが含まれるスキーマの名前に置き換えます。同様に、 MYTABLE は、キャプチャーモードにするテーブルの名前に置き換えます。
テーブルのキャプチャーモードを解除する	CALL ASNCDC.REMOVETABLE('MYSCHEMA', 'MYTABLE');
ASN サービスを再度初期化する	VALUES ASNCDC.ASNCDCSERVICES('reinit','asncdc'); テーブルをキャプチャーモードにした後か、キャプチャーモードからテーブルを削除した後に、これを行います。

3.9. DEBEZIUM コネクタでのキャプチャーモードの DB2 テーブルのスキーマの更新

Debezium Db2 コネクタはスキーマ変更をキャプチャーできますが、スキーマを更新するには、データベース管理者と協力してコネクタが変更イベントの生成を継続するようにする必要があります。これは、Db2 がレプリケーションを実装する方法に必要です。

Db2 のレプリケーション機能は、キャプチャーモードのテーブルごとに、すべての変更が含まれる変更データテーブルをそのソーステーブルに作成します。ただし、変更データテーブルスキーマは静的です。キャプチャーモードのテーブルのスキーマを更新する場合は、対応する変更データテーブルのスキーマを更新する必要があります。Debezium Db2 コネクタはこれを実行できません。昇格された権限を持つデータベース管理者は、キャプチャーモードのテーブルのスキーマを更新する必要があります。



警告

同じテーブルの新しいスキーマ更新の前に、スキーマ更新の手順を完全に実行することが重要です。そのため、スキーマ更新の手順を1度で完了するために、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。

通常、テーブルスキーマを更新する手順は2つあります。

- オフライン - Debezium の停止中に実行されます。
- オンライン - Debezium の稼働中に実行されます。

それぞれの方法に長所と短所があります。

3.9.1. Debezium Db2 コネクターでのオフラインスキーマ更新の実行

オフラインでスキーマの更新を行う前に、Debezium Db2 コネクターを停止します。これはより安全なスキーマ更新の手順ですが、高可用性の要件のあるアプリケーションには実現できない可能性があります。

前提条件

- スキーマの更新が必要なキャプチャーモードのテーブル1つ以上。

手順

1. データベースを更新するアプリケーションを一時停止します。
2. Debezium コネクターがストリーミングされていない変更イベントレコードをすべてストリーミングするまで待ちます。
3. Debezium コネクターを停止します。
4. すべての変更をソーステーブルスキーマに適用します。
5. ASN レジスターテーブルで、スキーマが更新されたテーブルを **INACTIVE** でマーク付けします。
6. ASN キャプチャーサービスを再初期化します。
7. キャプチャーモードからテーブルを削除するために Debezium UDF を実行して、キャプチャーモードから古いスキーマを含まれるソーステーブルを削除します。
8. テーブルをキャプチャーモードに追加するために Debezium UDF を実行して、スキーマが新しいソーステーブルをキャプチャーモードに追加します。
9. ASN レジスターテーブルで、更新されたソーステーブルを **ACTIVE** としてマーク付けします。
10. ASN キャプチャーサービスを再初期化します。
11. データベースを更新するアプリケーションを再開します。

12. Debezium コネクタを再起動します。

3.9.2. Debezium Db2 コネクタでのオンラインスキーマ更新の実行

オンラインスキーマの更新ではアプリケーションやデータ処理のダウンタイムは必要ありません。そのため、オンラインスキーマの更新を実行する前に Debezium Db2 コネクタを停止しません。また、オンラインスキーマの更新手順は、オフラインスキーマの更新手順よりも簡単です。

ただし、テーブルがキャプチャモードの場合は、列名の変更後も Db2 レプリケーション機能は引き続き古い列名を使用します。新しい列名は、Debezium の変更イベントでは表示されません。変更イベントにある新しい列名を確認するには、コネクタを再起動する必要があります。

前提条件

- スキーマの更新が必要なキャプチャモードのテーブル1つ以上。

テーブルの最後に列を追加する場合の手順

1. 変更するスキーマのソーステーブルをロックします。
2. ASN レジスターテーブルで、ロックされたテーブルを **INACTIVE** としてマーク付けします。
3. [ASN キャプチャサービスを再初期化します。](#)
4. ソーステーブルのスキーマにすべての変更を適用します。
5. 対応する変更データテーブルのスキーマにすべての変更を適用します。
6. ASN レジスターテーブルで、ソーステーブルを **ACTIVE** としてマーク付けします。
7. [ASN キャプチャサービスを再初期化します。](#)
8. 任意手順:コネクタを再起動して、変更イベントにある更新された列名を確認します。

テーブルの中に列を追加する場合の手順

1. 変更するソーステーブルをロックします。
2. ASN レジスターテーブルで、ロックされたテーブルを **INACTIVE** としてマーク付けします。
3. [ASN キャプチャサービスを再初期化します。](#)
4. 変更するソーステーブルごとに以下を行います。
 - a. ソーステーブルのデータをエクスポートします。
 - b. ソーステーブルを切り捨てます。
 - c. ソーステーブルを変更して列を追加します。
 - d. エクスポートしたデータを変更したソーステーブルに読み込みます。
 - e. ソーステーブルの対応する変更データテーブルのデータをエクスポートします。
 - f. 変更データテーブルを切り捨てます。
 - g. 変更データテーブルを変更して、列を追加します。

-
- h. エクスポートしたデータを変更した変更データテーブルに読み込みます。
 5. ASN レジスターテーブルで、テーブルを **INACTIVE** としてマーク付けします。これにより、古い変更データテーブルが非アクティブとしてマーク付けされるため、それらのテーブルにあるデータは保持されますが、更新されなくなります。
 6. [ASN キャプチャーサービスを再初期化](#)します。
 7. 任意手順:コネクターを再起動して、変更イベントにある更新された列名を確認します。

第4章 MONGODB の DEBEZIUM コネクタ

Debezium の MongoDB コネクタは、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。

このコネクタと互換性のある MongoDB のバージョンについては、[Debezium](#) でサポートされる設定ページを参照してください。

Debezium MongoDB コネクタを使用するための情報および手順は、以下のように設定されています。

- [「Debezium MongoDB コネクタの概要」](#)
- [「Debezium MongoDB コネクタの仕組み」](#)
- [「Debezium MongoDB コネクタのデータ変更イベントの説明」](#)
- [「Debezium コネクタと連携する MongoDB の設定」](#)
- [「Debezium MongoDB コネクタのデプロイメント」](#)
- [「Debezium MongoDB コネクタのパフォーマンスの監視」](#)
- [「Debezium MongoDB コネクタによる障害および問題の処理方法」](#)

4.1. DEBEZIUM MONGODB コネクタの概要

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。新規サーバーをレプリカセットに追加すると、そのサーバーは最初にプライマリーのすべてのデータベースおよびコレクションの **スナップショット** を実行し、次にプライマリーの oplog を読み取り、スナップショットの開始後に加えられた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。

4.1.1. MongoDB コネクタが変更ストリームを使用してイベントレコードをキャプチャーする方法の説明

Debezium MongoDB コネクタはレプリカセットの一部ではありませんが、同様のレプリケーションメカニズムを使用して oplog データを取得します。主な違いは、コネクタが直接 oplog を読み取らない点です。代わりに、oplog データの取得およびデコードを MongoDB **変更ストリーム** 機能に委譲します。変更ストリームでは、MongoDB サーバーはコレクションで発生する変更をイベントストリームとして公開します。Debezium コネクタはストリームを監視し、変更をダウンストリームに配信します。コネクタが最初にレプリカセットを検出すると、oplog を調べ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スナップショットを作成します。コネクターがデータのコピーを終了すると、以前に読み取られた oplog の位置から開始する変更ストリームを作成します。

MongoDB コネクターは変更を処理すると、イベントを発信する先の oplog/stream の位置を定期的に記録します。コネクターが停止したら、最後に処理した oplog ストリームの位置を記録するため、再起動後にその位置からストリーミングを再開できます。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、常に停止した場所を正確に特定します。当然ながら、MongoDB oplogs は通常最大サイズに制限されているため、コネクターがそれらを読み取る前に oplog の操作がパージされる可能性があります。この場合、再起動後、足りていない oplog 操作をコネクターが検出し、スナップショットを実行してから、変更をストリームします。

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変更をストリーミングします。そのため、レプリカセットの選出が行われ、他のノードがプライマリーになると、コネクターはすぐ変更のストリーミングを停止し、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変更のストリーミングを開始します。同様に、プライマリーとして設定されたレプリカとコネクターが通信できない場合は、再接続を試みます (ネットワークまたはレプリカセットを圧迫しないように指数バックオフを使用します)。接続が再確立された後、コネクターはキャプチャーした最後のイベントからの変更を引き続きストリーミングします。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し、通信障害を自動的に処理します。

関連情報

- [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](#)
- [Change Streams](#)

4.2. DEBEZIUM MONGODB コネクターの仕組み

コネクターがサポートする MongoDB トポロジーの概要は、アプリケーションを計画するときに役立ちます。

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。

Debezium MongoDB コネクターの仕組みの詳細は、以下を参照してください。

- [「Debezium コネクターでサポートされる MongoDB トポロジー」](#)
- [「Debezium MongoDB コネクターでレプリカセットおよびシャードクラスターに論理名を使用する方法」](#)
- [「Debezium MongoDB コネクターでのスナップショットの実行方法」](#)
- [「Debezium MongoDB コネクターでの変更イベントレコードのストリーミング方法」](#)
- [「Debezium MongoDB 変更イベントレコードを受信する Kafka トピックのデフォルト名」](#)
- [「イベントキーが Debezium MongoDB コネクターのトピックパーティション設定を制御する方法」](#)
- [「トランザクション境界を表す Debezium MongoDB コネクターによって生成されたイベント」](#)

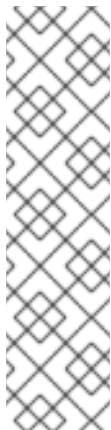
4.2.1. Debezium コネクターでサポートされる MongoDB トポロジー

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

MongoDB レプリカセット

Debezium 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を有効なメンバーから除外し、プロキシ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指示し、代わりに最初のシードアドレス (**mongodb.hosts** プロパティによって指定) 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。

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。

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。詳細は [MariaDB のドキュメント](#) を参照してください。

4.2.2. Debezium MongoDB コネクターでレプリカセットおよびシャードクラスターに論理名を使用する方法

コネクター設定プロパティ **topic.prefix** 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。すべてのトピック名の接頭辞として、各レプリカセットの変更ストリームの位置を記録する際に一意の識別子として使用されます。

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。

4.2.3. Debezium MongoDB コネクターでのスナップショットの実行方法

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、コネクターが変更の読み取りを停止した位置を示す **オフセット** を検出します。オフセットが検出され、oplog に存在する場合、タスクは記録されたオフセットの位置から即座に **ストリームの変更** 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 **スナップショット** 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て開始され、オフセット (スナップショット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 (**snapshot.max.threads** 設定プロパティの値まで)、この作業を並行して行います。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、スナップショット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。

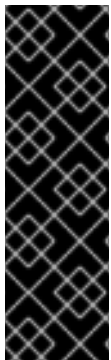
このスナップショットは、コネクターのフィルターと一致するすべてのコレクションがコピーされるまで継続されます。タスクのスナップショットが完了する前にコネクターが停止した場合は、コネクターを再起動すると、再びスナップショットを開始します。



注記

コネクターがレプリカセットのスナップショットを実行している間、タスクの再割り当てと再設定を回避します。コネクターは、スナップショットの進捗を報告するログメッセージを生成します。最大限の制御を行うために、コネクターごとに個別の Kafka Connect クラスタを実行します。

4.2.3.1. アドホックスナップショット



重要

アドホックスナップショットは、Debezium MongoDB コネクターのテクノロジープレビュー機能です。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の機能は、最新の技術をいち早く提供して、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。

デフォルトでは、コネクターは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクターではスナップショットプロセスは繰り返し処理されません。コネクターがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクターを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。収集データを再キャプチャーするメカニズムを提供するために、Debezium はアドホックスナップショットを実行するオプションを備えています。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクター設定が変更され、異なるコレクションのセットをキャプチャーします。
- Kafka トピックを削除して、再構築する必要があります。
- 設定エラーや他の問題が原因で、データの破損が発生します。

いわゆる **アドホックスナップショット** を開始することで、以前にスナップショットをキャプチャーしたコレクションに対してスナップショットを再実行することができます。アドホックスナップショットでは、[シグナリングコレクション](#) を使用する必要があります。シグナルリクエストを Debezium シグナルコレクションに送信して、アドホックスナップショットを開始します。

既存のコレクションのアドホックスナップショットを開始すると、コネクターはコレクションにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、[トピックの自動作成](#) が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックスナップショットシグナルは、スナップショットに追加するコレクションを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のコレクション

のサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のコレクションの内容のサブセットをキャプチャーできます。

キャプチャするコレクションは、シグナリングコレクションに **execute-snapshot** メッセージを送信することで指定します。**execute-snapshot** シグナルのタイプを **incremental** に設定し、スナップショットに含めるコレクション名を次の表に示すように指定します。

表4.1 アドホックの **execute-snapshot** シグナルレコードの例

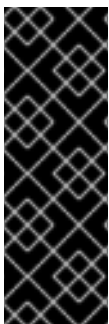
フィールド	デフォルト	値
type	incremental	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 incremental スナップショットのみです。
data-collections	該当なし	スナップショットされるコレクションの完全修飾名にマッチする正規表現を含む配列。名前の形式は signal.data.collection 設定オプションと同じです。
additional-condition	該当なし	コレクションの内容のサブセットを取得するために、コレクションの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

execute-snapshot シグナルタイプのエントリーをシグナルコレクションに追加して、アドホックスナップショットを開始します。コネクターがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各コレクションの開始ポイントおよびエンドポイントとして使用します。コレクションのエントリー数と設定されたチャンクサイズに基づいて、Debezium はコレクションをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、**execute-snapshot** アクションタイプは **増分スナップショット** のみをトリガーします。詳細は、[スナップショットの増分](#) を参照してください。

4.2.3.2. 増分スナップショット



重要

増分スナップショットは、Debezium MongoDB コネクターのテクノロジープレビュー機能です。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の機能は、最新の技術をいち早く提供して、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。

スナップショットを柔軟に管理するため、Debezium には **増分スナップショット** と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、[Debezium コネクターにシグナルを送信するための](#) Debezium メカニズムに依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各コレクションを段階的にキャプチャー

します。スナップショットがキャプチャーするコレクションと、[各チャンクのサイズ](#)を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

Debezium は、増分スナップショットの進行に伴い、その進捗を追跡するために透かしを使用し、キャプチャした各コレクション行の記録を保持します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクターはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセス再開後、スナップショットは、最初からコレクションを再キャプチャーするのではなく、停止したポイントから開始されます。
- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。例えば、コネクターの設定を変更してコレクションをその [collection.include.list](#) プロパティにコレクションを追加します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各コレクションをプライマリーキー別に分類して、[設定されたチャンクサイズ](#)に基づいてコレクションをチャンクに分割します。チャンクごとに作業し、チャンク内の各コレクション行をキャプチャします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットが進むと、他のプロセスがデータベースにアクセスし続け、コレクションのレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT**、**UPDATE**、**DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できます。つまり、スナップショットがその行の **READ** イベントを含むチャンクをキャプチャーする前に、ストリーミングプロセスがコレクション行を変更するイベントを発行する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリーミングされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて到着した **READ** イベントと、同じコレクション行を変更するストリームイベント間の衝突を解決するために、Debezium はいわゆる **スナップショットウィンドウ** を採用しています。スナップショットウィンドウは、増分スナップショットが指定されたコレクションチャンクのデータをキャプチャーする間隔を区切ります。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は2種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャーするスナップショットレコード

は、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをコレクションの Kafka トピックに発行します。

コネクターは各スナップショットチャンクにプロセスを繰り返します。



警告

増分スナップショットでは、プライマリーキーが安定した順序で並べる必要があります。ただし、**文字列** はエンコーディングと特殊文字が予期しない動作につながる可能性があるため、順序が安定しない場合があります (**Mongo sort String**)。増分スナップショットを実行する場合は、プライマリーキーに他のタイプを使用することを検討してください。



注記

増分スナップショットは、現在、単一のレプリカセットのデプロイメントに対してのみサポートされています。

4.2.3.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、ソースのデータベース上のシグナリングコレクションに **アドホックなスナップショットシグナル** を送信することです。

MongoDB の **insert()** メソッドを使用して、シグナルコレクションにシグナルを送信します。

Debezium は、信号コレクションの変化を検出した後、信号を読み取り、要求されたスナップショット操作を実行します。

送信するクエリーは、スナップショットに含めるコレクションを指定し、オプションでスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに含めるテーブルを指定するには、テーブルをリストアップした **data-collections** 配列、またはテーブルのマッチングに使用する正規表現の配列を指定します。たとえば、**{"data-collections": ["public.Collection1", "public.Collection2"]}**

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるコレクションの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのコレクションを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **MyCollection** のテーブルを含めるには、**"public"."MyCollection"** の形式を使用します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. シグナリングコレクションにスナップショットシグナルドキュメントを挿入します。

```
<signalDataCollection>.insert({"id" : <idNumber>,"type" : <snapshotType>, "data" : {"data-collections" ["<collectionName>", "<collectionName>"],"type": <snapshotType>}});
```

以下に例を示します。

```
db.debeziumSignal.insert({
  "type" : "execute-snapshot",
  "data" : {
    "data-collections" ["\"public\".\"Collection1\"", "\"public\".\"Collection2\""],
    "type": "incremental"
  }
});
```

コマンドの **id**、**type**、および **data** パラメーターの値は、シグナリングコレクションのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表4.2 シグナリングコレクションに増分スナップショットシグナルを送信するための MongoDB insert() コマンドのフィールドの説明

項目	値	説明
1	db.debeziumSignal	ソースデータベース上のシグナリングコレクションの完全修飾名を指定します。

項目	値	説明
2	null	<p>id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。</p> <p>前述の例の insert メソッドは、オプションの _id パラメーターの使用が省略されています。ドキュメントはパラメーターの値を明示的に割り当てないため、MongoDB が自動的にドキュメントに割り当てる任意の ID がシグナルリクエストの ID 識別子になります。</p> <p>この文字列を使用して、シグナリングコレクションのエントリーにロギングメッセージを識別します。Debeziumはこの識別子文字列を使用しません。代わりに、スナップショット作成中に、Debeziumは独自の ID 文字列をウォーターマークシグナルとして生成します。</p>
3	execute-snapshot	<p>type パラメーターを指定し、シグナルがトリガーする操作を指定します。</p>
4	data-collections	<p>シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるコレクション名の配列またはコレクション名と一致する正規表現を指定します。</p> <p>この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。signal.data.collection 設定プロパティーでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。</p>
5	incremental	<p>実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。</p> <p>現在、唯一の有効なオプションはデフォルト値 incremental だけです。</p> <p>値を指定しない場合には、コネクターは増分スナップショットを実行します。</p>

以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 シグナルコレクションに送信する SQL クエリーに type 値を指定します。 値を指定しない場合には、コネクタは増分スナップショットを実行します。
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

4.2.3.4. 増分スナップショットの停止

ソースデータベースのコレクションにシグナルを送信することで、増分スナップショットを停止することもできます。シグナリングコレクションにドキュメントを挿入して、スナップショット停止のシグナルを送信します。Debezium はシグナルコレクションの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのコレクションを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

- シグナリングコレクションにスナップショット停止のシグナルドキュメントを挿入します。

```
<signalDataCollection>.insert({"id" : <idNumber>, "type" : "stop-snapshot", "data" : {"data-collections" ["<collectionName>", "<collectionName>"], "type": "incremental"}});
```

以下に例を示します。

```
db.debeziumSignal.insert({
  "type" : "stop-snapshot",
  "data" : {
    "data-collections" ["\"public\".\"Collection1\"","\"public\".\"Collection2\""],
    "type": "incremental"
  }
});
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナリングコレクションのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表4.3 シグナリングコレクションに増分スナップショットの停止ドキュメントを送信するための insert コマンドのフィールドの説明

項目	値	説明
1	db.debeziumSignal	ソースデータベース上のシグナリングコレクションの完全修飾名を指定します。
2	null	前述の例の insert メソッドは、オプションの _id パラメーターの使用が省略されています。ドキュメントはパラメーターの値を明示的に割り当てないため、MongoDB が自動的にドキュメントに割り当てる任意の ID がシグナルリクエストの ID 識別子になります。 この文字列を使用して、シグナリングコレクションのエントリにロギングメッセージを識別します。Debezium はこの識別子文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの任意コンポーネントで、スナップショットから削除するコレクション名の配列またはコレクション名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

4.2.4. Debezium MongoDB コネクターでの変更イベントレコードのストリーミング方法

レプリカセットレコードのコネクタータスクがオフセットを取得すると、オフセットを使用して変更のストリーミングを開始する oplog の位置を判断します。その後、タスクは (設定によって) レプリカセットのプライマリーノードに接続するか、レプリカセット全体の変更ストリームに接続し、その位置から変更のストリーミングを開始します。すべての作成、挿入、および削除操作を処理して Debezium の **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、Kafka Connect ワーカー設定プロパティである **offset.flush.interval.ms** 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

Kafka パイプライン内のすべてのコンポーネントが正常に動作している場合、Kafka コンシューマーはすべてのメッセージを **1度だけ** 受信します。ただし、問題が発生した場合、Kafka はコンシューマーが **少なくとも1回** のみすべてのメッセージを受信することを保証できます。予期しない結果を回避するには、コンシューマーは重複メッセージを処理する必要があります。

前述のように、コネクタタスクは常にレプリカセットのプライマリノードを使用して oplog からの変更をストリーミングし、コネクタが可能な限り最新の操作を確認できるようにし、代わりにセカンダリーが使用された場合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を選出すると、コネクタは即座に変更のストリーミングを停止し、新しいプライマリに接続して、同じ場所にある新しいプライマリノードから変更のストリーミングを開始します。同様に、コネクタ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、レプリカセットが過剰にならないように指数バックオフを使用して再接続を試みます。接続の確立後、停止した場所から変更のストリーミングを続行します。これにより、コネクタ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、通信障害を自動的に処理できます。

要約すると、MongoDB コネクタはほとんどの状況で実行を継続します。通信の問題により、問題が解決されるまでコネクタが待機する可能性があります。

4.2.5. Debezium 変更イベントの **before** フィールドに入力するための MongoDB サポート

MongoDB 6.0 以降では、変更ストリームを設定して、ドキュメントのイメージ前の状態を出力し、MongoDB 変更イベントの **before** フィールドにデータを投入できます。MongoDB で事前のイメージを使用できるようにするには、**db.createCollection()**、**create**、または **collMod** を使用して、コレクションの **changeStreamPreAndPostImages** を設定する必要があります。Debezium MongoDB が変更イベントに事前イメージを追加できるようにするには、コネクタの **capture.mode** を ***_with_pre_image** オプションのいずれかに設定します。



MONGODB 変更イベントのサイズ制限

MongoDB 変更イベントのサイズは 16 メガバイトに制限されます。したがって、事前イメージを使用すると、このしきい値を超過し、障害が発生する可能性があります。変更ストリームの制限を超えないようにする方法は、[MongoDB のドキュメント](#) を参照してください。

4.2.6. Debezium MongoDB 変更イベントレコードを受信する Kafka トピックのデフォルト名

MongoDB コネクタは、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、**topic.prefix** 設定プロパティで指定されるコネクタの **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。

4.2.7. イベントキーが Debezium MongoDB コネクターのトピックパーティション設定を制御する方法

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 はイベントキーに基づいてトピックのパーティションを作成する方法を決定できます。Kafka Connect ワーカー設定に **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み、合計順序を維持します。キーでイベントのパーティションを行うと、同じキーを持つすべてのイベントは常に同じパーティションに移動します。これにより、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されます。

4.2.8. トランザクション境界を表す Debezium MongoDB コネクターによって生成されたイベント

Debezium は、トランザクションメタデータ境界を表すイベントを生成でき、データイベントメッセージを補完できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクターのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクターをデプロイする前に発生するトランザクションのメタデータは利用できません。

Debezium はすべてのトランザクションの **BEGIN** および **END** に対して、以下のフィールドが含まれるイベントを生成します。

status

BEGIN または **END**

id

一意のトランザクション識別子の文字列表現。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

指定のデータコレクションからの変更によって出力されたイベントの数を提供する **data_collection** と **event_count** のペアの配列。

以下の例では、一般的なメッセージを示します。

```
{
  "status": "BEGIN",
```

```

    "id": "1462833718356672513",
    "event_count": null,
    "data_collections": null
  }

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

```

topic.transaction オプションで上書きされない限り、トランザクションイベントは **<topic.prefix>.transaction** という名前のトピックに書き込まれます。

変更データイベントのエンリッチメント

トランザクションメタデータを有効にすると、データメッセージ **Envelope** は新しい **transaction** フィールドでエンリッチされます。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下は、メッセージの内容の例です。

```

{
  "after": {"_id": {"$numberLong": "1004"}, "first_name": "Anne", "last_name": "Kretchmar", "email": "annek@noanswer.org"},
  "source": {
    ...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "1462833718356672513",
    "total_order": "1",
    "data_collection_order": "1"
  }
}

```

4.3. DEBEZIUM MONGODB コネクタのデータ変更イベントの説明

Debezium MongoDB コネクタは、データを挿入、更新、または削除する各ドキュメントレベルの操作に対してデータ変更イベントを生成します。各イベントにはキーと値が含まれます。キーと値の構造は、変更されたコレクションによって異なります。

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** を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があります。コンシューマーによる処理が困難になることがあります。これに対応するために、各イベントにはコンテンツのスキーマが含まれます。スキーマレジストリーを使用している場合は、コンシューマーがレジストリーからスキーマを取得するために使用できるスキーマ ID が含まれます。これにより、各イベントが自己完結型になります。

以下のスケルトン JSON は、変更イベントの基本となる 4 つの部分を示しています。ただし、アプリケーションで使用するために選択した Kafka Connect コンバータの設定方法によって、変更イベントのこれら 4 部分の表現が決定されます。**schema** フィールドは、変更イベントが生成されるようにコンバータを設定した場合のみ変更イベントに含まれます。同様に、イベントキーおよびイベントペイロードは、変更イベントが生成されるようにコンバータを設定した場合のみ変更イベントに含まれます。JSON コンバータを使用し、変更イベントの基本となる 4 つの部分すべてを生成するように設定すると、変更イベントの構造は次のようになります。

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

表4.4 変更イベントの基本内容の概要

項目	フィールド名	説明
1	schema	最初の schema フィールドはイベントキーの一部です。イベントキーの payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、最初の schema フィールドには、変更されたドキュメントのキーの構造を記述されます。
2	payload	最初の payload フィールドはイベントキーの一部です。前述の schema フィールドによって記述された構造を持ち、変更されたドキュメントのキーが含まれます。
3	schema	2 つ目の schema フィールドはイベント値の一部です。イベント値の payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、2 つ目の schema は変更されたドキュメントの構造を記述します。通常、このスキーマには入れ子になったスキーマが含まれます。

項目	フィールド名	説明
4	payload	2つ目の payload フィールドはイベント値の一部です。前述の schema フィールドによって記述された構造を持ち、変更されたドキュメントの実際のデータが含まれます。

デフォルトでは、コネクタによって、変更イベントレコードがイベントの元のコレクションと同じ名前を持つトピックにストリーミングされます。[トピック名](#) を参照してください。



警告

MongoDB コネクタは、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。つまり、論理サーバー名はアルファベットまたはアンダースコア (a-z、A-Z、または `_`) で始まる必要があります。論理サーバー名の残りの各文字と、データベース名とコレクション名の各文字は、アルファベット、数字、またはアンダースコア (a-z、A-Z、0-9、または `_`) でなければなりません。無効な文字がある場合は、アンダースコアに置き換えられます。

論理サーバー名、データベース名、またはコレクション名に無効な文字が含まれ、名前を区別する唯一の文字が無効であると、無効な文字はすべてアンダースコアに置き換えられるため、予期せぬ競合が発生する可能性があります。

詳細は、以下のトピックを参照してください。

- [「Debezium MongoDB 変更イベントのキー」](#)
- [「Debezium MongoDB 変更イベントの値」](#)

4.3.1. Debezium MongoDB 変更イベントのキー

変更イベントのキーには、変更されたドキュメントのキーのスキーマと、変更されたドキュメントの実際のキーのスキーマが含まれます。特定のコレクションでは、スキーマとそれに対応するペイロードの両方に単一の **id** フィールドが含まれます。このフィールドの値は、[MongoDB Extended JSON のシリアライゼーションの厳格モード](#) から派生する文字列として表されるドキュメントの識別子です。

論理名が **fulfillment** のコネクタ、**inventory** データベースが含まれるレプリカセット、および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。

ドキュメントの例

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

変更イベントキーの例

customers コレクションへの変更をキャプチャーする変更イベントのすべてに、イベントキースキーマがあります。**customers** コレクションに前述の定義がある限り、**customers** コレクションへの変更をキャプチャーする変更イベントのキー構造はすべて以下ようになります。JSON では、以下のようになります。

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

表4.5 変更イベントキーの説明

項目	フィールド名	説明
1	schema	キーのスキーマ部分は、キーの payload 部分の内容を記述する Kafka Connect スキーマを指定します。
2	fulfillment.inventory.customers.Key	キーのペイロードの構造を定義するスキーマの名前。このスキーマは、変更したドキュメントのキーの構造を説明します。キースキーマ名の形式は connector-name.database-name.collection-name.Key です。この例では、以下のようになります。 <ul style="list-style-type: none"> ● fulfillment はこのイベントを生成したコネクターの名前です。 ● inventory は変更されたコレクションが含まれるデータベースです。 ● customers は更新されたドキュメントが含まれるコレクションです。
3	任意	イベントキーの payload フィールドに値が含まれる必要があるかどうかを示します。この例では、キーのペイロードに値が必要です。ドキュメントにキーがない場合、キーの payload フィールドの値は任意です。
4	fields	各フィールドの名前、型、および必要かどうかなど、 payload で想定される各フィールドを指定します。
5	payload	この変更イベントが生成されたドキュメントのキーが含まれます。この例では、キーには型 string の1つの id フィールドが含まれ、その値は 1004 です。

この例では、整数の識別子を持つドキュメントを使用しますが、有効な MongoDB ドキュメント識別子は、ドキュメント識別子を含め、同じように動作します。ドキュメント識別子の場合、イベントキーの **payload.id** 値は、厳格モードを使用する MongoDB Extended JSON シリアライゼーションとして更新されたドキュメントの元の **_id** フィールドを表す文字列です。以下の表では、さまざまな型の **_id** フィールドを表す例を示します。

表4.6 イベントキーペイロードのドキュメント **_id** フィールドを表す例

タイプ	MongoDB _id の値	キーのペイロード
Integer	1234	{ "id" : "1234" }
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\" }" }

4.3.2. Debezium MongoDB 変更イベントの値

変更イベントの値はキーよりも若干複雑です。キーと同様に、値には **schema** セクションと **payload** セクションがあります。**schema** セクションには、入れ子のフィールドを含む、**Envelope** セクションの **payload** 構造を記述するスキーマが含まれています。データを作成、更新、または削除する操作のすべての変更イベントには、Envelope 構造を持つ値 payload があります。

変更イベントキーの例を紹介するために使用した、同じサンプルドキュメントについて考えてみましょう。

ドキュメントの例

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

このドキュメントへの変更に対する変更イベントの値部分には、以下の各イベントタイプについて記述されています。

- [作成イベント](#)

- 更新イベント
- 削除イベント
- 廃棄 (tombstone) イベント

作成 イベント

以下の例は、**customers** コレクションにデータを作成する操作に対して、コネクタによって生成される変更イベントの値の部分を示しています。

```
{
  "schema": { ❶
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": true,
        "name": "io.debezium.data.Json", ❷
        "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", ❸
"field": "source"
},
{
  "type": "string",
  "optional": true,
  "field": "op"
},
{
  "type": "int64",
  "optional": true,
  "field": "ts_ms"
}
],
"optional": false,
"name": "dbserver1.inventory.customers.Envelope" ❹
},
"payload": { ❺
  "after": {"_id": {"$numberLong": "1004"}, "first_name": "Anne", "last_name":
  "Kretchmar", "email": "annek@noanswer.org"}, ❻
  "source": { ❷
    "version": "2.1.4.Final",
    "connector": "mongodb",
    "name": "fulfillment",
    "ts_ms": 1558965508000,
    "snapshot": false,

```

```

"db": "inventory",
"rs": "rs0",
"collection": "customers",
"ord": 31,
"h": 1546547425148721999
},
"op": "c", 8
"ts_ms": 1558965515240 9
}
}

```

表4.7 作成 イベント値フィールドの説明

項目	フィールド名	説明
1	schema	値のペイロードの構造を記述する、値のスキーマ。変更イベントの値スキーマは、コネクターが特定のコレクションに生成するすべての変更イベントで同じになります。
2	name	schema セクションで、各 name フィールドは、値のペイロードのフィールドに対するスキーマを指定します。 io.debezium.data.Json はペイロードの after 、 patch 、および filter フィールドのスキーマです。このスキーマは customers コレクションに固有です。 作成 イベントは、 after フィールドが含まれる唯一のイベントです。 更新 イベントには、 filter フィールドと patch フィールドが含まれます。 delete イベントには filter フィールドが含まれますが、 after フィールドや patch フィールドは含まれません。
3	name	io.debezium.connector.mongo.Source はペイロードの source フィールドのスキーマです。このスキーマは MongoDB コネクターに固有です。コネクターは生成するすべてのイベントにこれを使用します。
4	name	dbserver1.inventory.customers.Envelope は、ペイロードの全体的な構造のスキーマで、 dbserver1 はコネクター名、 inventory はデータベース、 customers はコレクションを指します。このスキーマはコレクションに固有です。
5	payload	値の実際のデータ。これは、変更イベントが提供する情報です。 イベントの JSON 表現はそれが記述するドキュメントよりもはるかに大きいように見えることがあります。これは、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です。しかし、 Avro コンバーター を使用すると、コネクターが Kafka トピックにストリーミングするメッセージのサイズを大幅に小さくすることができます。

項目	フィールド名	説明
6	after	<p>イベント発生後のドキュメントの状態を指定する任意のフィールド。この例では、after フィールドには新しいドキュメントの _id、first_name、last_name、および email フィールドの値が含まれます。after の値は常に文字列です。慣例により、ドキュメントの JSON 表現が含まれます。StreamsMongoDB oplog エントリーには、_create_ イベントと update イベントに対してのみドキュメントの完全な状態が含まれます (capture.mode オプションが change_streams_update_full に設定されている場合)。つまり、create イベントは capture.mode オプションに関係なく after フィールドが含まれる唯一のイベントです。</p>
7	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、イベントの発生元、イベントの発生順序、およびイベントが同じトランザクションの一部であるかどうかなど、このイベントと他のイベントを比較するために使用できる情報が含まれています。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン。 ● イベントを生成したコネクタの名前。 ● 生成されたイベントの namespace を形成し、コネクタが書き込む Kafka トピック名で使用される、MongoDB レプリカセットの論理名。 ● 新しいドキュメントが含まれるコレクションおよびデータベースの名前。 ● イベントがスナップショットの一部である場合。 ● データベースで変更が加えられた時点のタイムスタンプおよびタイムスタンプ内のイベントの順序。 ● MongoDB 操作の一意的識別子 (oplog イベントの h フィールド)。 ● MongoDB セッションの一意的識別子 lsid と、トランザクション内で変更が実行された場合のトランザクション番号 txnNumber (変更ストリームキャプチャモードのみ) です。
8	op	<p>コネクタによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、c は操作によってドキュメントが作成されたことを示しています。有効な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (読み取り、スナップショットのみに適用)

項目	フィールド名	説明
9	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

Chang Streams Capture モード

サンプル **customers** コレクションにある更新の変更イベントの値には、そのコレクションの **作成** イベントと同じスキーマがあります。同様に、イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。**update** イベントに **after** 値が含まれるのは、**capture.mode** オプションが **change_streams_update_full** に設定されている場合のみです。**capture.mode** オプションが ***_with_pre_image** オプションのいずれかに設定されている場合、**before** 値が指定されます。この場合、新たな構造化フィールド **updateDescription** が追加されました。

- **updatedFields** は、更新されたドキュメントフィールドの JSON 表現とその値を含む文字列フィールドです
- **removedFields** は、ドキュメントから削除されたフィールド名のリストです。
- **truncatedArrays** は、省略されたドキュメントの配列の一覧です。

以下は、コネクターによって **customers** コレクションでの更新に生成されるイベントの変更イベント値の例になります。

```
{
  "schema": { ... },
  "payload": {
    "op": "u", ①
    "ts_ms": 1465491461815, ②
    "before": {"_id": {"$numberLong": "1004"}, "first_name": "unknown", "last_name":
  "Kretchmar", "email": "annek@noanswer.org"}, ③
    "after": {"_id": {"$numberLong": "1004"}, "first_name": "Anne Marie", "last_name":
  "Kretchmar", "email": "annek@noanswer.org"}, ④
    "updateDescription": {
      "removedFields": null,
      "updatedFields": {"first_name": "Anne Marie"}, ⑤
      "truncatedArrays": null
    },
    "source": { ⑥
      "version": "2.1.4.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": false,
      "db": "inventory",
      "rs": "rs0",
      "collection": "customers",
```

```

"ord": 1,
"h": null,
"tord": null,
"stxnid": null,
"lsid":{"id":{"binary":"FA7YEzXgQXSX9OxmzllH2w==","type":"04"},"uid":{"binary":"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=","type":"00"}},
"txnNumber":1
}
}
}

```

表4.8 更新 イベント値フィールドの説明

項目	フィールド名	説明
1	op	コネクタによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、 u は操作によってドキュメントが更新されたことを示しています。
2	ts_ms	コネクタ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。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。 source オブジェクトで、 ts_ms は変更がデータベースに加えられた時間を示します。 payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。
3	before	変更前の実際の MongoDB ドキュメントの JSON 文字列表現が含まれています。+ キャプチャーモードが *_with_preimage オプションのいずれかに設定されていない場合、 update イベント値には before フィールドが含まれません。
4	after	実際の MongoDB ドキュメントを表す JSON 文字列が含まれます。キャプチャーモードが change_streams_update_full に設定されていない場合、 更新 イベントの値に after フィールドが含まれません。
5	updatedFields	ドキュメントの更新されたフィールド値の JSON 文字列表現が含まれます。この例では、更新により first_name フィールドが新しい値に変更されました。

項目	フィールド名	説明
6	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、同じコレクションの 作成 イベントと同じ情報が含まれますが、oplog の異なる位置からのイベントであるため、値は異なります。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン。 ● イベントを生成したコネクターの名前。 ● 生成されたイベントの namespace を形成し、コネクターが書き込む Kafka トピック名で使用される、MongoDB レプリカセットの論理名。 ● 更新されたドキュメントが含まれるコレクションおよびデータベースの名前。 ● イベントがスナップショットの一部である場合。 ● データベースで変更が加えられた時点のタイムスタンプおよびタイムスタンプ内のイベントの順序。 ● MongoDB セッションの一意的識別子 lsid とトランザクション番号 txnNumber (変更がトランザクションの中で実行された場合) です。



警告

イベント内の **after** の値は、ドキュメントの `at-point-of-time` の値として処理される必要があります。この値は動的に計算されるのではなく、コレクションから取得される。このため、複数の更新が次々に行われる場合、すべての **更新** イベントには、文書に保存されている最後の値を表す同じ **after** 値が含まれる可能性がある。

アプリケーションが段階的な変更の進化に依存している場合は、**updateDescription** のみに依存する必要があります。

削除 イベント

`delete change` イベントの値は、`create` や `update` と同じ **schema** 部分を持ちます。削除 イベントの **payload** 部分には、同じコレクションの **作成** と **更新** イベントとは異なる値が含まれます。特に、削除 イベントには **after** 値も **updateDescription** 値も含まれません。以下は、**customers** コレクションのドキュメントの **削除** イベントの例になります。

```
{
  "schema": { ... },
  "payload": {
    "op": "d", 1
    "ts_ms": 1465495462115, 2
  }
}
```

```

"before":{"_id":{"$numberLong":"1004"},"first_name":"Anne Marie","last_name":
\\"Kretchmar","email":"annek@noanswer.org"},"3
"source":{"4
"version":"2.1.4.Final",
"connector":"mongodb",
"name":"fulfillment",
"ts_ms":1558965508000,
"snapshot":true,
"db":"inventory",
"rs":"rs0",
"collection":"customers",
"ord":6,
"h":1546547425148721999
}
}
}

```

表4.9 削除 イベント値フィールドの説明

項目	フィールド名	説明
1	op	操作の型を記述する必須の文字列。 op フィールドの値は d で、ドキュメントが削除されたことを示します。
2	ts_ms	コネクタ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。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。 source オブジェクトで、 ts_ms は変更がデータベースに加えられた時間を示します。 payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。
3	before	変更前の実際の MongoDB ドキュメントの JSON 文字列表現が含まれています。+ キャプチャーモードが *_with_preimage オプションのいずれかに設定されていない場合、 update イベント値には before フィールドが含まれません。

項目	フィールド名	説明
4	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、同じコレクションの 作成 または 更新 イベントと同じ情報が含まれますが、oplog の異なる位置からのイベントであるため、値は異なります。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン。 ● イベントを生成したコネクターの名前。 ● 生成されたイベントの namespace を形成し、コネクターが書き込む Kafka トピック名で使用される、MongoDB レプリカセットの論理名。 ● 削除されたドキュメントが含まれたコレクションおよびデータベースの名前。 ● イベントがスナップショットの一部である場合。 ● データベースで変更が加えられた時点のタイムスタンプおよびタイムスタンプ内のイベントの順序。 ● MongoDB 操作の一意的識別子 (oplog イベントの h フィールド)。 ● MongoDB セッションの一意的識別子 lsid と、トランザクション内で変更が実行された場合のトランザクション番号 txnNumber (変更ストリームキャプチャモードのみ) です。

MongoDB コネクターイベントは、[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。ログコンパクションにより、少なくとも各キーの最新のメッセージが保持される限り、一部の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。

廃棄 (tombstone) イベント

一意に識別ドキュメントの MongoDB コネクターイベントはすべて同じキーを持ちます。ドキュメント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、**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、Kafka がそのキーを持つすべてのメッセージを削除するには、メッセージの値が **null** である必要があります。これを可能にするために、Debezium の MongoDB コネクターは **削除** イベントを出力した後に、**null** 値以外で同じキーを持つ特別な廃棄 (tombstone) イベントを出力します。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。

4.4. DEBEZIUM コネクターと連携する MONGODB の設定

MongoDB コネクターは MongoDB の変更ストリーム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、**listDatabases** 権限も必要です。変更ストリームを使用する場合 (デフォルト)、

ユーザーはクラスター全体の特権アクションである **find** および **changeStream** も持っている必要があります。

pre-image を使用して **before** フィールドに入力する場合は、最初に **db.createCollection()**、**create**、または **collMod** を使用してコレクションの **changeStreamPreAndPostImages** を有効にする必要があります。

4.5. DEBEZIUM MONGODB コネクターのデプロイメント

以下の方法のいずれかを使用して Debezium MongoDB コネクターをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタープラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。

関連情報

- [「Debezium MongoDB コネクターの設定プロパティを説明します。」](#)

4.5.1. AMQ Streams を使用した MongoDB コネクターデプロイメント

Debezium 1.7 以降、Debezium コネクターのデプロイに推奨される方法は、AMQ Streams を使用してコネクタープラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクターアーティファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクターがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクターを開始します。

Kafka Connect イメージのビルド仕様では、デプロイ可能なコネクターを指定できます。各コネクタープラグインに対して、デプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーティファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をデプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクターを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。

- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

4.5.2. AMQ Streams を使用した Debezium MongoDB コネクターのデプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクターをデプロイするには、最初にコネクター用の Kafka Connect イメージをビルドする必要がありました。コネクターを OpenShift にデプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクタープラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクター定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリまたは別の設定済みの HTTP サーバーから必要なアーティファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのデプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクターを起動します。

前提条件

- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスターが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合は、以下が必要です。

- [ImageStream](#) リソースがクラスターにデプロイされている。クラスターの ImageStream を明示的に作成している。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスターにログインします。
2. コネクターの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例4.1 Debezium コネクターを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium MongoDB コネクターアーカイブ。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクターで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプティング SMT アーカイブと、Debezium コネクターで使用する関連スクリプティングエンジン。SMT アーカイブとスクリプト言語の依存関係はオプションのコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場合にのみ、これらのコンポーネントを追加します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
  type: imagestream ❹
  image: debezium-streams-connect:latest
  plugins: ❺
  - name: debezium-connector-mongodb
    artifacts:
      - type: zip ❻
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mongodb/2.1.4.Final-redhat-00001/debezium-connector-mongodb-2.1.4.Final-redhat-00001-plugin.zip ❼
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3.0.Final-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3.0.Final-redhat-<build-number>.zip ❽
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip ❾
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar ❿
      - type: jar

```

```

url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
jsr223/3.0.11/groovy-jsr223-3.0.11.jar
- type: jar
url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
json3.0.11/groovy-json-3.0.11.jar

bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

...

```

表4.10 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクタを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクタのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタプラグインに対して、コネクタと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクタアーカイブは、 .zip ファイル形式で提供されます。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。
7	artifacts.url の値は、コネクタアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。Debezium コネクタアーティファクトは Red Hat リポジトリで入手できます。OpenShift クラスターは指定されたサーバーにアクセスする必要があります。

項目	説明
8	(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクターが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。
9	(オプション) Debezium コネクターで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベースルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 60px; height: 60px; background-color: black; margin-right: 10px;"></div> <div> <p>重要</p> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタープラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクターが失敗します。</p> </div> </div> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>別の方法として、Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスタを起動します。設定に一覧表示されているコネクターアーティファクトはクラスタで利用できます。

- KafkaConnector** リソースを作成し、デプロイする各コネクターのインスタンスを定義します。たとえば、以下の **KafkaConnector** CR を作成し、**mongodb-inventory-connector.yaml** として保存します。

例4.2 Debezium コネクターの KafkaConnector カスタムリソースを定義する `mongodb-inventory-connector.yaml` ファイル

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-mongodb ❶
spec:
  class: io.debezium.connector.mongodb.MongoDbConnector ❷
  tasksMax: 1 ❸
  config: ❹
    mongodb.hosts: rs0/192.168.99.100:27017 ❺
    mongodb.user: debezium ❻
    mongodb.password: dbz ❼
    topic.prefix: inventory-connector-mongodb ❽
    collection.include.list: inventory[*] ❾

```

表4.11 コネクター設定の説明

項目	説明
1	Kafka Connect クラスタに登録するコネクターの名前。
2	コネクタークラスの名前。
3	同時に動作できるタスクの数。
4	コネクターの設定。
5	ホストデータベースインスタンスのアドレスおよびポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名前。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。
8	データベースインスタンスまたはクラスタのトピック接頭辞。 指定する名前は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクターから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名前はクラスタのコネクター間で一意である必要があります。 コネクターを Avro コネクター と統合する場合、この名前空間は関連する Kafka Connect スキーマの名前や、対応する Avro スキーマの名前空間でも使用されます。
9	コネクターが変更をキャプチャーするコレクションの名前。

5. 以下のコマンドを実行してコネクターリソースを作成します。

-

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium MongoDB のデプロイメントを確認](#)する準備が整いました。

4.5.3. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium MongoDB コネクタのデプロイ

Debezium MongoDB コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、2つのカスタムリソース (CR) を作成します。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium MongoDB コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用するのと同じ OpenShift インスタンスに適用します。

前提条件

- MongoDB が稼働し、[MongoDB を設定して Debezium コネクタと連携する](#)手順が完了済みである必要があります。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。
- Podman または Docker がインストールされている。
- Debezium コネクタを実行するコンテナを追加する予定のコンテナレジストリー ([quay.io](#) や [docker.io](#) など) でコンテナを作成および管理するアカウントとパーミッションを持っている。

手順

1. Kafka Connect の Debezium MongoDB コンテナを作成します。
 - a. [registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12](#) をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-mongodb.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium 2
RUN cd /opt/kafka/plugins/debezium/ \
```

```
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
mongodb/2.1.4.Final-redhat-00001/debezium-connector-mongodb-2.1.4.Final-redhat-
00001-plugin.zip \
&& unzip debezium-connector-mongodb-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-mongodb-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-mongodb.yaml** という名前の Dockerfile を作成します。

- b. 前のステップで作成した **debezium-container-for-mongodb.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

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

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

上記のコマンドは、**debezium-container-for-mongodb** という名前のコンテナイメージを構築します。

- c. カスタムイメージを **quay.io** などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

```
podman push <myregistry.io>/debezium-container-for-mongodb:latest
```

```
docker push <myregistry.io>/debezium-container-for-mongodb:latest
```

- d. 新しい Debezium MongoDB **KafkaConnect** カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
```


前提条件

- Debezium コネクタが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクタの名前をクリックします (例: **inventory-connector-mongodb**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-mongodb -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例4.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-mongodb
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector
```

...

Status:

Conditions:

Last Transition Time: 2021-12-08T17:41:34.897153Z

Status: True

Type: Ready

Connector Status:

Connector:

State: RUNNING

```

worker_id: 10.131.1.124:8083
Name:      inventory-connector-mongodb
Tasks:
  Id:      0
  State:   RUNNING
worker_id: 10.131.1.124:8083
Type:      source
Observed Generation: 1
Tasks Max: 1
Topics:
inventory-connector-mongodb.inventory
inventory-connector-mongodb.inventory.addresses
inventory-connector-mongodb.inventory.customers
inventory-connector-mongodb.inventory.geom
inventory-connector-mongodb.inventory.orders
inventory-connector-mongodb.inventory.products
inventory-connector-mongodb.inventory.products_on_hand
Events: <none>

```

2. コネクターによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. **KafkaTopics** リストから確認するトピックの名前をクリックします (例: **inventory-connector-mongodb.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例4.4 KafkaTopic リソースのステータス

NAME	PARTITIONS	REPLICATION FACTOR	READY	CLUSTER
connect-cluster-configs	1	True		debezium-kafka-cluster
connect-cluster-offsets	1	True		debezium-kafka-cluster
connect-cluster-status	1	True		debezium-kafka-cluster
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fad1c6a	50	1	True	debezium-kafka-cluster

```

inventory-connector-mongodb--a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster 1 1 True
inventory-connector-mongodb.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480 debezium-kafka-cluster
1 1 True
inventory-connector-mongodb.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b debezium-kafka-cluster 1
1 True
inventory-connector-mongodb.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5 debezium-kafka-cluster
1 1 True
inventory-connector-mongodb.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d debezium-kafka-cluster
1 1 True
inventory-connector-mongodb.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef debezium-kafka-cluster 1
1 True
inventory-connector-mongodb.inventory.products_on_hand---
8649e0f17fcc9212e266e31a7aeea4585e5c6b5 debezium-kafka-cluster 1
1 True
schema-changes.inventory debezium-kafka-cluster
1 1 True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55 debezium-
kafka-cluster 1 1 True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b debezium-kafka-cluster 1 1
True

```

3. トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。

```

oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>

```

以下に例を示します。

```

oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-mongodb.inventory.products_on_hand

```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory-connector-mongodb.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例4.5 Debezium 変更イベントの内容


```

{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},"optional":false,"name":"inventory-
connector-mongodb.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-mongodb.inventory.products_on_hand.Value","field":"before"},
{"type":"struct","fields":{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-mongodb.inventory.products_on_hand.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":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"},"optional":false,"name":"io.debezium.connecto
r.mongodb.Source","field":"source"},{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"},{"type":"struct","fields":
[{"type":"string","optional":false,"field":"id"},
{"type":"int64","optional":false,"field":"total_order"},
{"type":"int64","optional":false,"field":"data_collection_order"},"optional":true,"field":"transacti
on"},"optional":false,"name":"inventory-connector-
mongodb.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":
{"product_id":101,"quantity":3,"source":{"version":"2.1.4.Final-redhat-
00001","connector":"mongodb","name":"inventory-connector-
mongodb","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table
":"products_on_hand","server_id":0,"gtid":null,"file":"mongodb-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}

```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

4.5.5. Debezium MongoDB コネクターの設定プロパティを説明します。

Debezium MongoDB コネクターには、アプリケーションに適したコネクター動作を実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。プロパティに関する情報は、以下のように設定されています。

- [必要な Debezium MongoDB コネクター設定プロパティ](#)
- [高度な Debezium MongoDB コネクター設定プロパティ](#)

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

表4.12 必要な Debezium MongoDB コネクター設定プロパティ

プロパティ	デフォルト	説明
<code>name</code>	デフォルトなし	コネクターの一意名。同じ名前でも再登録を試みると失敗します。(このプロパティはすべての Kafka Connect コネクターに必要です)
<code>connector.class</code>	デフォルトなし	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。
<code>mongodb.hosts</code>	デフォルトなし	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。 <div style="display: flex; align-items: flex-start;"> <div style="flex: 1;">  </div> <div style="flex: 2;"> <p>注記</p> <p>現在のプライマリーアドレスを指定することが必須です。この制限は、次の Debezium リリースで削除される予定です。</p> </div> </div>


プロパティ	デフォルト	説明
mongodb.connection.string	デフォルトなし	<p>MongoDB レプリカセットの最初の検出中にコネクターが使用する接続文字列を指定します。このオプションを使用するには、mongodb.members.auto.discover の値を true に設定する必要があります。このプロパティと mongodb.hosts プロパティを同時に設定しないでください。</p> <p>注記</p> <p>コネクターは、最初のレプリカセット検出プロセス中にのみこの接続文字列を使用します。この検出プロセス中、コネクターは他のプロパティ (mongodb.user、mongodb.password、mongodb.authsource、SSL 設定プロパティなど) で指定された接続値を無視します。検出プロセスの完了後に、コネクターがプライマリーレプリカセットメンバーに対して直接接続を確立しようとする、コネクターは標準の接続プロパティを使用するようになり、mongodb.connection.string の値を無視します。ただし、設定の他の場所に認証情報がない場合は、コネクターは接続文字列の値から認証情報を抽出できます。たとえば、mongodb.user プロパティが設定されておらず、接続文字列に MongoDB ユーザー名が含まれる場合、コネクターは文字列から情報を読み取ります。</p>

プロパティ	デフォルト	説明
topic.prefix	デフォルトなし	<p>このコネクタが監視するコネクタや 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 コネクタによって監視される必要があります。名前を設定する文字は、英数字、ハイフン、ドット、アンダースコアのみです。論理名は、このコネクタからレコードを受信する Kafka トピックに名前を付ける際の接頭辞として使用されるため、他のすべてのコネクタで一意である必要があります。</p> <div data-bbox="884 801 1430 1305" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <p> 警告</p> <p>このプロパティの値を変更しないでください。名前の値を変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。</p> </div>
mongodb.user	デフォルトなし	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	デフォルトなし	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。
mongodb.authsource	admin	MongoDB クレデンシャルが含まれるデータベース (認証ソース)。これは、MongoDB が admin 以外の認証データベースで認証を使用するように設定されている場合に必要です。
mongodb.ssl.enabled	false	コネクタは SSL を使用して MongoDB インスタンスに接続します。

プロパティ	デフォルト	説明
<code>mongodb.ssl.invalid.host.name.allowed</code>	<code>false</code>	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。 <code>true</code> に設定すると、接続で中間者攻撃は阻止されません。
<code>database.include.list</code>	空の文字列	<p>監視対象のデータベース名に一致する正規表現のオプションのコンマ区切りリスト。デフォルトでは、すべてのデータベースが監視されます。</p> <p><code>database.include.list</code> が設定されている場合、コネクターはプロパティが指定するデータベースのみを監視します。他のデータベースは監視から除外されます。</p> <p>データベースの名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データベースの名前文字列全体に対して照合されます。データベース名に存在する可能性のある部分文字列とは一致しません。このプロパティを設定に含める場合は、<code>database.exclude.list</code> プロパティも設定しないでください。</p>
<code>database.exclude.list</code>	空の文字列	<p>監視対象から除外するデータベース名に一致する正規表現のオプションのコンマ区切りリスト。<code>database.exclude.list</code> が設定されている場合、コネクターは、プロパティで指定されたものを除くすべてのデータベースを監視します。</p> <p>データベースの名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データベースの名前文字列全体に対して照合されます。データベース名に存在する可能性のある部分文字列とは一致しません。このプロパティを設定に含める場合は、<code>database.include.list</code> プロパティを設定しないでください。</p>

プロパティ	デフォルト	説明
collection.include.list	空の文字列	<p>監視対象の MongoDB コレクションの完全修飾名前空間に一致する、オプションのコンマ区切りの正規表現のリスト。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.include.list が設定されている場合、コネクターはプロパティが指定するコレクションのみを監視します。他のコレクションは監視から除外されます。コレクション識別子の形式は databaseName.collectionName です。</p> <p>名前空間の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、名前の部分文字列との一致ではなく、名前空間の名前文字列全体に対して照合されます。このプロパティを設定に含める場合は、collection.exclude.list プロパティを設定しないでください。</p>
collection.exclude.list	空の文字列	<p>監視から除外する MongoDB コレクションの完全修飾名前空間に一致する正規表現のオプションのコンマ区切りリスト。collection.exclude.list が設定されている場合、コネクターはプロパティで指定されたコレクション以外のすべてのコレクションを監視します。コレクション識別子の形式は databaseName.collectionName です。</p> <p>名前空間の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データベースの名前文字列全体に対して照合されます。データベース名に存在する可能性のある部分文字列とは一致しません。このプロパティを設定に含める場合は、collection.include.list プロパティを設定しないでください。</p>

プロパティ	デフォルト	説明
<code>snapshot.mode</code>	<code>Initial</code>	<p>コネクターの開始時にスナップショットを実行する基準を指定します。このプロパティを以下の値のいずれかに設定します。</p> <p>Initial</p> <p>コネクターの開始時にオフセットピックで値が検出されない場合、コネクターはデータベースのスナップショットを実行します。</p> <p>never</p> <p>コネクターが起動すると、スナップショットプロセスをスキップし、すぐにデータベースで oplog に記録する操作の変更イベントのストリーミングを開始します。</p>
<code>capture.mode</code>	<code>change_streams_update_full</code>	<p>コネクターが MongoDB サーバーから update イベントでの変更のキャプチャーに使用するメソッドを指定します。このプロパティを以下のいずれかの値に設定します。</p> <p>change_streams</p> <p>update イベントメッセージには完全なドキュメントは含まれません。メッセージには、変更前 のドキュメントの状態を表すフィールドは含まれません。</p> <p>change_streams_update_full</p> <p>update イベントメッセージには完全なドキュメントが含まれます。メッセージには、更新前 のドキュメントの状態を表す before フィールドは含まれません。イベントメッセージは、ドキュメントの完全な状態を after フィールドで返します。</p>

プロパティ	デフォルト	説明	注記
			<p>一部の状況では、capture.mode が完全なドキュメントを返すように設定されている場合、update イベントメッセージの updateDescription フィールドと after フィールドが矛盾した値を報告することがあります。このような不一致は、ドキュメントに複数の更新が立て続けに適用された後に発生する可能性があります。コネクタは、イベントの updateDescription フィールドに記述された更新を受信した後にのみ、MongoDB データベースから完全なドキュメントを要求します。コネクタがデータベースからソースドキュメントを取得する前に、後から行われた更新によってソースドキュメントが変更された場合、コネクタは、この後の更新で変更されたドキュメントを受け取ります。</p> <p>完全なドキュメントを返すように capture.mode を設定すると、メッセージの updateDescription フィールドと after フィールドの内容が一致しない場合があります。ドキュメントに複数の変更が立て続けに適用されると、不一致が生じる可能性があります。コネクタは、更新の適用後にしか完全なドキュメントのリクエストを送信しないので、updateDescription フィールドに記述されている更新が済んでから、後から行われる更新でソースドキュメントが変更される可能性があります。コネクタがクエリーに応答して受け取る完全なドキュメントには、その後の変更結果が反映されます。</p> <p>change_streams_update_full_with_pre_image</p> <p>update イベントのメッセージには、完全なドキュメントが含まれ、変更前のドキュメントの状態を表すフィールドが含まれます。</p> <p>change_streams_with_pre_image</p>

プロパティ	デフォルト	説明 update イベントには完全なドキュメントは含まれませんが、変更前のドキュメントの状態を表すフィールドが含まれます。
snapshot.include.collection.list	collection.include.list に指定されたすべてのコレクション	<p>スナップショットに含めるスキーマの完全修飾名 (<databaseName>.<collectionName>) と一致する正規表現のコンマ区切りのリスト (任意)。指定された項目は、コネクターの collection.include.list プロパティで指定する必要があります。このプロパティは、コネクターの snapshot.mode プロパティが never 以外の値に設定されている場合にのみ有効です。</p> <p>このプロパティは増分スナップショットの動作には影響しません。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの名前文字列全体と照合されます。</p>

プロパティ	デフォルト	説明
<code>field.exclude.list</code>	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 <code>databaseName.collectionName.fieldName.nestedFieldName</code> で、 <code>databaseName</code> および <code>collectionName</code> 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。
<code>field.renames</code>	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 <code>databaseName.collectionName.fieldName.nestedFieldName:newNestedFieldName</code> で、 <code>databaseName</code> および <code>collectionName</code> 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。
<code>tasks.max</code>	1	このコネクタのために作成する必要があるタスクの最大数。MongoDB コネクタ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せん。そのため、コネクタを単一の 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。
<code>snapshot.max.threads</code>	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。

プロパティ	デフォルト	説明
tombstones.on.delete	true	<p>削除 イベントの後に廃棄 (tombstone) イベントが続くかどうかを制御します。</p> <p>true: 削除操作は、削除 イベントと後続の破棄 (tombstone) イベントで表されます。</p> <p>false - delete イベントのみが出力されます。</p> <p>log compaction がトピックで有効になっている場合には、ソースレコードの削除後に廃棄 (tombstone) イベントを出力すると (デフォルト動作)、Kafka は削除された行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。</p>
snapshot.delay.ms	デフォルトなし	<p>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)。</p> <p>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。</p>
snapshot.fetch.size	0	<p>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。</p> <p>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。</p>
schema.name.adjustment.mode	none	<p>コネクターで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能:</p> <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。

以下の **高度な** 設定プロパティには、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。

表4.13 Debezium MongoDB コネクターの詳細設定プロパティ

プロパティ	デフォルト	説明
max.batch.size	2048	<p>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。</p>

プロパティ	デフォルト	説明
max.queue.size	8192	ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクタが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクタがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。 max.queue.size の値を、 max.batch.size の値よりも大きくなるように設定します。
max.queue.size.in.bytes	0	ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。 max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、 max.queue.size=1000 、 max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。
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 500 ミリ秒 (0.5 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	デフォルト	説明
<code>connect.max.attempts</code>	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。 <code>connect.backoff.initial.delay.ms</code> と <code>connect.backoff.max.delay.ms</code> 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。
<code>mongodb.members.auto.discover</code>	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 (<code>true</code>)、または <code>mongodb.hosts</code> のアドレスをそのまま使用する必要があるかどうか (<code>false</code>) を指定するブール値。デフォルトは <code>true</code> で、MongoDB が プロキシと面する 場合を除き、すべてのケースで使用する必要があります。
<code>heartbeat.interval.ms</code>	0	<p>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。</p> <p>このプロパティには、コネクターがメッセージをハートビートピックに送信する頻度を定義する間隔(ミリ秒単位)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 <code>oplog/change streams</code> 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、<code>oplog</code> 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。</p> <p>このプロパティを 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。デフォルトでは無効にされています。</p>

プロパティ	デフォルト	説明
sanitize.field.names	コネクタ設定が、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。
skipped.operations	t	ストリーミング中にスキップされる oplog 操作のコンマ区切りリスト。挿入/作成は c 、更新は u 、削除は d 、切り捨ては t 、操作をスキップしない場合は none となります。デフォルトでは、省略操作はスキップされます (このコネクタによる出力ではない)。
snapshot.collection.filter.overrides	デフォルトなし	<p>スナップショットに含まれるコレクション項目を制御します。このプロパティはスナップショットにのみ影響します。 databaseName.collectionName の形式でコレクション名のコンマ区切りリストを指定します。</p> <p>指定する各コレクションに対して、別の設定プロパティ (snapshot.collection.filter.overrides.databaseName.collectionName) も指定します。たとえば、他の設定プロパティの名前は snapshot.collection.filter.overrides.customers.orders などです。このプロパティは、スナップショットに必要なアイテムのみを取得する有効なフィルター式に設定します。コネクタがスナップショットを実行すると、フィルター式と一致する項目のみを取得します。</p>
provide.transaction.metadata	false	<p>true に設定すると、Debezium はトランザクション境界でイベントを生成し、トランザクションメタデータでデータイベントエンベロープをエンリッチします。</p> <p>詳細は、 トランザクションメタデータ を参照してください。</p>
retriable.restart.connector.wait.ms	10000 (10 秒)	再試行可能なエラーが発生した後にコネクタを再起動するまで待機する時間 (ミリ秒単位)。

プロパティ	デフォルト	説明
<code>mongodb.poll.interval.ms</code>	30000	コネクターが新規、削除、または変更したレプリカセットをポーリングする間隔。
<code>mongodb.connect.timeout.ms</code>	10000 (10 秒)	新しい接続試行が中断されるまでドライバーが待機する時間 (ミリ秒単位)。
<code>mongodb.heartbeat.frequency.ms</code>	10000 (10 秒)	クラスターモニターが各サーバーへのアクセスを試行する頻度。
<code>mongodb.socket.timeout.ms</code>	0	ソケットでの送受信がタイムアウトするまでにかかる時間 (ミリ秒単位)。0 の値は、この動作を無効にします。
<code>mongodb.server.selection.timeout.ms</code>	30000 (30 秒)	ドライバーがタイムアウトし、エラーが出力される前に、サーバーが選択されるまでドライバーが待つ時間 (ミリ秒単位)。
<code>cursor.max.await.time.ms</code>	0	実行タイムアウトの例外を発生させる前に、oplog/Change Streams カーソルが結果を生成するのを待つ最大期間 (ミリ秒単位) を指定します。値 0 は、サーバー/ドライバーのデフォルト待機タイムアウトを使用することを示します。
<code>signal.data.collection</code>	デフォルトなし	シグナルをコネクターへの送信に使用されるデータコレクションの完全修飾名。コレクション名を指定するには、 <databaseName>.<collectionName> の形式を使用します。 シグナリングは、Debezium MongoDB コネクターの技術プレビュー機能です。
<code>incremental.snapshot.chunk.size</code>	1024	増分スナップショットチャンク中にコネクターがフェッチしてメモリーに読み込むドキュメントの最大数です。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファーにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。増分スナップショットは、Debezium MongoDB コネクターのテクノロジープレビュー機能です。

プロパティ	デフォルト	説明
topic.naming.strategy	<code>io.debezium.schema.DefaultTopicNamingStrategy</code>	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する TopicNamingStrategy クラスの名前。デフォルトは DefaultTopicNamingStrategy 。
topic.delimiter	<code>.</code>	トピック名の区切り文字を指定します。デフォルトは <code>.</code> です。
topic.cache.size	<code>10000</code>	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。
topic.heartbeat.prefix	<code>__debezium-heartbeat</code>	コネクターがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は <code>__debezium-heartbeat.fulfillment</code> になります。
topic.transaction	<code>transaction</code>	コネクターがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は <code>fulfillment.transaction</code> になります。

4.6. DEBEZIUM MONGODB コネクターのパフォーマンスの監視

Debezium MongoDB コネクターには、Zookeeper、Kafka、および Kafka Connect にある JMX メトリクスの組み込みサポートに加えて、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、スナップショットの実行中にコネクター操作に関する情報を提供します。
- [メトリクスのストリーミング](#) は、コネクターが変更をキャプチャーし、変更イベントレコードをストリーミングする際のコネクター操作に関する情報を提供します。

[Debezium モニタリングのドキュメント](#) では、JMX を使用してこれらのメトリクスを公開する方法の詳細を提供します。

4.6.1. MongoDB スナップショット作成中の Debezium の監視

MBean は `debezium.mongodb:type=connector-metrics,context=snapshot,server=<mongodb.server.name>,task=<task.id>` です。

スナップショット操作がアクティブでない場合や、最後のコネクターの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	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 include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	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。
QueueRemainingCapacity	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。

属性	タイプ	説明
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。
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ず。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

Debezium MongoDB コネクタは、以下のカスタムスナップショットメトリクスも提供します。

属性	タイプ	説明
NumberOfDisconnects	long	データベースの切断数。

4.6.2. Debezium MongoDB コネクターレコードストリーミングの監視

MBean は `debezium.mongodb:type=connector-metrics,context=streaming,server=<mongodb.server.name>,task=<task.id>` です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。
TotalNumberOfCreateEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクターが見た更新イベントの合計数。
TotalNumberOfDeleteEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。

属性	タイプ	説明
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。
Connected	boolean	コネクタ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との間の期間 (ミリ秒単位)。この値は、データベースサーバーとコネクタが稼働しているマシンのクロック間の差異に対応します。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

Debezium MongoDB コネクタは、以下のカスタムストリーミングメトリクスも提供します。

属性	タイプ	説明
NumberOfDisconnects	long	データベースの切断数。

属性	タイプ	説明
NumberOfPrimaryElections	long	プライマリーノードの選出数。

4.7. DEBEZIUM MONGODB コネクターによる障害および問題の処理方法

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。

障害が発生しても、システムからイベントがなくなることはありません。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があります。このような状態では、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。

以下のトピックでは、Debezium MongoDB コネクターがさまざまな種類の障害および問題を処理する方法を詳説します。

- [設定および起動エラー](#)
- [MongoDB が使用不可能になる](#)
- [Kafka Connect のプロセスは正常に停止する](#)
- [Kafka Connect プロセスのクラッシュ](#)
- [Kafka が使用不可能になる](#)
- [snapshot.mode が initial に設定されている場合、コネクターが長時間停止した後に失敗します。](#)
- [MongoDB による書き込みの損失](#)

設定および起動エラー

以下の状況では、起動時にコネクターが失敗し、エラーまたは例外がログに記録され、実行が停止されます。

- 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 MongoDB に接続できない。

失敗したら、コネクターは指数バックオフを使用して再接続を試みます。再接続試行の最大数を設定できます。

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、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。

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。

再接続の続試行は、3つのプロパティで制御されます。

- **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
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

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。

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タスクを再起動します。ただし、MongoDB コネクタ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 イベントの一意的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。

Kafka が使用不可能になる

変更イベントはコネクタ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発生す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、コネクタは停止した場所から再開します。

snapshot.mode が initial に設定されている場合、コネクタが長時間停止した後に失敗します。

コネクタが正常に停止された場合、ユーザーはレプリカセットメンバーで操作を継続する可能性があります。コネクタがオフライン時に発生する変更は、MongoDB の oplog に記録されます。ほとんどの場合、コネクタは再起動後、oplog のオフセット値を読み取って、各レプリカセットに対して最後にストリーミングした操作を特定し、その時点から変更のストリーミングを再開します。再起動後、コネクタの停止中に発生したデータベース操作は通常どおり Kafka に発行され、しばらくすると、コネクタはデータベースとの遅れを取り戻します。コネクタの操作が遅れを取り戻すのに必要な時間は、Kafka の機能とパフォーマンス、およびデータベースで発生した変更の量によって異なります。

ただし、コネクタが長時間停止したままになっていると、コネクタが非アクティブとなっている間に MongoDB が oplog をパージし、コネクタの最後の位置に関する情報が失われる可能性があります。コネクタの再起動後、コネクタが処理した最後の操作をマークする以前のオフセット値が oplog に含まれていないため、ストリーミングを再開できません。また、**snapshot.mode** プロパティが **initial** に設定されていて、オフセット値が存在しない場合にコネクタは通常どおりにスナッ

プッシュを実行できません。この場合、oplog には前のオフセットの値が含まれていないため、不一致が存在しますが、オフセット値はコネクターの内部 Kafka オフセットトピックに存在します。エラーが発生し、コネクターが失敗します。

障害から回復するには、障害が発生したコネクターを削除し、同じ設定で異なるコネクター名で新しいコネクターを作成します。新しいコネクターを開始すると、スナップショットが実行されてデータベースの状態が取り込まれ、ストリーミングが再開されます。

MongoDB による書き込みの損失

失敗した場合の特定の状況では、MongoDB はコミットを失う可能性があり、その場合には MongoDB コネクターでは、失われた変更をキャプチャできません。たとえば、プライマリーが変更を適用して、その oplog にその変更を記録した後に、突然クラッシュした場合には、セカンダリーノードがコンテンツを読み取るまでに oplog が利用できなくなる可能性があります。その結果、新しいプライマリーノードとして選択されるセカンダリーノードには、oplog からの最新の変更が含まれていない可能性があります。

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5章 MYSQL の DEBEZIUM コネクタ

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ログ (binlog) があります。これには、テーブルスキーマの変更やテーブル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。

Debezium MySQL コネクタは binlog を読み取り、行レベルの **INSERT**、**UPDATE**、および **DELETE** 操作の変更イベントを生成し、変更イベントを Kafka トピックに出力します。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 コネクタは各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。MySQL コネクタは、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。

このコネクタと互換性のある MySQL データベースのバージョンについては、[Debezium](#) でサポートされる設定ページを参照してください。

Debezium MySQL コネクタの使用に関する情報および手順は、以下のように整理されています。

- [「Debezium MySQL コネクタの仕組み」](#)
- [「Debezium MySQL コネクタのデータ変更イベントの説明」](#)
- [「Debezium MySQL コネクタによるデータ型のマッピング方法」](#)
- [「Debezium コネクタを実行するための MySQL の設定」](#)
- [「Debezium MySQL コネクタのデプロイメント」](#)
- [「Debezium MySQL コネクタのパフォーマンスの監視」](#)
- [「Debezium MySQL コネクタによる障害および問題の処理方法」](#)

5.1. DEBEZIUM MYSQL コネクタの仕組み

コネクタがサポートする MySQL トポロジーの概要は、アプリケーションを計画するときに役立ちます。Debezium MySQL コネクタを最適に設定および実行するには、コネクタによるテーブルの構造の追跡方法、スキーマ変更の公開方法、スナップショットの実行方法、および Kafka トピック名の決定方法を理解しておくことが便利です。

詳細は以下を参照してください。

- [「Debezium コネクタでサポートされる MySQL トポロジー」](#)
- [「Debezium MySQL コネクタによるデータベーススキーマの変更の処理方法」](#)
- [「Debezium MySQL コネクタによるデータベーススキーマの変更の公開方法」](#)
- [「Debezium MySQL コネクタによるデータベーススナップショットの実行方法」](#)
- [「Debezium MySQL 変更イベントレコードを受信する Kafka トピックのデフォルト名」](#)

5.1.1. Debezium コネクタでサポートされる MySQL トポロジー

Debezium MySQL コネクタは以下の 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。

プライマリーおよびレプリカ

Debezium MySQL コネクタはプライマリーサーバーまたはレプリカの1つ (レプリカの binlog が**有効**になっている場合) に従うことができますが、コネクタはサーバーが認識できるクラスターのみで変更を確認できます。通常、これはマルチプライマリートポロジ以外では問題ではありません。

コネクタは、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。このサーバーに障害が発生した場合、サーバーを再起動またはリカバリーしないと、コネクタは継続できません。

高可用性クラスター

MySQL にはさまざまな **高可用性** ソリューションが存在し、問題や障害の耐性をつけ、即座に回復することが大変容易になります。ほとんどの HA MySQL クラスターは GTID を使用します。そのため、レプリカはあらゆるプライマリーサーバーの変更をすべて追跡できます。

マルチプライマリー

ネットワークデータベース (NDB) クラスターのレプリケーション は、複数のプライマリーサーバーからそれぞれをレプリケートする1つ以上の MySQL レプリカを使用します。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す。このトポロジには GTID を使用する必要があります。

Debezium MySQL コネクタはこれらのマルチプライマリー MySQL レプリカをソースとして使用することができ、新しいレプリカが古いレプリカに追い付ければ、異なるマルチプライマリー MySQL レプリカにフェイルオーバーできます。つまり、新しいレプリカには最初のレプリカで確認されたすべてのトランザクションが含まれます。これは、新しいマルチプライマリー MySQL レプリカへの再接続を試み、binlog で適切な場所を見つけようとする際に、特定の GTID ソースが含まれるまたは除外されるようにコネクタを設定できるため、コネクタ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。

これらのホステッドオプションではグローバル読み取りロックが許可されないため、テーブルレベルロックを使用して **整合性スナップショット** を作成します。

5.1.2. Debezium MySQL コネクタによるデータベーススキーマの変更の処理方法

データベースクライアントがデータベースのクエリーを行うと、クライアントはデータベースの現在のスキーマを使用します。しかし、データベーススキーマはいつでも変更が可能です。そのため、挿入、更新、または削除の操作が記録されるたびに、コネクタはどのスキーマであるかを特定する必要があります。また、テーブルのスキーマが変更される前に記録された、比較的古いイベントをコネクタが処理するので、コネクタは現在のスキーマだけを使用することはできません。

スキーマの変更後に発生する変更を正しく処理するために、MySQL にはデータの行レベルの変更だけでなく、データベースに適用される DDL ステートメントも含まれます。コネクタは binlog を読み取り、DDL ステートメントを見つけると、それらの DDL ステートメントを解析し、各テーブルのスキーマのインメモリー表現を更新します。コネクタはこのスキーマ表現を使用して、挿入、更新、または

削除の操作時にテーブルの構造を特定し、適切な変更イベントを生成します。別のデータベーススキーマ履歴 Kafka トピックでは、コネクタは各 DDL ステートメントがある binlog の場所とともにすべての DDL ステートメントを記録します。

コネクタが正常にクラッシュまたは停止された後にコネクタが再起動されると、コネクタは特定の場所 (特定の時点) から binlog の読み取りを開始します。コネクタは、データベーススキーマ履歴の Kafka トピックを読み取り、コネクタが起動する binlog の時点まですべての DDL ステートメントを解析することで、この時点で存在したテーブル構造を再ビルドします。

このデータベーススキーマ履歴トピックは、コネクタのみ用です。コネクタは任意で、[コンシューマーアプリケーション向けの異なるトピックへのスキーマ変更イベントの生成を表示](#)できます。

MySQL コネクタが、**gh-ost** または **pt-online-schema-change** などのスキーマ変更ツールが適用されるテーブルで変更をキャプチャーすると、移行プロセス中にヘルパーテーブルが作成されます。これらのヘルパーテーブルへの変更をキャプチャーするようにコネクタを設定する必要があります。ヘルパーテーブル用に生成されたレコードがコンシューマーに必要ない場合は、メッセージ変換を1回適用して、除去できます。

詳細は、Debezium イベントレコードを受け取る [トピックのデフォルト名](#) を参照してください。

5.1.3. Debezium MySQL コネクタによるデータベーススキーマの変更の公開方法

Debezium MySQL コネクタを設定すると、データベースのキャプチャーされたテーブルに適用されるスキーマの変更を記述するスキーマ変更イベントを生成できます。コネクタはスキーマ変更イベントを `<topicPrefix>` という名前の Kafka トピックに書き込みます。ここで、**topicPrefix** は `topic.prefix` コネクタ設定プロパティで指定された名前空間です。コネクタがスキーマ変更トピックに送信するメッセージには、ペイロードと、任意で変更イベントメッセージのスキーマが含まれます。

スキーマ変更イベントメッセージのペイロードには、以下の要素が含まれます。

ddl

スキーマの変更につながる SQL **CREATE**、**ALTER**、または **DROP** ステートメントを提供します。

databaseName

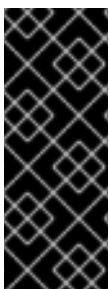
DDL ステートメントが適用されるデータベースの名前。**databaseName** の値は、メッセージキーとして機能します。

pos

ステートメントが表示される binlog の位置。

tableChanges

スキーマの変更後のテーブルスキーマ全体の構造化表現。**tableChanges** フィールドには、テーブルの各列のエントリなどのアレイが含まれます。構造化された表現は JSON または Avro 形式でデータを表示するため、コンシューマーは DDL パーサーを介して最初にメッセージを処理しなくてもメッセージを簡単に読み取りできます。



重要

キャプチャーモードであるテーブルでは、コネクタはスキーマ変更トピックにスキーマ変更の履歴だけでなく、内部データベーススキーマ履歴トピックにも格納します。内部データベーススキーマ履歴トピックはコネクタのみの使用を対象としており、使用するアプリケーションによる直接使用を目的としていません。スキーマ変更に関する通知が必要なアプリケーションが、スキーマ変更トピックからの情報のみを使用するようにしてください。

重要

データベーススキーマ履歴トピックをパーティションに分割しないでください。データベーススキーマ履歴トピックが正しく機能するには、コネクターが出力するイベントレコードの一貫したグローバル順序を維持する必要があります。

トピックがパーティション間で分割されないようにするには、以下のいずれかの方法を使用してトピックのパーティション数を設定します。

- データベーススキーマ履歴トピックを手動で作成する場合は、パーティション数を **1** に指定します。
- Apache Kafka ブローカーを使用してデータベーススキーマ履歴トピックを自動的に作成する場合に、トピックが作成されるので、**Kafka num.partitions** 設定オプションの値を **1** に設定します。



警告

コネクターがスキーマ変更トピックに出力するメッセージの形式は、初期の状態であり、通知なしに変更される可能性があります。

例: MySQL コネクタースキーマ変更トピックに出力されるメッセージ

以下の例は、JSON 形式の一般的なスキーマ変更メッセージを示しています。メッセージには、テーブルスキーマの論理表現が含まれます。

```
{
  "schema": { },
  "payload": {
    "source": { 1
      "version": "2.1.4.Final",
      "connector": "mysql",
      "name": "mysql",
      "ts_ms": 1651535750218, 2
      "snapshot": "false",
      "db": "inventory",
      "sequence": null,
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 570,
      "row": 0,
      "thread": null,
      "query": null
    },
    "databaseName": "inventory", 3
    "schemaName": null,
    "ddl": "ALTER TABLE customers ADD middle_name varchar(255) AFTER first_name", 4
    "tableChanges": [ 5
      {
```

```

"type": "ALTER", ⑥
"id": "\"inventory\".\"customers\"", ⑦
"table": { ⑧
  "defaultCharsetName": "utf8mb4",
  "primaryKeyColumnNames": [ ⑨
    "id"
  ],
  "columns": [ ⑩
    {
      "name": "id",
      "jdbcType": 4,
      "nativeType": null,
      "typeName": "INT",
      "typeExpression": "INT",
      "charsetName": null,
      "length": null,
      "scale": null,
      "position": 1,
      "optional": false,
      "autoIncremented": true,
      "generated": true
    },
    {
      "name": "first_name",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "VARCHAR",
      "typeExpression": "VARCHAR",
      "charsetName": "utf8mb4",
      "length": 255,
      "scale": null,
      "position": 2,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "middle_name",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "VARCHAR",
      "typeExpression": "VARCHAR",
      "charsetName": "utf8mb4",
      "length": 255,
      "scale": null,
      "position": 3,
      "optional": true,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "last_name",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "VARCHAR",

```

```

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

```

表5.1 スキーマ変更トピックに出力されたメッセージのフィールドの説明

項目	フィールド名	説明
1	source	source フィールドは、コネクターがテーブル固有のトピックに書き込む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。
2	ts_ms	<p>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。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。</p> <p>ソースオブジェクトの <code>ts_ms</code> は、データベースで変更が行われた時刻を示す。 <code>payload.source.ts_ms</code> の値を <code>payload.ts_ms</code> の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

項目	フィールド名	説明
3	databaseName schemaName	変更が含まれるデータベースとスキーマを識別します。 databaseName フィールドの値は、レコードのメッセージキーとして使用されます。
4	ddl	このフィールドには、スキーマの変更を行う DDL が含まれます。 ddl フィールドには複数の DDL ステートメントが含まれることがあります。各ステートメントは、 databaseName フィールドのデータベースに適用されます。ステートメントは、データベースに適用された順序で示されます。 クライアントは、複数のデータベースに適用される複数の DDL ステートメントを送信できます。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。
5	tableChanges	DDL コマンドによって生成されるスキーマの変更が含まれる1つ以上の項目の配列。
6	type	変更の種類を説明します。値は以下のいずれかになります。 CREATE テーブルの作成 ALTER テーブルの変更 DROP テーブルの削除
7	id	作成、変更、または破棄されたテーブルの完全な識別子。テーブルの名前が変更されると、この識別子は<old>,<new> のテーブル名が連結されます。
8	table	適用された変更後のテーブルメタデータを表します。
9	primaryKeyColumnNames	テーブルのプライマリーキーを設定する列のリスト。
10	列	変更されたテーブルの各列のメタデータ。
11	attributes	各テーブル変更のカスタム属性メタデータ。

詳細は、[スキーマ履歴トピック](#) を参照してください。

5.1.4. Debezium MySQL コネクタ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、コネクタによってこのスナップショットが作成される方法を示しています。このフローは、デフォルト **initial** のスナップショットモード用です。その他のスナップショットモードの詳細は、[MySQL コネクタ `snapshot.mode` 設定プロパティ](#) を参照してください。

表5.2 グローバル読み取りロックを使用して最初のスナップショットを実行するためのワークフロー

ステップ	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。 スナップショット自体は、コネクタによる binlog の位置やテーブルスキーマの読み取りを妨害する可能性のある DDL を他のクライアントが適用しないように防ぐことはありません。コネクタは binlog の位置を読み取る間にグローバル読み取りロックを保持し、後のステップで説明するように、ロックを解除します。
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	コネクタが変更をキャプチャーするように設定されたデータベースと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。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	該当する場合は、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。
7	データベーステーブルをスキャンします。コネクタは、行ごとに、 CREATE イベントを関係するテーブル固有の Kafka トピックに出力します。
8	トランザクションをコミットします。
9	コネクタオフセットの完了済みスナップショットを記録します。

コネクタの再起動

最初のスナップショット の実行中にコネクタが失敗または停止したり、再分散された場合、コネクタの再起動後に新しいスナップショットが実行されます。この **最初のスナップショット** が完了すると、Debezium MySQL コネクタは 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。

コネクタ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 コネクタのトラブルシューティングに関する詳細は、[問題が発生したときの動作](#) を参照してください。

グローバル読み取りロックが許可されない

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。これには、Debezium コネクターのデータベースユーザーに **LOCK TABLES** 権限が必要になります。

表5.3 テーブルレベルロックを使用して最初のスナップショットを実行するためのワークフロー

ステップ	アクション
1	テーブルレベルロックを取得します。
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	データベースとテーブルの名前を読み取り、選別します。
4	現在の binlog の位置を読み取ります。
5	コネクターが変更をキャプチャーするように設定されたデータベースとテーブルのスキーマを読み取ります。
6	該当する場合は、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。
7	データベーステーブルをスキャンします。コネクターは、行ごとに、 CREATE イベントを関係するテーブル固有の Kafka トピックに出力します。
8	トランザクションをコミットします。
9	テーブルレベルロックを解除します。
10	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。

5.1.4.1. アドホックスナップショット

デフォルトでは、コネクターは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクターではスナップショットプロセスは繰り返し処理されません。コネクターがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクターを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。テーブルデータを再キャプチャーするメカニズムを提供するため、Debezium にはアドホックスナップショットを実行するオプションがあります。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクター設定は、異なるテーブルセットをキャプチャーするように変更されます。
- Kafka トピックを削除して、再構築する必要があります。

- 設定エラーや他の問題が原因で、データの破損が発生します。

アドホックと呼ばれるスナップショットを開始することで、以前にスナップショットをキャプチャーしたテーブルのスナップショットを再実行できます。アドホックスナップショットには、[シグナルテーブル](#)を使用する必要があります。シグナルリクエストを Debezium シグナルテーブルに送信して、アドホックスナップショットを開始します。

既存のテーブルのアドホックスナップショットを開始すると、コネクタはテーブルにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、[トピックの自動作成](#)が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックのスナップショットシグナルは、スナップショットに追加するテーブルを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のテーブルのサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のテーブルの内容のサブセットをキャプチャできます。

`execute-snapshot` メッセージをシグナルテーブルに送信してキャプチャーするテーブルを指定します。以下の表で説明されているように、`run-snapshot` シグナルのタイプを `incremental` に設定し、スナップショットに追加するテーブルの名前を指定します。

表5.4 アドホックの `execute-snapshot` シグナルレコードの例

フィールド	デフォルト	値
<code>type</code>	<code>incremental</code>	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 <code>incremental</code> スナップショットのみです。
<code>data-collections</code>	該当なし	スナップショットされるテーブルの完全修飾名にマッチする正規表現を含む配列。名前の形式は <code>signal.data.collection</code> 設定オプションと同じです。
<code>additional-condition</code>	該当なし	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

`execute-snapshot` シグナルタイプのエントリーをシグナルテーブルに追加して、アドホックスナップショットを開始します。コネクタがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各テーブルの開始ポイントおよびエンドポイントとして使用します。テーブルのエントリー数と設定されたチャンクサイズに基づいて、Debezium はテーブルをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、`execute-snapshot` アクションタイプは [増分スナップショット](#) のみをトリガーします。詳細は、[スナップショットの増分](#)を参照してください。

5.1.4.2. 増分スナップショット

スナップショットを柔軟に管理するため、Debezium には [増分スナップショット](#) と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、[Debezium コネクタにシグナルを送信するための](#) Debezium メカニズムに依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各テーブルを段階的にキャプチャーします。スナップショットがキャプチャーするテーブルと、[各チャンクのサイズ](#)を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

増分スナップショットが進むと、Debezium はウォーターマークを使用して進捗を追跡し、キャプチャーする各テーブル行のレコードを管理します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクターはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセスが再開すると、スナップショットは最初からテーブルをキャプチャーするのではなく、停止した時点から開始します。
- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。たとえば、コネクター設定を変更してテーブルを [table.include.list](#) プロパティに追加した後スナップショットを再実行します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各テーブルをプライマリーキー別に分類して、[設定されたチャンクサイズ](#)に基づいてテーブルをチャンクに分割します。チャンクごとに作業し、テーブルの行ごとにチャンクでキャプチャーします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットの作成が進むにつれ、他のプロセスがデータベースへのアクセスを継続し、テーブルレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT**、**UPDATE**、**DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できます。つまり、ストリーミングプロセスは、スナップショットがその行の **READ** イベントが含まれるチャンクをキャプチャーする前に、テーブルの行を変更するイベントを生成する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリーミングされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて入ってきた **READ** イベントと、同じテーブルの行を変更するストリーミングイベント間の競合の解決を容易にするために、Debezium は **スナップショットウィンドウ** と呼ばれるものを使用します。スナップショットウィンドウは、増分スナップショットが指定のテーブルチャンクのデータをキャプチャーしている途中に、間隔を決定します。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は 2 種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャーするスナップショットレコードは、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをテーブルの Kafka トピックに出力します。

コネクタは各スナップショットチャンクにプロセスを繰り返します。

5.1.4.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、[アドホックスナップショットシグナル](#) をソースデータベースのシグナルテーブルに送信することです。

シグナルを SQL **INSERT** クエリーとしてシグナルテーブルに送信します。

Debezium がシグナルテーブルの変更を検出すると、シグナルを読み取り、要求されたスナップショット操作を実行します。

送信するクエリーはスナップショットに追加するテーブルを指定し、必要に応じてスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに追加するテーブルを指定するには、テーブルをリストする **data-collections** 配列またはテーブルの照合に使用する正規表現の配列を指定します。以下に例を示します。

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

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるテーブルの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのテーブルを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **My.Table** のテーブルを含めるには、**"public"."My.Table"** の形式を使用します。

前提条件

- [シグナルが有効になっている](#)。
 - ソースデータベースにシグナリングデータコレクションが存在します。

- シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. SQL クエリーを送信し、アドホック増分スナップショット要求をシグナルテーブルに追加します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) ①
values ('ad-hoc-1', ②
       'execute-snapshot', ③
       '{"data-collections": ["schema1.table1", "schema2.table2"], ④
       "type":"incremental"}, ⑤
       "additional-condition":"color=blue"}'); ⑥
```

コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表5.5 シグナルテーブルに増分スナップショットシグナルを送信する SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリへのログメッセージを特定します。Debezium はこの文字列を使用しません。代わりに、スナップショット作成中に、Debezium は独自の ID 文字列をウォーターマークシグナルとして生成します。
3	execute-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるテーブル名の配列またはテーブル名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。

項目	値	説明
5	incremental	実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 値を指定しない場合には、コネクタは増分スナップショットを実行します。
6	additional-condition	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。 additional-condition パラメーターの詳細は、 additional-condition 付きのアドホック増分スナップショット を参照してください。

additional-condition 付きのアドホック増分スナップショット

スナップショットに、テーブル内のコンテンツのサブセットのみを含める場合は、スナップショットシグナルシグナルに **additional-condition** パラメーターを追加してシグナル要求を変更できます。

一般的なスナップショットの SQL クエリーは、以下の形式を取ります。

```
SELECT * FROM <tableName> ....
```

additional-condition パラメーターを追加して、以下の例のように **WHERE** 条件を SQL クエリーに追加します。

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```

以下の例は、シグナルテーブルに追加の条件を含むアドホック増分スナップショット要求を送信する SQL クエリーを示しています。

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

たとえば、以下の列が含まれる **products** テーブルがあるとします。

- **id** (プライマリーキー)
- **color**
- **quantity**

products テーブルの増分スナップショットに **color=blue** のデータ項目のみを含める場合は、次の SQL ステートメントを使用してスナップショットをトリガーできます。

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

additional-condition パラメーターを使用すると、列が2つ以上となる条件を指定することもできます。たとえば、前述の例の **products** テーブルを使用して、**color=blue** および **quantity>10** だけに一致するアイテムのみのデータが含まれる増分スナップショットをトリガーするクエリーを送信できます。

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-snapshot',
{"data-collections": ["schema1.products"], "type": "incremental", "additional-condition": "color=blue AND
quantity>10"});
```

以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 シグナルテーブルに送信する SQL クエリーでの type 値の指定は任意です。 値を指定しない場合には、コネクターは増分スナップショットを実行します。
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

5.1.4.4. 増分スナップショットの停止

ソースデータベースのテーブルにシグナルを送信して、増分スナップショットを停止することもできます。SQL **INSERT** クエリーを送信して、停止スナップショットシグナルをテーブルに送信します。

Debezium はシグナルテーブルの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのテーブルを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. SQL クエリーを送信して、シグナリングテーブルへのアドホックインクリメンタル スナップショットを停止します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
      'stop-snapshot', 3
      '{"data-collections": ["schema1.table1", "schema2.table2"], 4
      "type": "incremental"}'); 5
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表5.6 シグナリングテーブルに増分スナップショット停止信号を送信するための SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。この文字列を使用して、シグナルテーブルのエントリーへのログメッセージを特定します。Debezium はこの文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。

項目	値	説明
4	data-collections	シグナルの data フィールドのオプションコンポーネントで、スナップショットから削除するテーブル名の配列またはテーブル名とマッチする正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

5.1.5. Debezium MySQL 変更イベントレコードを受信する Kafka トピックのデフォルト名

デフォルトでは、MySQL コネクターは、テーブルで発生するすべての **INSERT**、**UPDATE**、**DELETE** 操作の変更イベントを、そのテーブルに固有の単一の Apache Kafka トピックに書き込みます。

コネクターは以下の規則を使用して変更イベントトピックに名前を付けます。

topicPrefix.databaseName.tableName

fulfillment はトピック接頭辞、**inventory** はデータベース名で、データベースに **orders**、**customers**、および **products** という名前のテーブルが含まれるとします。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 トピックにイベントを出力します。

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

以下のリストは、デフォルト名のコンポーネントの定義を示しています。

topicPrefix

topic.prefix コネクター設定プロパティで指定されたトピック接頭辞。

schemaName

操作が発生したスキーマの名前。

tableName

操作が発生したテーブルの名前。

コネクターは同様の命名規則を適用して、内部データベーススキーマの履歴トピック (**スキーマ変更トピック** と **トランザクションメタデータトピック**) にラベルを付けます。

デフォルトのトピック名が要件を満たさない場合は、カスタムトピック名を設定できます。カスタムトピック名を設定するには、論理トピックルーティング SMT に正規表現を指定します。論理トピックルーティング SMT を使用してトピックの命名をカスタマイズする方法は、**トピックルーティング** を参照してください。

トランザクションメタデータ

Debezium は、トランザクション境界を表し、データ変更イベントメッセージをエンリッチするイベントを生成できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクタのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクタをデプロイする前に発生するトランザクションのメタデータは利用できません。

Debezium は、すべてのトランザクションで **BEGIN** および **END** 区切り文字のトランザクション境界イベントを生成します。トランザクション境界イベントには以下のフィールドが含まれます。

status

BEGIN または **END**

id

一意のトランザクション識別子の文字列表現。

ts_ms

データソースでのトランザクション境界イベント (**BEGIN** または **END** イベント) の時間。もしデータソースが Debezium にイベント時間を提供しないなら、このフィールドは代わりに Debezium がイベントを処理する時間を表します。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

data_collection と **event_count** 要素のペアの配列。これは、コネクタがデータコレクションから発信された変更に対して出力するイベントの数を示します。

例

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

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

```

    }
  ]
}

```

topic.transaction オプションで上書きされない限り、コネクターはトランザクションイベントを **<topic.prefix>.transaction** トピックに出力します。

変更データイベントのエンリッチメント

トランザクションメタデータを有効にすると、データメッセージ **Envelope** は新しい **transaction** フィールドでエンリッチされます。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下は、メッセージの例になります。

```

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

```

GTID が有効ではないシステムの場合は、binlog のファイル名と binlog の位置の組み合わせを使用してトランザクション識別子が作成されます。たとえば、トランザクション BEGIN イベントに対応する binlog のファイル名と位置が `mysql-bin.000002` および `1913` の場合には、Debezium が構築したトランザクション識別子は **file=mysql-bin.000002,pos=1913** になります。

5.2. DEBEZIUM MYSQL コネクターのデータ変更イベントの説明

Debezium MySQL コネクターは、行レベルの **INSERT**、**UPDATE**、および **DELETE** 操作ごとにデータ変更イベントを生成します。各イベントにはキーと値が含まれます。キーと値の構造は、変更されたテーブルによって異なります。

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** を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があり、コンシュー

マーによる処理が困難になることがあります。これに対応するために、各イベントにはコンテンツのスキーマが含まれます。スキーマレジストリーを使用している場合は、コンシューマーがレジストリーからスキーマを取得するために使用できるスキーマ ID が含まれます。これにより、各イベントが自己完結型になります。

以下のスケルトン JSON は、変更イベントの基本となる 4 つの部分を示しています。ただし、アプリケーションで使用するために選択した Kafka Connect コンバーターの設定方法によって、変更イベントのこれら 4 部分の表現が決定されます。**schema** フィールドは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。同様に、イベントキーおよびイベントペイロードは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。JSON コンバーターを使用し、変更イベントの基本となる 4 つの部分すべてを生成するように設定すると、変更イベントの構造は次のようになります。

```
{
  "schema": { ❶
    ...
  },
  "payload": { ❷
    ...
  },
  "schema": { ❸
    ...
  },
  "payload": { ❹
    ...
  },
}
```

表5.7 変更イベントの基本内容の概要

項目	フィールド名	説明
1	schema	最初の schema フィールドはイベントキーの一部です。イベントキーの payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、最初の schema フィールドは、変更されたテーブルのプライマリーキーの構造、またはテーブルにプライマリーキーがない場合は変更されたテーブルの一意キーの構造を記述します。 message.key.columns コネクタ設定プロパティを設定すると、テーブルのプライマリーキーをオーバーライドできます。この場合、最初の schema フィールドはそのプロパティによって識別されるキーの構造を記述します。
2	payload	最初の payload フィールドはイベントキーの一部です。前述の schema フィールドによって記述された構造を持ち、変更された行のキーが含まれます。
3	schema	2 つ目の schema フィールドはイベント値の一部です。イベント値の payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、2 つ目の schema は変更された行の構造を記述します。通常、このスキーマには入れ子になったスキーマが含まれます。

項目	フィールド名	説明
4	payload	2つ目の payload フィールドはイベント値の一部です。前述の schema フィールドによって記述された構造を持ち、変更された行の実際のデータが含まれます。

デフォルトでは、コネクターによって、変更イベントレコードがイベントの元のテーブルと同じ名前を持つトピックにストリーミングされます。[トピック名](#) を参照してください。



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。つまり、論理サーバー名はアルファベットまたはアンダースコア (a-z、A-Z、または _) で始まる必要があります。論理サーバー名の残りの各文字と、データベース名とテーブル名の各文字は、アルファベット、数字、またはアンダースコア (a-z、A-Z、0-9、または _) でなければなりません。無効な文字がある場合は、アンダースコアに置き換えられます。

論理サーバー名、データベース名、またはテーブル名に無効な文字が含まれ、名前を区別する唯一の文字が無効であると、無効な文字はすべてアンダースコアに置き換えられるため、予期せぬ競合が発生する可能性があります。

詳細は以下を参照してください。

- [「Debezium MySQL 変更イベントのキー」](#)
- [「Debezium MySQL 変更イベントの値」](#)

5.2.1. Debezium MySQL 変更イベントのキー

変更イベントのキーには、変更されたテーブルのキーのスキーマと、変更された行の実際のキーのスキーマが含まれます。スキーマとそれに対応するペイロードの両方には、コネクターによってイベントが作成された時点において、変更されたテーブルの **PRIMARY KEY** (または一意の制約) に存在した各列のフィールドが含まれます。

以下の **customers** テーブルについて考えてみましょう。この後に、このテーブルの変更イベントキーの例を示します。

```
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;
```

customers テーブルへの変更をキャプチャーする変更イベントのすべてに、イベントキースキーマがあります。**customers** テーブルに前述の定義がある限り、**customers** テーブルへの変更をキャプチャーする変更イベントのキー構造はすべて以下ようになります。JSON では、以下のようになります。

す。

```
{
  "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
  }
}
```

表5.8 変更イベントキーの説明

項目	フィールド名	説明
1	schema	キーのスキーマ部分は、キーの payload 部分の内容を記述する Kafka Connect スキーマを指定します。
2	mysql-server-1.inventory.customers.Key	<p>キーのペイロードの構造を定義するスキーマの名前。このスキーマは、変更されたテーブルのプライマリーキーの構造を記述します。キースキーマ名の形式は connector-name.database-name.table-name.Key です。この例では、以下のようになります。</p> <ul style="list-style-type: none"> ● mysql-server-1 はこのイベントを生成したコネクタの名前です。 ● inventory は変更されたテーブルが含まれるデータベースです。 ● customers は更新されたテーブルです。
3	任意	イベントキーの payload フィールドに値が含まれる必要があるかどうかを示します。この例では、キーのペイロードに値が必要です。テーブルにプライマリーキーがない場合は、キーの payload フィールドの値は任意です。
4	fields	各フィールドの名前、型、および必要かどうかなど、 payload で想定される各フィールドを指定します。
5	payload	この変更イベントが生成された行のキーが含まれます。この例では、キーには値が 1001 の1つの id フィールドが含まれます。

5.2.2. Debezium MySQL 変更イベントの値

変更イベントの値はキーよりも若干複雑です。キーと同様に、値には **schema** セクションと **payload** セクションがあります。**schema** セクションには、入れ子のフィールドを含む、**Envelope** セクションの **payload** 構造を記述するスキーマが含まれています。データを作成、更新、または削除する操作のすべての変更イベントには、Envelope 構造を持つ値 payload があります。

変更イベントキーの例を紹介するために使用した、同じサンプルテーブルについて考えてみましょう。

```
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;
```

このテーブルへの変更に対する変更イベントの値部分には以下について記述されています。

- **create** イベント
- **update** イベント
- **プライマリーキーの更新**
- **削除** イベント
- **廃棄 (tombstone)** イベント
- **truncate** イベント

作成 イベント

以下の例は、**customers** テーブルにデータを作成する操作に対して、コネクターによって生成される変更イベントの値の部分を示しています。

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



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

```

    "optional": false,
    "name": "io.debezium.connector.mysql.Source", 3
    "field": "source"
  },
  {
    "type": "string",
    "optional": false,
    "field": "op"
  },
  {
    "type": "int64",
    "optional": true,
    "field": "ts_ms"
  }
],
"optional": false,
"name": "mysql-server-1.inventory.customers.Envelope" 4
},
"payload": { 5
  "op": "c", 6
  "ts_ms": 1465491411815, 7
  "before": null, 8
  "after": { 9
    "id": 1004,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": { 10
    "version": "2.1.4.Final",
    "connector": "mysql",
    "name": "mysql-server-1",
    "ts_ms": 0,
    "snapshot": false,
    "db": "inventory",
    "table": "customers",
    "server_id": 0,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 154,
    "row": 0,
    "thread": 7,
    "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
  }
}
}
}

```

表5.9 作成 イベント値フィールドの説明

項目	フィールド名	説明
----	--------	----

項目	フィールド名	説明
1	schema	値のペイロードの構造を記述する、値のスキーマ。変更イベントの値スキーマは、コネクターが特定のテーブルに生成するすべての変更イベントで同じになります。
2	name	<p>スキーマ セクションで、各 name フィールドは、値のペイロードのフィールドに対するスキーマを指定します。</p> <p>mysql-server-1.inventory.customers.Value は、before と after ペイロードのスキーマです。このスキーマは customers テーブルに固有です。</p> <p>before および after フィールドのスキーマ名は logicalName.tableName.Value の形式で、スキーマ名がデータベースで一意になるようにします。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 スキーマは、それぞれ独自に進化し、独自の履歴を持つことになります。</p>
3	name	io.debezium.connector.mysql.Source は、ペイロードの source フィールドのスキーマです。このスキーマは MySQL コネクターに固有です。コネクターは生成するすべてのイベントにこれを使用します。
4	name	mysql-server-1.inventory.customers.Envelope は、ペイロードの全体的な構造のスキーマで、 dbserver1 はコネクター名、 inventory はデータベース、 customers はテーブルを指します。
5	payload	<p>値の実際のデータ。これは、変更イベントが提供する情報です。</p> <p>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。これは、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です。しかし、Avro コンバーター を使用すると、コネクターが Kafka トピックにストリーミングするメッセージのサイズを大幅に小さくすることができます。</p>
6	op	<p>コネクターによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、c は操作によって行が作成されたことを示しています。有効な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (読み取り、スナップショットのみに適用)

項目	フィールド名	説明
7	ts_ms	<p>コネクタ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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>
8	before	<p>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。この例のように、op フィールドが create (作成) の c である場合、この変更イベントは新しい内容に対するものであるため、before は null になります。</p>
9	after	<p>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。この例では、after フィールドには、新しい行の id、first_name、last_name、および email 列の値が含まれます。</p>
10	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、イベントの発生元、イベントの発生順序、およびイベントが同じトランザクションの一部であるかどうかなど、このイベントと他のイベントを比較するために使用できる情報が含まれています。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ名 ● イベントが記録された binlog 名 ● binlog の位置 ● イベント内の行 ● イベントがスナップショットの一部であるか ● 新しい行が含まれるデータベースおよびテーブルの名前 ● イベントを作成した MySQL スレッドの ID (スナップショット以外) ● MySQL サーバー ID (利用可能な場合) ● データベースに変更が加えられた時点のタイムスタンプ <p>binlog_rows_query_log_events MySQL 設定オプションが有効で、コネクタ設定 include.query プロパティが有効な場合、source フィールドは、変更イベントの起因となった元の SQL ステートメントが含まれる query フィールドも提供します。</p>

更新イベント

サンプル **customers** テーブルにある更新の変更イベントの値には、そのテーブルの **作成** イベントと同じスキーマがあります。同様に、イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。以下は、コネクタによって **customers** テーブルでの更新に生成されるイベントの変更イベント値の例になります。

```

{
  "schema": { ... },
  "payload": {
    "before": { ❶
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { ❷
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { ❸
      "version": "2.1.4.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581029100,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "u", ❹
    "ts_ms": 1465581029523 ❺
  }
}

```

表5.10 更新 イベント値フィールドの説明

項目	フィールド名	説明
1	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。更新 イベント値の before フィールドには、各テーブル列のフィールドと、データベースのコミット前にその列にあった値が含まれます。この例では、 first_name 値は Anne です。
2	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。 before と after の構造を比較すると、この行への更新内容を判断できます。この例では、 first_name 値は Anne Marie です。

項目	フィールド名	説明
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。source フィールド構造には create イベントと同じフィールドがありますが、一部の値が異なります。たとえば、更新 イベントは binlog の異なる位置から発生します。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ名 ● イベントが記録された binlog 名 ● binlog の位置 ● イベント内の行 ● イベントがスナップショットの一部であるか ● 更新された行が含まれるデータベースおよびテーブルの名前 ● イベントを作成した MySQL スレッドの ID (スナップショット以外) ● MySQL サーバー ID (利用可能な場合) ● データベースに変更が加えられた時点のタイムスタンプ <p>binlog_rows_query_log_events MySQL 設定オプションが有効で、コネクタ設定 include.query プロパティが有効な場合、source フィールドは、変更イベントの起因となった元の SQL ステートメントが含まれる query フィールドも提供します。</p>
4	op	<p>操作の型を記述する必須の文字列。更新 イベントの値では、op フィールドの値は u で、更新によってこの行が変更したことを示します。</p>
5	ts_ms	<p>コネクタ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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>



注記

行のプライマリーキー/一意キーの列を更新すると、行のキーの値が変更されます。キーが変更されると、3つのイベントが Debezium によって出力されます。3つのイベントとは、**DELETE** イベント、行の古いキーを持つ **廃棄 (tombstone)**、およびそれに続く行の新しいキーを持つイベントです。詳細は次のセクションで説明します。

プライマリーキーの更新

行のプライマリーキーフィールドを変更する **UPDATE** 操作は、プライマリーキーの変更と呼ばれます。プライマリーキーの変更では、**UPDATE** イベントレコードの代わりにコネクタが古いキーの **DELETE** イベントレコードと、新しい (更新された) キーの **CREATE** イベントレコードを出力します。

これらのイベントには通常の構造と内容があり、イベントごとにプライマリーキーの変更に関連するメッセージヘッダーがあります。

- **DELETE** イベントレコードには、メッセージヘッダーとして `__debezium.newkey` が含まれます。このヘッダーの値は、更新された行の新しいプライマリーキーです。
- **CREATE** イベントレコードには、メッセージヘッダーとして `__debezium.oldkey` が含まれます。このヘッダーの値は、更新された行にあった以前の (古い) プライマリーキーです。

delete イベント

削除 変更イベントの値は、同じテーブルの **作成** および **更新** イベントと同じ **schema** の部分になります。サンプル **customers** テーブルの **削除** イベントの **payload** 部分は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, 2
    "source": { 3
      "version": "2.1.4.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581902300,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 805,
      "row": 0,
      "thread": 7,
      "query": "DELETE FROM customers WHERE id=1004"
    },
    "op": "d", 4
    "ts_ms": 1465581902461 5
  }
}
```

表5.11 削除 イベント値フィールドの説明

項目	フィールド名	説明
1	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。 削除 イベント値の before フィールドには、データベースのコミットで削除される前に行にあった値が含まれます。

項目	フィールド名	説明
2	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。 削除 イベント値の 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	<p>イベントのソースメタデータを記述する必須のフィールド。削除 イベント値の source フィールド構造は、同じテーブルの 作成 および 更新 イベントと同じになります。多くの source フィールドの値も同じです。削除 イベント値では、ts_ms および pos フィールドの値や、その他の値が変更された可能性があります。ただし、削除 イベント値の source フィールドは、同じメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ名 ● イベントが記録された binlog 名 ● binlog の位置 ● イベント内の行 ● イベントがスナップショットの一部であるか ● 更新された行が含まれるデータベースおよびテーブルの名前 ● イベントを作成した MySQL スレッドの ID (スナップショット以外) ● MySQL サーバー ID (利用可能な場合) ● データベースに変更が加えられた時点のタイムスタンプ <p>binlog_rows_query_log_events MySQL 設定オプションが有効で、コネクタ設定 include.query プロパティが有効な場合、source フィールドは、変更イベントの起因となった元の SQL ステートメントが含まれる query フィールドも提供します。</p>
4	op	操作の型を記述する必須の文字列。 op フィールドの値は d で、行が削除されたことを示します。
5	ts_ms	<p>コネクタ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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

削除 変更イベントレコードは、この行の削除を処理するために必要な情報を持つコンシューマーを提供します。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 コネクタイベントは、[Kafka の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。ログコンパクションにより、少なくとも各キーの最新のメッセージが保持される限り、一部の古いメッ

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。

廃棄 (tombstone) イベント

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、**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、Kafka が同じキーを持つすべてのメッセージを削除するには、メッセージの値が **null** である必要があります。これを可能にするために、Debezium の MySQL コネクターは **削除** イベントを出力した後に、**null** 値以外で同じキーを持つ特別な廃棄 (tombstone) イベントを出力します。

切り捨て (truncate) イベント

切り捨て (truncate) 変更イベントは、テーブルが切り捨てられていることを伝えます。この場合のメッセージキーは **null** で、メッセージの値は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "source": { ❶
      "version": "2.1.4.Final",
      "name": "mysql-server-1",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 1465581029100,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 223344,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 484,
      "row": 0,
      "thread": 7,
      "query": "UPDATE customers SET first_name='Anne Marie' WHERE id=1004"
    },
    "op": "t", ❷
    "ts_ms": 1465581029523 ❸
  }
}
```

表5.12 切り捨て (truncate) イベント値フィールドの説明

項目	フィールド名	説明
----	--------	----

項目	フィールド名	説明
1	source	<p>イベントのソースメタデータを記述する必須のフィールド。切り捨て (truncate) イベント値の source フィールド構造は、同じテーブルの 作成、更新、および削除 イベントと同じで、以下のメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ型および名前 ● イベントが記録された binlog 名 ● binlog の位置 ● イベント内の行 ● イベントがスナップショットの一部であるか ● データベースおよびテーブルの名前 ● イベントを省略した MySQL スレッドの ID (スナップショット以外) ● MySQL サーバー ID (利用可能な場合) ● データベースに変更が加えられた時点のタイムスタンプ
2	op	<p>操作の型を記述する必須の文字列。op フィールドの値は t で、このテーブルが切り捨てされたことを示します。</p>
3	ts_ms	<p>コネクタ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。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。</p> <p>+ source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

1つの **TRUNCATE** ステートメントが複数のテーブルに適用された場合、切り捨てられたテーブルごとに1つの**切り捨て (truncate)** 変更イベントレコードが出力されます。

切り捨て (truncate) イベントは、テーブル全体に加えた変更を表し、メッセージキーを持たないので、単一のパーティションを持つトピックを使用しない限り、テーブルに関する変更イベント (**作成、更新** など) とそのテーブルの **切り捨て (truncate)** イベントの順番は保証されません。たとえば、これらのイベントが異なるパーティションから読み取られる場合、コンシューマーは **更新** イベントを **切り捨て (truncate)** イベントの後でのみ受け取る可能性があります。

5.3. DEBEZIUM MYSQL コネクタによるデータ型のマッピング方法

Debezium MySQL コネクタは、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 データ型は、イベントの値を表す方法を指定します。

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。

コネクターは MySQL データ型を **リテラル型** および **セマンティック型** の両方にマップできます。

- **リテラル型**: Kafka Connect スキーマタイプ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。

デフォルトのデータ型変換が要件に合わない場合は、コネクター用の **カスタムコンバータの作成** が可能です。

詳細は以下を参照してください。

- [基本型](#)
- [時間型](#)
- [10 進数型](#)
- [ブール値](#)
- [空間型](#)

基本型

以下の表は、コネクターによる基本的な MySQL データ型のマッピング方法を示しています。

表5.13 基本型のマッピングの説明

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
BIT(>1)	BYTES	io.debezium.data.Bits length 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。たとえば、 n はビットです。 バイト数 = 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

MySQL 型	リテラル型	セマンティック型
REAL[(M,D)]	FLOAT32	該当なし
FLOAT[(P)]	FLOAT32 または FLOAT64	精度は、ストレージサイズを決定するためにのみ使用されます。0 から 23 までの精度 P は、4 バイトの単精度 FLOAT32 列になります。24 から 53 までの精度 P は、8 バイトの倍精度 FLOAT64 列になります。
FLOAT(M,D)	FLOAT64	MySQL 8.0.17 の時点では、非標準の FLOAT (M,D) および DOUBLE (M,D) 構文は非推奨となり、そのサポートが今後の MySQL バージョンで削除される予定です。 FLOAT64 をデフォルトとして設定します。
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 または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。
VARBINARY(M)	BYTES または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。
TINYBLOB	BYTES または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。
TINYTEXT	STRING	該当なし

MySQL 型	リテラル型	セマンティック型
BLOB	BYTES または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。 サイズが最大 2GB の値のみがサポートされます。Claim Check パターンを使用して、大きな列の値を外部化することが推奨されます。
TEXT	STRING	n/a 2GB までのサイズの値のみがサポートされています。Claim Check パターンを使用して、大きな列の値を外部化することが推奨されます。
MEDIUMBLOB	BYTES または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。
MEDIUMTEXT	STRING	該当なし
LOBLOB	BYTES または STRING	該当なし binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。 サイズが最大 2GB の値のみがサポートされます。Claim Check パターンを使用して、大きな列の値を外部化することが推奨されます。
LONGTEXT	STRING	n/a 2GB までのサイズの値のみがサポートされています。Claim Check パターンを使用して、大きな列の値を外部化することが推奨されます。
JSON	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。
ENUM	STRING	io.debezium.data.Enum allowed 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。

MySQL 型	リテラル型	セマンティック型
SET	STRING	io.debezium.data.EnumSet allowed 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 マイクロ秒の精度を持つ ISO 8601 形式。MySQL では、M を 0-6 の範囲にすることができます。

時間型

TIMESTAMP データ型を除き、MySQL の時間型は **time.precision.mode** コネクタ設定プロパティの値によって異なります。デフォルト値が **CURRENT_TIMESTAMP** または **NOW** として指定される **TIMESTAMP** 列では、Kafka Connect スキーマのデフォルト値として値 **1970-01-01 00:00:00** が使用されます。

MySQL では、ゼロの値は null よりも優先されることがあるため、**DATE**、**DATETIME**、および **TIMESTAMP** 列にゼロの値を使用できます。MySQL コネクタは、列定義で null 値が許可される場合はゼロの値を null 値として表し、列で null 値が許可されない場合はエポック日として表します。

タイムゾーンのない時間型

DATETIME 型は、2018-01-13 09:48:27 のようにローカルの日時を表します。タイムゾーンの情報はありません。このような列は、UTC を使用して列の精度に基づいてエポックミリ秒またはマイクロ秒に変換されます。**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。これは、書き込み時に MySQL によってサーバー (またはセッション) の現在のタイムゾーンから UTC に変換され、値を読み戻すときに UTC からサーバー (またはセッション) の現在のタイムゾーンに変換されます。以下に例を示します。

- 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。

このような列は、サーバー (またはセッション) の現在のタイムゾーンに基づいて、UTC の同等の **io.debezium.time.ZonedTimestamp** に変換されます。タイムゾーンは、デフォルトでサーバーからクエリーされます。これに失敗した場合は、データベース **connectionTimeZone** MySQL 設定オプションで明示的に指定される必要があります。たとえば、データベースのタイムゾーン (グローバルなタイムゾーンまたは **connectionTimeZone** オプションを使用してコネクタのために設定) が **America/Los_Angeles** である場合、値 **2018-06-20T13:37:03Z** を持つ **ZonedTimestamp** によって **TIMESTAMP** 値の **2018-06-20 06:37:03** が表されます。

Kafka Connect および Debezium を実行している JVM のタイムゾーンは、これらの変換には影響しません。

時間値に関連するプロパティの詳細は、[MySQL コネクタ設定プロパティ](#) のドキュメントを参照してください。

time.precision.mode=adaptive_time_microseconds(default)

MySQL コネクタは、イベントがデータベースの値を正確に表すようにするため、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判断します。すべての時間フィールドはマイクロ秒単位です。正しくキャプチャーされる **TIME** フィールドの値は、範囲が **00:00:00.000000** から

23:59:59.999999 までの正の値です。

表5.14 time.precision.mode=adaptive_time_microseconds の場合のマッピング

MySQL 型	リテラル型	セマンティック型
DATE	INT32	io.debezium.time.Date エポックからの日数を表します。
TIME[(M)]	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。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 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。
DATETIME(4), DATETIME(5), DATETIME(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。

time.precision.mode=connect

MySQL コネクターは定義された 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。**00:00:00.000** から **23:59:59.999** までの値のみを処理できます。テーブルの **time.precision.mode=connect** の値が、必ずサポートされる範囲内になるようにすることができます。場合のみ、**TIME** を設定します。**connect** 設定は、今後の Debezium バージョンで削除される予定です。

表5.15 time.precision.mode=connect の場合のマッピング

MySQL 型	リテラル型	セマンティック型
DATE	INT32	org.apache.kafka.connect.data.Date エポックからの日数を表します。
TIME[(M)]	INT64	org.apache.kafka.connect.data.Time 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 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。

10 進数型

Debezium コネクタは、[decimal.handling.mode](#) コネクタ設定プロパティの設定にしたがって 10 進数を処理します。

`decimal.handling.mode=precise`

表5.16 `decimal.handling.mode=precise` の場合のマッピング

MySQL 型	リテラル型	セマンティック型
<code>NUMERIC[(M[,D])]</code>	<code>BYTES</code>	<code>org.apache.kafka.connect.data.Decimal</code> <code>scale</code> 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。
<code>DECIMAL[(M[,D])]</code>	<code>BYTES</code>	<code>org.apache.kafka.connect.data.Decimal</code> <code>scale</code> 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`

表5.17 `decimal.handling.mode=double` の場合のマッピング

MySQL 型	リテラル型	セマンティック型
<code>NUMERIC[(M[,D])]</code>	<code>FLOAT64</code>	該当なし
<code>DECIMAL[(M[,D])]</code>	<code>FLOAT64</code>	該当なし

`decimal.handling.mode=string`

表5.18 `decimal.handling.mode=string` の場合のマッピング

MySQL 型	リテラル型	セマンティック型
<code>NUMERIC[(M[,D])]</code>	<code>STRING</code>	該当なし
<code>DECIMAL[(M[,D])]</code>	<code>STRING</code>	該当なし

ブール値

MySQL は、特定の方法で **BOOLEAN** の値を内部で処理します。**BOOLEAN** 列は、内部で **TINYINT(1)** データ型にマッピングされます。ストリーミング中にテーブルが作成されると、Debezium は元の DDL を受信するため、適切な **BOOLEAN** マッピングが使用されます。スナップショットの作成中、Debezium は **SHOW CREATE TABLE** を実行して、**BOOLEAN** と **TINYINT(1)** の両方の列に **TINYINT(1)** を返すテーブル定義を取得します。その後、Debezium は元の型のマッピングを取得する方法はないため、**TINYINT(1)** にマッピングします。

ソース列をブールデータ型に変換できるように、Debezium は **TinyIntOneToBooleanConverter** [カスタムコンバーター](#) を提供しており、以下のいずれかの方法で使用できます。

- すべての **TINYINT(1)** または **TINYINT(1) UNSIGNED** 列を **BOOLEAN** 型にマップします。

- 正規表現のコンマ区切りリストを使用して、列のサブセットを列挙します。このタイプの変換を使用するには、以下の例のように **selector** パラメーターを使用して **converters** 設定プロパティーを設定する必要があります。

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

- 注:MySQL8 では、**SHOW CREATE TABLE** を実行時に **tinyint unsigned** 型の長さが表示されないため、このコンバータは機能しません。新しいオプション **length.checker** はこの問題を解決することができます。デフォルト値は **true** です。**length.checker** を無効にし、以下の例のように、タイプに基づいてすべての列を変換するのではなく、変換が必要な列を **selector** プロパティーに指定します。

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

空間型

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。

表5.19 空間型マッピングの説明

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 :フィールドが2つの構造が含まれます。 <ul style="list-style-type: none"> ● srid (INT32): 構造に保存されたジオメトリオブジェクトの型を定義する、空間参照システム ID。 ● wkb (BYTES): wkb (Well-Known-Binary) 形式でエンコードされたジオメトリ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。

5.4. DEBEZIUM コネクターを実行するための MYSQL の設定

Debezium をインストールおよび実行する前に、一部の MySQL 設定タスクが必要になります。

詳細は以下を参照してください。

- [「Debezium コネクターの MySQL ユーザーの作成」](#)
- [「Debezium の MySQL binlog の有効化」](#)
- [「Debezium の MySQL グローバルトランザクション識別子の有効化」](#)
- [「Debezium の MySQL セッションタイムアウトの設定」](#)
- [「Debezium MySQL コネクターのクエリーログイベントの有効化」](#)

5.4.1. Debezium コネクターの MySQL ユーザーの作成

Debezium MySQL コネクターには MySQL ユーザーアカウントが必要です。この MySQL ユーザーは、Debezium MySQL コネクターが変更をキャプチャー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を持っている必要があります。

前提条件

- MySQL サーバー。
- SQL コマンドの基本知識。

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';
```

以下の表はパーミッションについて説明しています。



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している場合、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、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、[スナップショット](#) 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;
```

表5.20 ユーザーパーミッションの説明

キーワード	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します。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。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。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。

キーワード	説明
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。 <ul style="list-style-type: none"> ● SHOW MASTER STATUS ● SHOW SLAVE STATUS ● SHOW BINARY LOGS これは必ずコネクターに必要です。
ON	パーミッションが適用されるデータベースを指定します。
TO 'user'	パーミッションを付与するユーザーを指定します。
IDENTIFIED BY 'password'	ユーザーの MySQL パスワードを指定します。

5.4.2. 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。
- 適切な MySQL ユーザーの権限。

手順

1. **log-bin** オプションがすでにオンになっているかどうかを確認します。

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

2. **OFF** の場合は、以下に説明するプロパティで MySQL サーバー設定ファイルを設定します。

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

- 再度 binlog の状態をチェックして、変更を確認します。

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

表5.21 MySQL binlog 設定プロパティの説明

プロパティ	説明
server-id	server-id の値は、MySQL クラスター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 コネクターの設定中に、Debezium によって一意のサーバー ID がコネクターに割り当てられます。
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。
binlog_format	binlog-format は ROW または row に設定する必要があります。
binlog_row_image	binlog_row_image は FULL または full に設定する必要があります。
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。実際の環境に見合った値を設定します。 MySQL による binlog ファイルのページ を参照してください。

5.4.3. 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、レプリケーションを単純化し、プライマリーサーバーとレプリカサーバーの一貫性が保たれるかどうかを簡単に確認することができます。

GTID は MySQL 5.6.5 以降で利用できます。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。
- SQL コマンドの基本知識。
- MySQL 設定ファイルへのアクセス。

手順

- gtid_mode** を有効にします。

```
mysql> gtid_mode=ON
```

- 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%';
```

結果

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

表5.22 GTID オプションの説明

オプション	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。 <ul style="list-style-type: none"> ● ON = 有効化 ● OFF = 無効化
enforce_gtid_consistency	トランザクションに安全な方法でログに記録できるステートメントの実行を許可することにより、サーバーが GTID の整合性を強制するかどうかを指定するブール値。GTID を使用する場合に必須です。 <ul style="list-style-type: none"> ● ON = 有効化 ● OFF = 無効化

5.4.4. Debezium の MySQL 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。
- SQL コマンドの基本知識。
- MySQL 設定ファイルへのアクセス。

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>
```

表5.23 MySQL セッションタイムアウトオプションの説明

オプション	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)。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の発生を待つ時間 (秒単位)。詳細は MySQL のドキュメント を参照してください。

5.4.5. Debezium MySQL コネクター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。

このオプションは、MySQL 5.6 以降で利用できます。

前提条件

- MySQL サーバー。
- SQL コマンドの基本知識。
- MySQL 設定ファイルへのアクセス。

手順

- **binlog_rows_query_log_events** を有効にします。

```
mysql> binlog_rows_query_log_events=ON
```

binlog_rows_query_log_events は、binlog エントリーに **SQL** ステートメントが含まれるようにするためのサポートを有効または無効にする値に設定されます。

- **ON** = 有効化
- **OFF** = 無効化

5.4.6. Debezium MySQL コネクターの binlog 行値オプションを検証する

binlog_row_value_options 変数を確認し、値が **PARTIAL_JSON** に設定されていないことを確認してください。このような場合、コネクターは **UPDATE** イベントの消費に失敗する可能性があります。

前提条件

- MySQL サーバー。

- SQL コマンドの基本知識。
- MySQL 設定ファイルへのアクセス。

手順

1. 現在の変数値を確認する

```
mysql> show global variables where variable_name = 'binlog_row_value_options';
```

2. 結果

```
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| binlog_row_value_options |      |
+-----+-----+
```

3. 値が **PARTIAL_JSON** の場合、次のようにしてこの変数の設定を解除します。

```
mysql> set @@global.binlog_row_value_options="" ;
```

5.5. DEBEZIUM MYSQL コネクターのデプロイメント

以下の方法のいずれかを使用して Debezium MySQL コネクターをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタープラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。

関連情報

- [「Debezium MySQL コネクター設定プロパティの説明」](#)

5.5.1. AMQ Streams を使用した MySQL コネクターデプロイメント

Debezium 1.7 以降、Debezium コネクターのデプロイに推奨される方法は、AMQ Streams を使用してコネクタープラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクターアーティファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクターがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクターを開始します。

Kafka Connect イメージのビルド仕様では、デプロイ可能なコネクターを指定できます。各コネクタープラグインに対して、デプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーティファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をデプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクタを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。
- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

5.5.2. AMQ Streams を使用した Debezium MySQL コネクタのデプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクタをデプロイするには、最初にコネクタ用の Kafka Connect イメージをビルドする必要がありました。コネクタを OpenShift にデプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクタプラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクタ定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリまたは別の設定済みの HTTP サーバーから必要なアーティファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのデプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクタを起動します。

前提条件

- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスターが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合

- [ImageStream](#) リソースを新規コンテナイメージを保存するためにクラスターにデプロイします。クラスターの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスターにログインします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例5.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium MySQL コネクタアーカイブです。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクタで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプティング SMT アーカイブと、Debezium コネクタで使用する関連スクリプティングエンジン。SMT アーカイブとスクリプト言語の依存関係はオプションのコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場合にのみ、これらのコンポーネントを追加します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
    type: imagestream ❹
    image: debezium-streams-connect:latest
  plugins: ❺
    - name: debezium-connector-mysql
      artifacts:
        - type: zip ❻
```

```

url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
mysql/2.1.4.Final-redhat-00001/debezium-connector-mysql-2.1.4.Final-redhat-00001-
plugin.zip 7
- type: zip
url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-
connect-converter/2.3.0.Final-redhat-<build-number>/apicurio-registry-distro-connect-
converter-2.3.0.Final-redhat-<build-number>.zip 8
- type: zip
url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip 9
- type: jar
url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-
3.0.11.jar 10
- type: jar
url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
jsr223/3.0.11/groovy-jsr223-3.0.11.jar
- type: jar
url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
json3.0.11/groovy-json-3.0.11.jar

bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

...

```

表5.24 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクタを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクタのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタプラグインに対して、コネクタと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。

項目	説明
6	<p>artifacts.type の値は、artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip、tgz、または jar です。Debezium コネクターアーカイブは、.zip ファイル形式で提供されます。type の値は、url フィールドで参照されるファイルのタイプと一致する必要があります。</p>
7	<p>artifacts.url の値は、コネクターアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。Debezium コネクターアーティファクトは Red Hat リポジトリで入手できます。OpenShift クラスタは指定されたサーバーにアクセスできる必要があります。</p>
8	<p>(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクターが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。</p>
9	<p>(オプション) Debezium コネクターで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベースルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。</p>
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div data-bbox="459 1234 564 1458" style="background-color: #333; color: #fff; padding: 10px; margin-bottom: 10px;"> <p>重要</p> </div> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタープラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクターが失敗します。</p> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>別の方法として、Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスターを起動します。設定に一覧表示されているコネクタアーティファクトはクラスターで利用できます。

4. **KafkaConnector** リソースを作成し、デプロイする各コネクタのインスタンスを定義します。
たとえば、以下の **KafkaConnector** CR を作成し、**mysql-inventory-connector.yaml** として保存します。

例5.2 Debezium コネクタの KafkaConnector カスタムリソースを定義する **mysql-inventory-connector.yaml** ファイル

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-mysql ①
spec:
  class: io.debezium.connector.mysql.MySqlConnector ②
  tasksMax: 1 ③
  config: ④
    schema.history.internal.kafka.bootstrap.servers: debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092
    schema.history.internal.kafka.topic: schema-changes.inventory
    database.hostname: mysql.debezium-mysql.svc.cluster.local ⑤
    database.port: 3306 ⑥
    database.user: debezium ⑦
    database.password: dbz ⑧
    database.server.id: 184054 ⑨
    topic.prefix: inventory-connector-mysql ⑩
    table.include.list: inventory.* ⑪
  ...

```

表5.25 コネクタ設定の説明

項目	説明
1	Kafka Connect クラスターに登録するコネクタの名前。
2	コネクタクラスの名称。
3	同時に動作できるタスクの数。
4	コネクタの設定。
5	ホストデータベースインスタンスのアドレス。

項目	説明
6	データベースインスタンスのポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名前。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。
9	コネクタの一意の ID (数値)。
10	データベースインスタンスまたはクラスターのトピック接頭辞。 指定する名前は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクタから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名前はクラスターのコネクタ間で一意である必要があります。 コネクタを Avro コネクタ と統合する場合、この名前空間は関連する Kafka Connect スキーマの名前や、対応する Avro スキーマの名前空間でも使用されます。
11	コネクタが変更イベントをキャプチャーするテーブルの一覧。

5. 以下のコマンドを実行してコネクタリソースを作成します。

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

```
oc create -n debezium -f mysql-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium MySQL のデプロイメントを確認](#)する準備が整いました。

5.5.3. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium MySQL コネクタのデプロイ

Debezium MySQL コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、以下のカスタムリソース (CR) を作成する必要があります。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium MySQL コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用するのと同じ OpenShift インスタンスに適用します。

前提条件

- MySQL が稼働し、[Debezium コネクタと連携するように MySQL を設定する手順](#) が完了済みである必要があります。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。
- Podman または Docker がインストールされている。
- Debezium コネクタを実行するコンテナを追加する予定のコンテナレジストリー ([quay.io](#) や [docker.io](#) など) でコンテナを作成および管理するアカウントとパーミッションを持っている。

手順

1. Kafka Connect の Debezium MySQL コンテナを作成します。
 - a. [registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12](#) をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-mysql.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium 2
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/2.1.4.Final-redhat-00001/debezium-connector-mysql-2.1.4.Final-redhat-00001-plugin.zip \
&& unzip debezium-connector-mysql-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-mysql-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-mysql.yaml** という名前の Dockerfile を作成します。

- b. 前のステップで作成した **debezium-container-for-mysql.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

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

```
docker build -t debezium-container-for-mysql:latest .
```

上記のコマンドは、**debezium-container-for-mysql** という名前のコンテナイメージを構築します。

- c. カスタムイメージを **quay.io** などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

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

```
docker push <myregistry.io>/debezium-container-for-mysql:latest
```

- d. 新しい Debezium MySQL **KafkaConnect** カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  #...
  image: debezium-container-for-mysql ❷
...

```

項目	説明
1	KafkaConnector リソースはこの Kafka Connect クラスタでコネクターを設定するために使用されることを、 metadata.annotations は Cluster Operator に示します。
2	spec.image は Debezium コネクターを実行するために作成したイメージの名前を指定します。設定された場合、このプロパティによって Cluster Operator の STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE 変数がオーバーライドされます。

- e. 以下のコマンドを入力して、**KafkaConnect** CR を OpenShift Kafka Connect 環境に適用します。

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

このコマンドは、Debezium コネクターを実行するために作成したイメージの名前を指定する Kafka Connect インスタンスを追加します。

2. Debezium MySQL コネクターインスタンスを設定する **KafkaConnector** カスタムリソースを作成します。
通常、コネクター設定プロパティを設定する **.yaml** ファイルに Debezium MySQL コネクターを設定します。コネクター設定は、Debezium に対して、スキーマおよびテーブルのサブ

セットにイベントを生成するよう指示する可能性があり、または機密性の高い、大きすぎる、または不必要な指定の列で Debezium が値を無視、マスク、または切り捨てるようにプロパティを設定する可能性もあります。

以下の例では、ポート **3306** の MySQL ホスト (**192.168.99.100**) に接続し、**inventory** データベースへの変更をキャプチャーする Debezium コネクタを設定します。**dbserver1** は、サーバーの論理名です。

MySQL inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector-mysql ❶
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 1 ❷
  config: ❸
    database.hostname: mysql ❹
    database.port: 3306
    database.user: debezium
    database.password: dbz
    database.server.id: 184054 ❺
    topic.prefix: inventory-connector-mysql ❻
    table.include.list: inventory ❼
    schema.history.internal.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 ❽
    schema.history.internal.kafka.topic: schema-changes.inventory ❾
```

表5.26 コネクタ設定の説明

項目	説明
1	コネクタの名前。
2	1度に1つのタスクのみが動作する必要があります。MySQL コネクタは MySQL サーバーの binlog を読み取るため、単一のコネクタ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	コネクタの設定。
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の名前です (mysql)。
5	connector の一意 ID。

項目	説明
6	MySQL サーバーまたはクラスターのトピック接頭辞。この名前は、変更イベントレコードを受信するすべての Kafka トピックの接頭辞として使用されます。
7	コネクターは インベントリー テーブルからのみ変更をキャプチャーします。
8	DDL ステートメントをデータベーススキーマ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。再起動時に、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。
9	データベーススキーマ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。

3. Kafka Connect でコネクターインスタンスを作成します。たとえば、**KafkaConnector** リソースを **inventory-connector.yaml** ファイルに保存した場合は、以下のコマンドを実行します。

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

上記のコマンドは **inventory-connector** を登録し、コネクターは **KafkaConnector** CR に定義されている **inventory** データベースに対して実行を開始します。

Debezium MySQL コネクターに設定できる設定プロパティの完全リストは、[MySQL コネクター設定プロパティ](#)を参照してください。

結果

コネクターが起動すると、コネクターが設定された MySQL データベースの **整合性スナップショット**が実行されます。その後、コネクターは行レベルの操作のデータ変更イベントの生成を開始し、変更イベントレコードを Kafka トピックにストリーミングします。

5.5.4. Debezium MySQL コネクターが実行していることの確認

コネクターがエラーなしで正常に起動すると、コネクターがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウンストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクターが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクターのステータスを確認します。
- コネクターがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクターが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクターが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。

- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクターの名前をクリックします (例: **inventory-connector-mysql**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-mysql -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例5.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-mysql
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector

...

Status:
Conditions:
  Last Transition Time: 2021-12-08T17:41:34.897153Z
  Status:              True
  Type:                Ready
Connector Status:
Connector:
  State:  RUNNING
  worker_id: 10.131.1.124:8083
Name:      inventory-connector-mysql
Tasks:
  Id:      0
  State:   RUNNING
  worker_id: 10.131.1.124:8083
```

```

Type:          source
Observed Generation: 1
Tasks Max:     1
Topics:
inventory-connector-mysql.inventory
inventory-connector-mysql.inventory.addresses
inventory-connector-mysql.inventory.customers
inventory-connector-mysql.inventory.geom
inventory-connector-mysql.inventory.orders
inventory-connector-mysql.inventory.products
inventory-connector-mysql.inventory.products_on_hand
Events: <none>

```

2. コネクターによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. **KafkaTopics** リストから確認するトピックの名前をクリックします (例: **inventory-connector-mysql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例5.4 KafkaTopic リソースのステータス

```

NAME                                CLUSTER
PARTITIONS REPLICATION FACTOR  READY
connect-cluster-configs             debezium-kafka-cluster  1
1          True
connect-cluster-offsets             debezium-kafka-cluster  25
1          True
connect-cluster-status               debezium-kafka-cluster  5
1          True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
debezium-kafka-cluster  50      1      True
inventory-connector-mysql--a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster  1        1      True
inventory-connector-mysql.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480      debezium-kafka-cluster
1        1          True
inventory-connector-mysql.inventory.customers---

```

```

9931e04ec92ecc0924f4406af3fdace7545c483b    debezium-kafka-cluster  1
1          True
inventory-connector-mysql.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5    debezium-kafka-cluster
1          1          True
inventory-connector-mysql.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d    debezium-kafka-cluster
1          1          True
inventory-connector-mysql.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef    debezium-kafka-cluster  1
1          True
inventory-connector-mysql.inventory.products_on_hand---
8649e0f17fcc9212e266e31a7aeea4585e5c6b5    debezium-kafka-cluster  1
1          True
schema-changes.inventory                      debezium-kafka-cluster
1          1          True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55    debezium-
kafka-cluster  1          1          True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b    debezium-kafka-cluster  1  1
True

```

3. トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。

```

oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>

```

以下に例を示します。

```

oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-mysql.inventory.products_on_hand

```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory-connector-mysql.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例5.5 Debezium 変更イベントの内容

```

{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},"optional":false,"name":"inventory-
connector-mysql.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"inventory-

```

```

connector-mysql.inventory.products_on_hand.Value", "field": "before"},
{"type": "struct", "fields": [{"type": "int32", "optional": false, "field": "product_id"},
{"type": "int32", "optional": false, "field": "quantity"}], "optional": true, "name": "inventory-
connector-mysql.inventory.products_on_hand.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": "string", "optional": true, "name": "io.debezium.data.Enum", "version": 1, "parameters":
{"allowed": "true,last,false", "default": "false", "field": "snapshot"},
{"type": "string", "optional": false, "field": "db"},
{"type": "string", "optional": true, "field": "sequence"},
{"type": "string", "optional": true, "field": "table"},
{"type": "int64", "optional": false, "field": "server_id"},
{"type": "string", "optional": true, "field": "gtid"}, {"type": "string", "optional": false, "field": "file"},
{"type": "int64", "optional": false, "field": "pos"}, {"type": "int32", "optional": false, "field": "row"},
{"type": "int64", "optional": true, "field": "thread"},
{"type": "string", "optional": true, "field": "query"}], "optional": false, "name": "io.debezium.connecto
r.mysql.Source", "field": "source"}, {"type": "string", "optional": false, "field": "op"},
{"type": "int64", "optional": true, "field": "ts_ms"}, {"type": "struct", "fields":
[{"type": "string", "optional": false, "field": "id"},
{"type": "int64", "optional": false, "field": "total_order"},
{"type": "int64", "optional": false, "field": "data_collection_order"}], "optional": true, "field": "transacti
on"}, {"optional": false, "name": "inventory-connector-
mysql.inventory.products_on_hand.Envelope"}, "payload": {"before": null, "after":
{"product_id": 101, "quantity": 3, "source": {"version": "2.1.4.Final-redhat-
00001", "connector": "mysql", "name": "inventory-connector-
mysql", "ts_ms": 1638985247805, "snapshot": "true", "db": "inventory", "sequence": null, "table": "p
roducts_on_hand", "server_id": 0, "gtid": null, "file": "mysql-
bin.000003", "pos": 156, "row": 0, "thread": null, "query": null}, "op": "r", "ts_ms": 1638985247805, "t
ransaction": null}}

```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

5.5.5. Debezium MySQL コネクタ設定プロパティの説明

Debezium MySQL コネクタには、アプリケーションに適したコネクタ動作を実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。プロパティに関する情報は、以下のように設定されています。

- [必要なコネクタ設定プロパティ](#)
- [高度なコネクタ設定プロパティ](#)
- Debezium がデータベース履歴トピックから読み取るイベントを処理する方法を制御する [データベース履歴コネクタ設定プロパティ](#)。
 - [パススルーデータベーススキーマ履歴プロパティ](#)
- データベースドライバの動作を制御する [パススルーデータベースドライバプロパティ](#)。

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

表5.27 必要な Debezium MySQL コネクタースettingsプロパティ

プロパティ	デフォルト	説明
name	デフォルトなし	コネクタースettingsの一意名。同じ名前です再登録を試みると失敗します。このプロパティはすべての Kafka Connect コネクタースettingsに必要です。
connector.class	デフォルトなし	コネクタースettingsの Java クラスの名前。MySQL コネクタースettingsに常に io.debezium.connector.mysql.MySqlConnector を指定します。
tasks.max	1	このコネクタースettingsのために作成する必要があるタスクの最大数。MySQL コネクタースettings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	デフォルトなし	MySQL 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	3306	MySQL データベースサーバーのポート番号 (整数)。
database.user	デフォルトなし	MySQL データベースサーバーへの接続時に使用する MySQL ユーザーの名前。
database.password	デフォルトなし	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。

プロパティ	デフォルト	説明
topic.prefix	デフォルトなし	<p>Debezium が変更をキャプチャする特定の MySQL データベースサーバー/クラスターの名前空間を提供するトピック 接頭辞。トピック接頭辞は、このコネクターによって発行されたイベントを受信するすべての Kafka トピック名の接頭辞として使用されるため、他のすべてのコネクターで一貫である必要があります。データベースサーバーの論理名には英数字とハイフン、ドット、アンダースコアのみを使用する必要があります。</p> <div data-bbox="815 622 1428 1155" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <p>警告</p>  <p>このプロパティの値を変更しないでください。名前の値を変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。また、コネクターはデータベーススキーマ履歴トピックを復元できません。</p> </div>
database.server.id	デフォルトなし	<p>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貫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。</p>

プロパティ	デフォルト	説明
database.include.list	空の文字列	<p>変更をキャプチャーするデータベースの名前と一致する正規表現のコンマ区切りリスト (任意)。コネクタは、名前が database.include.list がないデータベースの変更をキャプチャーしません。デフォルトでは、コネクタはすべてのデータベースの変更をキャプチャーします。</p> <p>データベースの名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データベースの名前文字列全体に対して照合されます。データベース名に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、database.exclude.list プロパティも設定しないでください。</p>
database.exclude.list	空の文字列	<p>変更をキャプチャーしないデータベースの名前と一致する正規表現のコンマ区切りリスト (任意)。コネクタは、名前が database.exclude.list がないデータベースの変更をキャプチャーします。</p> <p>データベースの名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データベースの名前文字列全体に対して照合されます。データベース名に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、database.include.list プロパティも設定しないでください。</p>
table.include.list	空の文字列	<p>変更をキャプチャー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)。コネクタは、table.include.list に含まれていないテーブルの変更をキャプチャーしません。各識別子の形式は databaseName.tableName です。デフォルトでは、コネクタは変更がキャプチャーされる各データベースのシステムでないすべてのテーブルの変更をキャプチャーします。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.exclude.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
table.exclude.list	空の文字列	<p>変更をキャプチャーしない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)。コネクターは table.exclude.list に含まれていないテーブルの変更をキャプチャーします。各識別子の形式は <code>databaseName.tableName</code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.include.list プロパティも設定しないでください。</p>
column.exclude.list	空の文字列	<p>変更イベントレコード値から除外す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 <code>databaseName.tableName.columnName</code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。このプロパティを設定に含める場合は、column.include.list プロパティも設定しないでください。</p>
column.include.list	空の文字列	<p>変更イベントレコード値に含め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 <code>databaseName.tableName.columnName</code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、column.exclude.list プロパティを設定しないでください。</p>

プロパティ	デフォルト	説明
<code>column.truncate.to.length.chars</code>	該当なし	<p>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)。プロパティ名の 長さ で指定された文字数を超えた場合に、一連の列のデータを切り捨てる場合は、このプロパティを設定します。length を正の整数値に設定します (例: column.truncate.to.20.chars)。</p> <p>列の完全修飾名は、次の形式に従います: databaseName.tableName.columnName列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>
<code>column.mask.with.length.chars</code>	該当なし	<p>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)。一連の列の値をコネクタでマスクする場合 (たとえば、列に機密データが含まれている場合) は、このプロパティを設定します。length を正の整数に設定して、指定された列のデータをプロパティ名の 長さ で指定されたアスタリスク (*) 文字数で置き換えます。指定した列のデータを空の文字列に置き換えるには、長さ を 0 (ゼロ) に設定します。</p> <p>列の完全修飾名は、次の形式に従います: databaseName.tableName.columnName列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>

プロパティ	デフォルト	説明
<p><code>column.mask.hash.hashAlgorithm.with.salt.salt</code>; <code>column.mask.hash.v2.hashAlgorithm.with.salt.salt</code></p>	<p>該当なし</p>	<p>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)。列の完全修飾名の形式は <code><databaseName>.<tableName>.<columnName></code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。作成された変更イベントレコードでは、指定された列の値は仮名に置き換えられます。</p> <p>仮名は、指定された <code>hashAlgorithm</code> と <code>salt</code> を適用すると得られるハッシュ化された値で設定されます。使用されるハッシュ関数に基づいて、参照整合性は保持され、列値は仮名に置き換えられます。サポートされるハッシュ関数は、Java Cryptography Architecture Standard Algorithm Name Documentation の MessageDigest セクション に説明されています。</p> <p>以下の例では、CzQMA0cB5K が無作為に選択された salt になります。</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>必要な場合は、仮名は自動的に列の長さに短縮されます。コネクター設定には、異なるハッシュアルゴリズムと salt を指定する複数のプロパティを含めることができます。</p> <p>使用される <code>hashAlgorithm</code>、選択された <code>salt</code>、および実際のデータセットによっては、結果のデータセットが完全にマスクされない場合があります。</p> <p>値が異なる場所やシステムでハッシュ化されている場合は、ハッシュ化ストラテジーバージョン 2 を使用する必要があります。</p>

プロパティ	デフォルト	説明
<code>column.propagate.source.type</code>	該当なし	<p>列のメタデータを表す追加パラメーターをコネクタに発行させたい列の完全修飾名に一致する、オプションのコンマ区切りの正規表現のリスト。このプロパティが設定されている場合、コネクタは次のフィールドをイベントレコードのスキーマに追加します。</p> <ul style="list-style-type: none">● <code>__debezium.source.column.type</code>● <code>__debezium.source.column.length</code>● <code>__debezium.source.column.scale</code> <p>これらのパラメーターは、列の元の型名と長さ (可変幅型の場合) をそれぞれ伝達します。コネクタがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、次のいずれかの形式に従います: <code>databaseName.tableName.columnName</code>、または <code>databaseName.schemaName.tableName.columnName</code>。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p>

プロパティ	デフォルト	説明
datatype.propagate.source.type	該当なし	<p>データベース内の列に対して定義されているデータ型の完全修飾名を指定する正規表現のオプションのコンマ区切りリスト。このプロパティが設定されている場合、データ型が一致する列に対して、コネクターはスキーマに次の追加フィールドを含むイベントレコードを発行します。</p> <ul style="list-style-type: none"> ● <code>__debezium.source.column.type</code> ● <code>__debezium.source.column.length</code> ● <code>__debezium.source.column.scale</code> <p>これらのパラメーターは、列の元の型名と長さ (可変幅型の場合) をそれぞれ伝達します。コネクターがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名の形式は、<code>databaseName.tableName.typeName</code>、または <code>databaseName.schemaName.tableName.typeName</code> のいずれかになります。データ型の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データ型の名前文字列全体に対して照合されます。式は、型名に存在する可能性のある部分文字列と一致しません。</p> <p>MySQL 固有のデータ型名の一覧は、MySQL のデータ型マッピング を参照してください。</p>
time.precision.mode	adaptive_time_microseconds	<p>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。</p> <p>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をキャプチャーします。</p> <p>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。この組み込み表現は、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。</p>

プロパティ	デフォルト	説明
<code>decimal.handling.mode</code>	<code>precise</code>	<p>コネクタによる DECIMAL および NUMERIC 列の値の処理方法を指定します。</p> <p>precise (デフォルト) はバイナリー形式で変更イベントに表される <code>java.math.BigDecimal</code> 値を使用して正確に表します。</p> <p>double は <code>double</code> 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、簡単に使用できます。</p> <p>string 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。簡単に使用できますが、本来の型に関するセマンティック情報は失われます。</p>
<code>bigint.unsigned.handling.mode</code>	<code>long</code>	<p>変更イベントで BIGINT UNSIGNED 列を表す方法を指定します。可能な設定:</p> <p>long は Java の <code>long</code> を使用して値を表します。これは、正確ではない可能性がありますが、コンシューマーでの使用が簡単です。通常は long が優先設定です。</p> <p>precise は <code>java.math.BigDecimal</code> を使用して値を表し、バイナリー表現と Kafka Connect の <code>org.apache.kafka.connect.data.Decimal</code> 型を使用して変更イベントでエンコードされます。2⁶³ を超える値は long を使用して提供できないため、このような値を使用する場合はこの設定を使用します。</p>
<code>include.schema.changes</code>	<code>true</code>	<p>コネクタ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、コネクタがデータベーススキーマ履歴を内部で記録する方法には依存しません。</p>
<code>include.schema.comments</code>	<code>false</code>	<p>コネクタがメタデータオブジェクトでテーブルおよび列のコメントを解析して公開するかどうかを指定するブール値。このオプションを有効にすると、メモリー使用量に影響を及ぼします。論理スキーマオブジェクトの数およびサイズは、Debezium コネクタによって消費されるメモリーの量に大きく影響し、それぞれに大きな文字列データを追加すると、非常に高価になる可能性があります。</p>

プロパティ	デフォルト	説明
<code>include.query</code>	<code>false</code>	<p>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。</p> <p>このオプションを true に設定した場合は、MySQL の <code>binlog_rows_query_log_events</code> オプションを ON に設定する必要があります。 <code>include.query</code> が true の場合、スナップショットプロセスによって生成されるイベントに対するクエリーは存在しません。</p> <p><code>include.query</code> を true に設定すると、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除外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。そのため、デフォルト設定は false です。</p>
<code>event.deserialization.failure.handling.mode</code>	<code>fail</code>	<p>binlog イベントのデシリアライズ中にコネクターがどのように例外に反応するかを指定します。</p> <p>fail は例外を伝播し、問題のあるイベントとその binlog オフセットを示し、コネクターを停止させます。</p> <p>warn は問題のあるイベントとその binlog オフセットをログに記録し、イベントをスキップします。</p> <p>ignore は問題のあるイベントを渡し、何もログに記録しません。</p>
<code>inconsistent.schema.handling.mode</code>	<code>fail</code>	<p>内部スキーマ表現に存在しないテーブルに関連する binlog イベントに対してコネクターがどのように反応する必要があるかを指定します。つまり、内部表現はデータベースと一貫がありません。</p> <p>fail は例外を出力し、問題のあるイベントとその binlog オフセットを示し、コネクターを停止させます。</p> <p>warn は問題のあるイベントとその binlog オフセットをログに記録し、イベントをスキップします。</p> <p>skip は問題のあるイベントを渡し、何もログに記録しません。</p>
<code>max.batch.size</code>	<code>2048</code>	<p>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。</p>

プロパティ	デフォルト	説明
max.queue.size	8192	ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクターが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクターがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。 max.queue.size の値を、 max.batch.size の値よりも大きくなるように設定します。
max.queue.size.in.bytes	0	ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。 max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、 max.queue.size=1000 、 max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。
poll.interval.ms	500	コネクターがイベントのバッチの処理を開始する前に、新しい変更イベントの発生を待つ期間をミリ秒単位で指定する正の整数値。デフォルトは 500 ミリ秒 (0.5 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。

プロパティ	デフォルト	説明
gtid.source.includes	デフォルトなし	<p>コネクターが MySQL サーバー上のバイナリーログの位置を見つけるために使用する、GTID セット内のソース UUID と一致する正規表現のコンマ区切りリスト。このプロパティが設定されている場合、コネクターは、指定された include パターンのいずれかに一致するソース UUID を持つ GTID 範囲のみを使用します。</p> <p>GTID の値を一致させるために、Debezium は、アンカー 正規表現として指定した正規表現を適用します。つまり、指定された式は UUID 文字列全体に対して照合されます。UUID に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、gtid.source.excludes プロパティも設定しないでください。</p>
gtid.source.excludes	デフォルトなし	<p>コネクターが MySQL サーバー上のバイナリーログの位置を見つけるために使用する GTID セット内のソース UUID と一致する正規表現のコンマ区切りリスト。このプロパティが設定されている場合、コネクターは、指定された exclude パターンのいずれとも一致しないソース UUID を持つ GTID 範囲のみを使用します。</p> <p>GTID の値を一致させるために、Debezium は、アンカー 正規表現として指定した正規表現を適用します。つまり、指定された式は UUID 文字列全体に対して照合されます。UUID に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、gtid.source.includes プロパティも設定しないでください。</p>
tombstones.on.delete	true	<p>削除 イベントの後に廃棄 (tombstone) イベントが続くかどうかを制御します。</p> <p>true: 削除操作は、削除 イベントと後続の破棄 (tombstone) イベントで表されます。</p> <p>false - delete イベントのみが出力されます。</p> <p>log compaction がトピックで有効になっている場合には、ソースレコードの削除後に廃棄 (tombstone) イベントを出力すると (デフォルト動作)、Kafka は削除された行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。</p>

プロパティ	デフォルト	説明
<p>message.key.columns</p>	<p>該当なし</p>	<p>指定のテーブルの Kafka トピックに公開する変更イベントレコードのカスタムメッセージキーを形成するためにコネクタが使用する列を指定する式のリスト。</p> <p>デフォルトでは、Debezium はテーブルのプライマリーキー列を、出力するレコードのメッセージキーとして使用します。デフォルトの代わりに、またはプライマリーキーのないテーブルのキーを指定するには、1つ以上の列をもとにカスタムメッセージキーを設定できます。</p> <p>テーブルにカスタムメッセージキーを設定するには、テーブルを列挙した後、メッセージキーとして使用する列を列挙します。各リストエントリは、</p> <p><fully-qualified_tableName>:<keyColumn>,<keyColumn></p> <p>の形式を取ります。複数の列名をベースにテーブルキーを作成するには、列名の上にコンマを挿入します。</p> <p>各完全修飾テーブル名は、</p> <p><databaseName>.<tableName> の形式の正規表現です。</p> <p>プロパティには複数のテーブルのエントリを含めることができます。セミコロンを使用して、リスト内のテーブルエントリを区切ります。</p> <p>以下の例は、テーブル inventory.customers および purchase.orders:</p> <p>inventory.customers:pk1,pk2; (.*)purchaseorders:pk3,pk4</p> <p>のメッセージキーを設定します。テーブル inventory.customer の場合、列 pk1 と pk2 がメッセージキーとして指定されます。データベースで purchaseorders テーブルは、pk3 および pk4 サーバーの列をメッセージキーとして使用します。</p> <p>カスタムメッセージキーの作成に使用する列の数に制限はありません。ただし、一意の鍵を指定するために必要な最小数を使用することが推奨されます。</p>

プロパティ	デフォルト	説明
binary.handling.mode	bytes	<p>バイナリ列 (例: blob、binary、varbinary) を変更イベントでどのように表すかを指定します。可能な設定:</p> <p>bytes はバイナリデータをバイト配列として表します。</p> <p>base64 は、base64 でエンコードされた文字列としてバイナリデータを表します。</p> <p>base64-url-safe は、base64-url-safe-encoded 文字列としてバイナリデータを表します。</p> <p>hex は、バイナリデータを 16 進数でエンコードされた (base16) 文字列として表します。</p>
schema.name.adjustment.mode	none	<p>コネクターで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能:</p> <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。

高度な MySQL コネクター設定プロパティ

以下の表は、[高度な MySQL コネクタープロパティ](#) について説明しています。これらのプロパティのデフォルト値を変更する必要はほとんどありません。そのため、コネクター設定にデフォルト値を指定する必要はありません。

表5.28 MySQL コネクターの高度な設定プロパティの説明

プロパティ	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。

プロパティ	デフォルト	説明
<code>converters</code>	デフォルトなし	<p>コネクタが使用できる カスタムコンバーター インスタンスのシンボリック名のコンマ区切りリストを列挙します。</p> <p>たとえば、boolean です。</p> <p>このプロパティは、コネクタがカスタムコンバーターを使用できるようにするために必要です。</p> <p>コネクタに設定するコンバーターごとに、コンバーターインターフェイスを実装するクラスの完全修飾名を指定する .type プロパティも追加する必要があります。 .type プロパティでは、以下の形式を使用します。</p> <p><converterSymbolicName>.type</p> <p>以下に例を示します。</p> <pre>boolean.type: io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter</pre> <p>設定されたコンバータの動作をさらに制御したい場合は、1つ以上の設定パラメーターを追加して、コンバータに値を渡すことができます。これらの追加設定パラメーターをコンバータに関連付けるには、パラメーター名の前にコンバータのシンボル名を付けます。</p> <p>たとえば、boolean コンバーターが処理する列のサブセットを指定する selector パラメーターを定義するには、次のプロパティを追加します。</p> <pre>boolean.selector=db1.table1.*, db1.table2.column1</pre>
<code>table.ignore.builtin</code>	true	<p>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 <code>include</code> および <code>exclude</code> 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変更がキャプチャーされないように除外され、システムテーブルに変更が加えられてもイベントは生成されません。</p>

プロパティ	デフォルト	説明
<code>database.ssl.mode</code>	<code>disabled</code>	<p>暗号化された接続を使用するかどうかを指定します。可能な設定:</p> <p>disabled は暗号化されていない接続の使用を指定します。</p> <p>preferred は、サーバーがセキュアな接続に対応している場合は暗号化された接続を確立します。サーバーがセキュアな接続に対応していない場合は、暗号化されていない接続にフォールバックします。</p> <p>required は、暗号化された接続を確立し、何らかの理由で暗号化された接続を確立できない場合は失敗します。.</p> <p>verify_ca は、required と同様に動作しますが、追加でサーバーの TLS 証明書を設定された認証局 (CA) 証明書に対して検証します。サーバー TLS 証明書が有効な CA 証明書と一致しない場合は失敗します。</p> <p>verify_identity は verify_ca のように動作しますが、追加でサーバー証明書がリモート接続のホストと一致するかを検証します。</p>

プロパティ	デフォルト	説明
snapshot.mode	Initial	<p>コネクターの起動時にスナップショットを実行するための基準を指定します。可能な設定:</p> <p>initial - コネクターは、論理サーバー名にオフセットが記録されていない場合にのみスナップショットを実行します。</p> <p>initial_only - 論理サーバー名に対してオフセットが記録されてから停止した場合のみスナップショットを実行します。つまり、binlog から変更イベントを読み取りません。</p> <p>when_needed - コネクターは、必要に応じて、コネクターは起動時にスナップショットを実行します。つまり、オフセットが使用できない場合や、以前に記録されたオフセットがサーバーが利用できない binlog の場所や GTID を指定する場合などです。</p> <p>never - コネクターはスナップショットを使用しません。論理サーバー名での初回起動時に、コネクターは binlog の最初から読み取りします。この動作は注意して設定してください。これは、binlog にデータベースのすべての履歴が含まれることが保証されている場合のみ有効です。</p> <p>schema_only - コネクターはデータではなく、スキーマのスナップショットを実行します。この設定は、トピックにデータの整合性スナップショットが含まれる必要がなく、コネクターの開始以降の変更のみが含まれる必要がある場合に便利です。</p> <p>schema_only_recovery - これは、すでに変更をキャプチャーしているコネクターのリカバリー設定です。この設定により、コネクターを再起動すると、破損または損失したデータベーススキーマ履歴トピックのリカバリーが可能になります。これを定期的に設定して、予想外に増加しているデータベーススキーマ履歴トピックをクリーンアップすることができます。データベーススキーマ履歴トピックは無期限に保持する必要があります。</p>

プロパティ	デフォルト	説明
<p><code>snapshot.locking.mode</code></p>	<p>最小</p>	<p>コネクターがグローバル MySQL 読み込みロックを保持するかどうか、およびその期間を制御します。これにより、コネクターによるスナップショットの実行中にデータベースが更新されないようにします。可能な設定:</p> <p>minimal - コネクターはスナップショットの最初の部分のみグローバル読み取りロックを保持します。その間、データベーススキーマとその他のメタデータを読み取ります。スナップショットの残りの作業では、各テーブルから全行を選択する必要があります。REPEATABLE READ トランザクションを使用すると、コネクターは一貫した方法でこれを行うことができます。これは、グローバル読み取りロックが保持されなくなり、その他の 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該当します。</p> <p>minimal_percona - コネクターは、スナップショットの最初の部分のみ グローバルバックアップロック を保持します。その間、コネクターはデータベーススキーマとその他のメタデータを読み取ります。スナップショットの残りの作業では、各テーブルから全行を選択する必要があります。REPEATABLE READ トランザクションを使用すると、コネクターは一貫した方法でこれを行うことができます。これは、グローバルバックアップロックが保持されなくなり、その他の 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該当します。このモードはテーブルをディスクにフラッシュせず、長時間実行される読み取りによってブロックされず、Percona Server でのみ利用できます。</p> <p>extended - スナップショットの実行中にすべての書き込みをブロックします。MySQL が REPEATABLE READ セマンティックから除外する操作を送信するクライアントがある場合は、この設定を使用します。</p> <p>none - スナップショットの実行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。この設定はすべてのスナップショットモードで許可されますが、スナップショットの実行中にスキーマの変更がない場合に 限り、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 エンジンの動作とは異なります。</p>

プロパティ	デフォルト	説明
<code>snapshot.include.collection.list</code>	<code>table.include.list</code> に指定したすべてのテーブル	<p>スナップショットに含めるテーブルの完全修飾名 (<code><databaseName>.<tableName></code>) と一致する正規表現のコンマ区切りリスト (任意)。指定する項目は、コネクタの <code>table.include.list</code> プロパティで名前を付ける必要があります。このプロパティは、コネクタの <code>snapshot.mode</code> プロパティが <code>never</code> 以外の値に設定されている場合にのみ有効です。</p> <p>このプロパティは増分スナップショットの動作には影響しません。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p>

プロパティ	デフォルト	説明
<p>snapshot.select.statement.overrides</p>	<p>デフォルトなし</p>	<p>スナップショットに追加するテーブル行を指定します。スナップショットにテーブルの行のサブセットのみを含める場合は、プロパティを使用します。このプロパティはスナップショットにのみ影響します。コネクターがログから読み取るイベントには影響しません。</p> <p><databaseName>.<tableName> の形式で完全修飾テーブル名のコンマ区切りリストを指定します。たとえば、</p> <p>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</p> <p>をリスト内の各テーブルに対して、スナップショットを作成する場合には、その他の設定プロパティを追加して、コネクターがテーブルで実行するように SELECT ステートメントを指定します。指定した SELECT ステートメントは、スナップショットに追加するテーブル行のサブセットを決定します。</p> <p>snapshot.select.statement.overrides.<databaseName>.<tableName> の形式を使用して、この SELECT ステートメントプロパティの名前を指定します。例:</p> <p>snapshot.select.statement.overrides.customers.orders.</p> <p>例:</p> <p>スナップショットにソフト削除以外のレコードのみを含める場合は、soft-delete 列 (delete_flag) を含む customers.orders テーブルから、以下のプロパティを追加します。</p> <pre>"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre> <p>作成されるスナップショットでは、コネクターには delete_flag = 0 のレコードのみが含まれます。</p>

プロパティ	デフォルト	説明
min.row.count.to.stream.results	1000	<p>スナップショットの実行中、コネクタは変更をキャプチャするように設定されている各テーブルにクエリを実行します。コネクタは各クエリの結果を使用して、そのテーブルのすべての行のデータが含まれる読み取りイベントを生成します。このプロパティは、MySQL コネクタがテーブルの結果をメモリに格納するか、ストリーミングを行うかを決定します。メモリへの格納はすばやく処理できますが、大量のメモリを必要とします。ストリーミングを行うと、処理は遅くなりますが、非常に大きなテーブルにも対応できます。このプロパティの設定は、コネクタが結果のストリーミングを行う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。</p> <p>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場合は、このプロパティを 0 に設定します。</p>
heartbeat.interval.ms	0	<p>コネクタがハートビートメッセージを Kafka トピックに送信する頻度を制御します。デフォルトの動作では、コネクタはハートビートメッセージを送信しません。</p> <p>ハートビートメッセージは、コネクタ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のに便利です。ハートビートメッセージは、コネクタの再起動時に再送信する必要がある変更イベントの数を減らすのに役立つ可能性があります。ハートビートメッセージを送信するには、このプロパティを、ハートビートメッセージの間隔をミリ秒単位で示す正の整数に設定します。</p>
heartbeat.action.query	デフォルトなし	<p>コネクタがハートビートメッセージを送信するときにコネクタがソースデータベースで実行するクエリを指定します。</p> <p>たとえば、</p> <pre>INSERT INTO gtid_history_table (select * from mysql.gtid_executed)</pre> <p>を使用して、ソースデータベースで設定された実行済み GTID の状態を定期的にキャプチャできます。</p>

プロパティ	デフォルト	説明
database.initial.statement s	デフォルトなし	<p>トランザクションログを読み取る接続ではなく、データベースへの JDBC 接続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のリスト。SQL ステートメントでセミコロンを区切り文字としてではなく、文字として指定する場合は、2つのセミコロン (;;) を使用します。</p> <p>コネクターは独自の判断で JDBC 接続を確立する可能性があるため、このプロパティはセッションパラメーターの設定専用です。DML ステートメントを実行するものではありません。</p>
snapshot.delay.ms	デフォルトなし	<p>コネクターの起動時にスナップショットを実行するまでコネクターが待つ必要がある間隔(ミリ秒単位)。クラスターで複数のコネクターを起動する場合、このプロパティは、コネクターのリバランスが行われる原因となるスナップショットの中断を防ぐのに役立ちます。</p>
snapshot.fetch.size	デフォルトなし	<p>スナップショットの実行中、コネクターは行のバッチでテーブルの内容を読み取ります。このプロパティは、バッチの行の最大数を指定します。</p>
snapshot.lock.timeout.ms	10000	<p>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。コネクターがこの期間にテーブルロックを取得できないと、スナップショットは失敗します。Debezium 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。</p>
enable.time.adjuster	true	<p>コネクターによって2桁の西暦が4桁の西暦に変換されるかどうかを示すブール値。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。</p> <p>MySQL では、2桁または4桁の数値のいずれかで西暦の値を挿入できます。2桁の値の場合は、値は1970 - 2069の範囲の年にマッピングされます。デフォルトの動作では、コネクターは変換を行いません。</p>

プロパティ	デフォルト	説明
sanitize.field.names	コネクターが key.converter または value.converter プロパティを Avro コンバーターに設定する場合は true に設定します。それ以外は false に設定します。	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を示します。
skipped.operations	t	ストリーミング中にスキップされる oplog 操作のコンマ区切りリスト。挿入/作成は c 、更新は u 、削除は d 、切り捨ては t 、操作をスキップしない場合は none となります。デフォルトでは、切り捨て操作が省略されます。
signal.data.collection	デフォルト値なし	シグナルをコネクターに送信するために使用されるデータコレクションの完全修飾名。 <databaseName>.<tableName> の形式を使用してコレクション名を指定します。
incremental.snapshot.allow.schema.changes	false	増分スナップショット時のスキーマの変更を許可します。有効にすると、コネクターは増分スナップショットの実行中にスキーマの変更を検出し、ロック DDL を回避するために現在のチャンクを再選択します。 プライマリーキーへの変更はサポートされず、増分スナップショットの実行時に実行された場合には誤った結果が生じる可能性があります。もう1つの制限は、スキーマの変更が列のデフォルト値のみに影響する場合、DDL が binlog ストリームから処理されるまで変更が検出されないことです。これはスナップショットイベントの値には影響しませんが、スナップショットイベントのスキーマにはデフォルト値が古くなっている可能性があります。

プロパティ	デフォルト	説明
<code>incremental.snapshot.chunk.size</code>	1024	増分スナップショットのチャンクの実行中にコネクターがメモリーを取得して読み取る行の最大数。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。
<code>provide.transaction.meta data</code>	false	コネクターがトランザクション境界でイベントを生成し、トランザクションメタデータで変更イベントエンベロープを強化するかどうかを決定します。コネクターにこれを実行させる場合は true を指定します。詳細は、 トランザクションメタデータ を参照してください。
<code>topic.naming.strategy</code>	<code>io.debezium.schema.DefaultTopicNamingStrategy</code>	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する TopicNamingStrategy クラスの名前。デフォルトは DefaultTopicNamingStrategy 。
<code>topic.delimiter</code>	.	トピック名の区切り文字を指定します。デフォルトは . です。
<code>topic.cache.size</code>	10000	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。
<code>topic.heartbeat.prefix</code>	<code>__debezium-heartbeat</code>	コネクターがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は __debezium-heartbeat.fulfillment になります。
<code>topic.transaction</code>	transaction	コネクターがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は fulfillment.transaction になります。

Debezium コネクターデータベーススキーマ履歴設定プロパティ

Debezium には、コネクターがスキーマ履歴トピックと対話する方法を制御する **schema.history.internal.*** プロパティのセットが含まれています。

以下の表は、Debezium コネクターを設定するための **schema.history.internal** プロパティについて説明しています。

表5.29 コネクターデータベーススキーマ履歴の設定プロパティ

プロパティ	デフォルト	説明
schema.history.internal.kafka.a.topic	デフォルトなし	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。
schema.history.internal.kafka.a.bootstrap.servers	デフォルトなし	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。各ペアは、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。
schema.history.internal.kafka.a.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。
schema.history.internal.kafka.a.query.timeout.ms	3000	Kafka 管理クライアントを使用してクラスター情報を取得する際に、コネクターが待機すべき最大ミリ秒数を指定する整数値です。
schema.history.internal.kafka.a.create.timeout.ms	30000	Kafka 管理クライアントを使用して kafka 履歴トピックを作成する間、コネクターが待機する最大ミリ秒数を指定する整数値。
schema.history.internal.kafka.a.recovery.attempts	100	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 × recovery.poll.interval.ms です。
schema.history.internal.skip.unparseable.ddl	false	コネクターが不正または不明なデータベースのステートメントを無視するかどうか、または人が問題を修正するために処理を停止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 の処理中にデータの損失や分割を引き起こす可能性があるため、必ず注意して使用する必要があります。

プロパティ	デフォルト	説明
<code>schema.history.internal.store.only.captured.tables.ddl</code>	<code>false</code>	<p>コネクターがすべての DDL ステートメントを記録するかどうかを指定するブール値</p> <p><code>true</code> は、変更が Debezium によってキャプチャーされるテーブルに関連する DDL ステートメントのみを記録します。変更がキャプチャーされたテーブルを変更すると、不足しているデータが必要になる可能性があるため、<code>true</code> に設定するには注意が必要です。</p> <p>安全なデフォルトは <code>false</code> です。</p>

プロデューサーおよびコンシューマクライアントを設定するためのパススルーデータベーススキーマ履歴プロパティ

Debezium は、Kafka プロデューサーを使用して、データベーススキーマ履歴トピックにスキーマの変更を書き込みます。同様に、コネクターが起動すると、データベーススキーマ履歴トピックから読み取る Kafka コンシューマーに依存します。 `schema.history.internal.producer.*` および `schema.history.internal.consumer.*` 接頭辞で始まるパススルー設定プロパティのセットに値を割り当てて、Kafka プロデューサーおよびコンシューマクライアントの設定を定義します。パススループロデューサーおよびコンシューマデータベーススキーマ履歴プロパティは、以下の例のように Kafka ブローカーとのこれらのクライアントの接続をセキュアにする方法など、さまざまな動作を制御します。

```

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

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

```

Debezium は、プロパティを Kafka クライアントに渡す前に、プロパティ名から接頭辞を削除します。

[Kafka プロデューサー設定プロパティ](#) および [Kafka コンシューマー設定プロパティ](#) の詳細は、Kafka のドキュメントを参照してください。

Debezium コネクター Kafka は設定プロパティをシグナル化します。

MySQL コネクターが読み取り専用として設定されている場合、シグナルテーブルの代替は Kafka トピックを示します。

Debezium は、コネクターが Kafka シグナルトピックと対話する方法を制御する `signal.*` プロパティのセットを提供します。

以下の表は **signal** プロパティについて説明しています。

表5.30 Kafka のシグナル設定プロパティ

プロパティ	デフォルト	説明
signal.kafka.topic	デフォルトなし	コネクタがアドホックシグナルについて監視する Kafka トピックの名前。
signal.kafka.bootstrap.servers	デフォルトなし	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が使用するホストとポートのペアの一覧。各ペアは、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。
signal.kafka.poll.timeout.ms	100	信号をポーリングするときにコネクタが待機する最大ミリ秒数を指定する整数値。デフォルトは 100 ミリ秒です。

Debezium コネクタのパススルーは Kafka コンシューマクライアント設定プロパティを示唆します。

Debezium コネクタでは、Kafka コンシューマのパススルー設定が可能です。パススルーシグナルのプロパティは、接頭辞 **signals.consumer.*** で始まります。たとえば、コネクタは **signal.consumer.security.protocol=SSL** などのプロパティを Kafka コンシューマに渡します。

[データベーススキーマ履歴クライアントのパススループロパティ](#) の場合のように、Debezium はプロパティから接頭辞を削除してから Kafka シグナルコンシューマに渡します。

Debezium コネクタのパススルーデータベースドライバー設定プロパティ

Debezium コネクタでは、データベースドライバーのパススルー設定が可能です。パススルーデータベースプロパティは接頭辞 **driver.*** で始まります。たとえば、コネクタは **driver.foofoo=false** などのプロパティを JDBC URL に渡します。

[データベーススキーマ履歴クライアントのパススループロパティ](#) の場合のように、Debezium はプロパティから接頭辞を削除してからデータベースドライバーに渡します。

5.6. DEBEZIUM MYSQL コネクタのパフォーマンスの監視

Debezium MySQL コネクタは、Zookeeper、Kafka、および Kafka Connect によって提供される JMX メトリクスの組み込みサポートに加えて、3種類のメトリクスを提供します。

- **スナップショットメトリクス** は、スナップショットの実行中にコネクタ操作に関する情報を提供します。
- **ストリーミングメトリクス** は、コネクタが binlog を読み取る際のコネクタ操作に関する情報を提供します。
- **スキーマ履歴メトリクス** は、コネクタのスキーマ履歴の状態に関する情報を提供します。

[Debezium モニタリングのドキュメント](#) では、JMX を使用してこれらのメトリクスを公開する方法の詳細を説明しています。

5.6.1. MySQL データベースのスナップショット作成時の Debezium の監視

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

スナップショット操作がアクティブでない場合や、最後のコネクターの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	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 include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	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。
QueueRemainingCapacity	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。

属性	タイプ	説明
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

コネクタは、増分スナップショットの実行時に、以下の追加のスナップショットメトリクスも提供します。

属性	タイプ	説明
ChunkId	string	現在のスナップショットチャンクの識別子。
ChunkFrom	string	現在のチャンクを定義するプライマリーキーセットの下限。
ChunkTo	string	現在のチャンクを定義するプライマリーキーセットの上限。
TableFrom	string	現在スナップショットされているテーブルのプライマリーキーセットの下限。
TableTo	string	現在スナップショットされているテーブルのプライマリーキーセットの上限。

Debezium MySQL コネクターは、**HoldingGlobalLock** カスタムスナップショットメトリクスも提供します。このメトリクスは、コネクターが現在グローバルまたはテーブル書き込みロックを保持するかどうかを示すブール値に設定されます。

5.6.2. Debezium MySQL コネクターレコードストリーミングの監視

トランザクション関連の属性は、binlog イベントのバッファが有効になっている場合にのみ利用できます。MBean は **debezium.mysql:type=connector-metrics,context=streaming,server=<mysql.server.name>** です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。

属性	タイプ	説明
TotalNumberOfCreateEventsSeen	long	このコネクタが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクタが見た更新イベントの合計数。
TotalNumberOfDeleteEventsSeen	long	このコネクタが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクタに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクタによって取得されるテーブルの一覧。
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。
Connected	boolean	コネクタ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との間の期間(ミリ秒単位)。この値は、データベースサーバーとコネクタが稼働しているマシンのクロック間の差異に対応します。
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	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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

Debezium MySQL コネクターは、以下のストリーミングメトリクスも追加で提供します。

表5.31 追加のストリーミングメトリクスの説明

属性	タイプ	説明
BinlogFilename	string	コネクターによって最後に読み取られた binlog ファイルの名前。
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処理される最新の GTID セットの文字列表現。
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。
NumberOfDisconnects	long	MySQL コネクターによる切断の数。
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	想定された BEGIN + COMMIT/ROLLBACK のプロトコルに準拠していないトランザクションの数。この値は、通常の条件下では 0 である必要があります。

属性	タイプ	説明
NumberOfLargeTransactions	long	先読みバッファーに適合し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、この値は NumberOfCommittedTransactions と NumberOfRolledBackTransactions よりも大幅に小さくする必要があります。

5.6.3. Debezium MySQL コネクタのスキーマ履歴の監視

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

以下の表は、利用可能なスキーマ履歴メトリクスの一覧です。

属性	タイプ	説明
Status	string	データベーススキーマ履歴の状態を示す STOPPED 、 RECOVERING (ストレージから履歴を復元)、または 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。

属性	タイプ	説明
LastAppliedChange	string	最後に適用された変更の文字列表現。

5.7. DEBEZIUM MYSQL コネクターによる障害および問題の処理方法

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 は変更イベントレコードごとに **1度だけ** 配信します。

障害が発生しても、システムはイベントを失いません。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があります。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。

詳細は以下を参照してください。

- [設定および起動エラー](#)
- [MySQL が使用不可能になる](#)
- [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](#)
- [MySQL が binlog ファイルをパージする](#)

設定および起動エラー

以下の状況では、起動時にコネクターが失敗し、エラーまたは例外がログに記録され、実行が停止されます。

- 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ない。
- MySQL に履歴がない binlog の位置でコネクターが再起動を試行する。

このような場合、エラーメッセージには問題の詳細が含まれ、推奨される回避策も含まれることがあります。設定の修正したり、MySQL の問題に対処した後、コネクターを再起動します。

MySQL が使用不可能になる

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。サーバーが再び使用できるようになったら、コネクターを再起動します。

ただし、高可用性 MySQL クラスタで GTID が有効になっている場合は、コネクターをすぐに再起動できます。これはクラスタの別の 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 の場所を特定し、その特定の場所から新しいサーバーの binlog の読み取りを開始します。

GTID が有効になっていない場合、コネクターは接続した MySQL サーバーのみの binlog の位置を記録します。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。

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に短い遅延が発生します。

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 コネクタータスク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、代替のタスクによってクラッシュ前に処理された同じイベントの一部が生成され、重複したイベントが作成される可能性があります。

各変更イベントメッセージには、重複イベントの特定に使用できるソース固有の情報が含まれます。以下に例を示します。

- イベント元
- MySQL サーバーのイベント時間
- binlog ファイル名と位置
- GTID (使用されている場合)

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 コネクタータスクは接続が再確立されるまで一時停止され、一時停止した位置で再開されます。

MySQL が binlog ファイルをパージする

Debezium MySQL コネクタータスク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、コネクタータスクの最後の位置が失われる可能性があります。コネクタータスクが再起動すると、MySQL サーバーに開始点がなくなり、コネクタータスク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、コネクタータスクはエラーによって失敗します。

MySQL コネクタータスクが最初のスナップショットを実行する方法は、[Debezium MySQL コネクタータスク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6章 ORACLE の DEBEZIUM コネクタ

Debezium の Oracle コネクタは、Oracle サーバーのデータベースで発生する行レベルの変更をキャプチャーして記録します。これには、コネクタの実行中に追加されたテーブルが含まれます。コネクタを設定して、スキーマおよびテーブルの特定のサブセットの変更イベントを出力したり、特定の列で値を無視、マスク、または切り捨てたりするように設定できます。

このコネクタと互換性のある Oracle データベースのバージョンについては、[Debezium](#) でサポートされる設定ページを参照してください。

ネイティブの LogMiner データベースパッケージを使用して、Debezium が Oracle から最も新しい変更イベントを取り込みます。

Debezium Oracle コネクタの使用に関する情報および手順は、以下のように整理されています。

- [「Debezium Oracle コネクタの仕組み」](#)
- [「Debezium Oracle コネクタのデータ変更イベントの説明」](#)
- [「Debezium Oracle コネクタによるデータ型のマッピング方法」](#)
- [「Debezium と連携させるための Oracle の設定」](#)
- [「Debezium Oracle コネクタのデプロイメント」](#)
- [「Debezium Oracle コネクタ設定プロパティの説明」](#)
- [「Debezium Oracle コネクタのパフォーマンスの監視」](#)
- [「Oracle コネクタのよくある質問」](#)

6.1. DEBEZIUM ORACLE コネクタの仕組み

Debezium Oracle コネクタを最適に設定し実行するには、コネクタがどのようにスナップショットを実行し、変更イベントをストリームして、Kafka トピック名を決定し、メタデータを使用して、イベントバッファリングを実装するのかを理解することが役に立ちます。

詳細は、以下のトピックを参照してください。

- [「Debezium Oracle コネクタによるデータベーススナップショットの実行方法」](#)
- [「Debezium Oracle 変更イベントレコードを受信する Kafka トピックのデフォルト名」](#)
- [「Debezium Oracle コネクタによるデータベーススキーマの変更の公開方法」](#)
- [「トランザクション境界を表す Debezium Oracle コネクタによって生成されたイベント」](#)
- [「Debezium Oracle コネクタのイベントバッファリング使用方法」](#)

6.1.1. Debezium Oracle コネクタによるデータベーススナップショットの実行方法

通常、Oracle サーバーの redo ログは、WAL セグメントにデータベースの全履歴を保持するようには設定されていません。そのため、Debezium Oracle コネクタはログからデータベースの履歴全体を取得できません。コネクタがデータベースの現在の状態のベースラインを確立できるようにするには、コネクタの初回起動時に、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。

snapshot.mode コネクタ設定プロパティの値を設定することで、コネクタがスナップショットを作成する方法をカスタマイズできます。デフォルトでは、コネクタのスナップショットモードは **initial** に設定されます。

初期スナップショットを作成するデフォルトのコネクタワークフロー

スナップショットモードがデフォルトに設定されている場合には、コネクタは以下の作業を完了してスナップショットを作成します。

1. キャプチャするテーブルを決定します。
2. スナップショットの作成中に構造的な変更が発生しないように、キャプチャした各テーブルの **ROW SHARE MODE** ロックを取得します。Debezium は短期間のみ、ロックを保持します。
3. サーバーの redo ログから現在のシステム変更番号 (SCN) の位置を読み取ります。
4. 関連するテーブルすべての構造をキャプチャします。
5. ステップ 2 で取得したロックを解放します。
6. 手順 3 で読み込まれた SCN 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て (**SELECT * FROM ... AS OF SCN 123**)、各行に **READ** イベントを生成し、イベントレコードをテーブル固有の Kafka トピックに書き込みます。
7. コネクタ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。

スナップショットプロセスが開始されたら、コネクタの障害、リバランス、またはその他の理由でプロセスが中断されると、コネクタの再起動後にプロセスが再起動されます。コネクタによって最初のスナップショットが完了した後、更新に抜けがないように、ステップ 3 で読み取りした位置からストリーミングを続行します。何らかの理由でコネクタが再び停止した場合に、コネクタは再起動後に最後に停止した位置から変更のストリーミングを再開します。

表6.1 snapshot.mode コネクタ設定プロパティの設定

設定	説明
always	各コネクタの開始時にスナップショットを実行します。スナップショットが完了すると、コネクタは、後続のデータベース変更のに備え、イベントレコードのストリーミングを開始します。
Initial	コネクタは、 最初のスナップショットを作成するためのデフォルトのワークフロー で説明されているように、データベーススナップショットを実行します。スナップショットが完了すると、コネクタは、後続のデータベース変更のに備え、イベントレコードのストリーミングを開始します。
initial_only	コネクタはデータベースのスナップショットを実行し、変更イベントレコードをストリーミングする前に停止して、それ以降の変更イベントのキャプチャを許可しません。
schema_only	コネクタは関連するすべてのテーブルの構造をキャプチャし、 デフォルトのスナップショットワークフロー に記載されているすべてのステップを実行します。ただし、コネクタの起動時 (Step 6) の時点でデータセットを表す READ イベントが作成されない点が異なります。

設定	説明
schema_only_recovery	<p>損失または破損したデータベーススキーマの履歴トピックを復元するにはこのオプションを設定します。再起動後、コネクタはソーステーブルからトピックを再構築するスナップショットを実行します。また、このプロパティを設定して、予期しない増加が発生するデータベーススキーマ履歴トピックを定期的にプルーニングすることもできます。</p> <p>警告: 最後のコネクタのシャットダウン後にスキーマの変更がデータベースにコミットされた場合、このモードを使用してスナップショットを実行しないでください。</p>

詳細については、コネクタ設定プロパティテーブルの **snapshot.mode** をご覧ください。

6.1.1.1. アドホックスナップショット

デフォルトでは、コネクタは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクタではスナップショットプロセスは繰り返し処理されません。コネクタがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクタを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。テーブルデータを再キャプチャーするメカニズムを提供するため、Debezium にはアドホックスナップショットを実行するオプションがあります。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクタ設定は、異なるテーブルセットをキャプチャーするように変更されます。
- Kafka トピックを削除して、再構築する必要があります。
- 設定エラーや他の問題が原因で、データの破損が発生します。

アドホックと呼ばれるスナップショットを開始することで、以前にスナップショットをキャプチャーしたテーブルのスナップショットを再実行できます。アドホックスナップショットには、**シグナルテーブル**を使用する必要があります。シグナルリクエストを Debezium シグナルテーブルに送信して、アドホックスナップショットを開始します。

既存のテーブルのアドホックスナップショットを開始すると、コネクタはテーブルにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、**トピックの自動作成**が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックのスナップショットシグナルは、スナップショットに追加するテーブルを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のテーブルのサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のテーブルの内容のサブセットをキャプチャできます。

execute-snapshot メッセージをシグナルテーブルに送信してキャプチャーするテーブルを指定します。以下の表で説明されているように、**run-snapshot** シグナルのタイプを **incremental** に設定し、スナップショットに追加するテーブルの名前を指定します。

表6.2 アドホックの **execute-snapshot** シグナルレコードの例

フィールド	デフォルト	値
type	incremental	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 incremental スナップショットのみです。
data-collections	該当なし	スナップショットされるテーブルの完全修飾名にマッチする正規表現を含む配列。名前の形式は signal.data.collection 設定オプションと同じです。
additional-condition	該当なし	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

execute-snapshot シグナルタイプのエントリーをシグナルテーブルに追加して、アドホックスナップショットを開始します。コネクタがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各テーブルの開始ポイントおよびエンドポイントとして使用します。テーブルのエントリー数と設定されたチャンクサイズに基づいて、Debezium はテーブルをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、**execute-snapshot** アクションタイプは **増分スナップショット** のみをトリガーします。詳細は、[スナップショットの増分](#) を参照してください。

6.1.1.2. 増分スナップショット

スナップショットを柔軟に管理するため、Debezium には **増分スナップショット** と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、[Debezium コネクタにシグナルを送信するための Debezium メカニズム](#) に依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各テーブルを段階的にキャプチャーします。スナップショットがキャプチャーするテーブルと、[各チャンクのサイズ](#) を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

増分スナップショットが進むと、Debezium はウォーターマークを使用して進捗を追跡し、キャプチャーする各テーブル行のレコードを管理します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクタはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセスが再開すると、スナップショットは最初からテーブルをキャプチャーするのではなく、停止した時点から開始します。
- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。たとえば、コネクタ設定を変更してテーブルを **table.include.list** プロパティに追加した後にスナップショットを再実行します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各テーブルをプライマリーキー別に分類して、**設定されたチャンクサイズ**に基づいてテーブルをチャンクに分割します。チャンクごとに作業し、テーブルの行ごとにチャンクでキャプチャーします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットの作成が進むにつれ、他のプロセスがデータベースへのアクセスを継続し、テーブルレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT**、**UPDATE**、**DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できません。つまり、ストリーミングプロセスは、スナップショットがその行の **READ** イベントが含まれるチャンクをキャプチャーする前に、テーブルの行を変更するイベントを生成する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリームされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて入ってきた **READ** イベントと、同じテーブルの行を変更するストリーミングイベント間の競合の解決を容易にするために、Debezium は **スナップショットウィンドウ** と呼ばれるものを使用します。スナップショットウィンドウは、増分スナップショットが指定のテーブルチャンクのデータをキャプチャーしている途中に、間隔を決定します。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は 2 種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャーするスナップショットレコードは、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをテーブルの Kafka トピックに出力します。

コネクターは各スナップショットチャンクにプロセスを繰り返します。



警告

Oracle の Debezium コネクターでは、増分スナップショットの実行中のスキーマの変更はサポートしません。

6.1.1.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、[アドホックスナップショットシグナル](#) をソースデータベースのシグナルテーブルに送信することです。

シグナルを SQL **INSERT** クエリーとしてシグナルテーブルに送信します。

Debezium がシグナルテーブルの変更を検出すると、シグナルを読み取り、要求されたスナップショット操作を実行します。

送信するクエリーはスナップショットに追加するテーブルを指定し、必要に応じてスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに追加するテーブルを指定するには、テーブルをリストする **data-collections** 配列またはテーブルの照合に使用する正規表現の配列を指定します。以下に例を示します。

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

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるテーブルの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのテーブルを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **My.Table** のテーブルを含めるには、**"public"."My.Table"** の形式を使用します。

前提条件

- [シグナルが有効になっている](#)。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは [signal.data.collection](#) プロパティで指定されます。

手順

1. SQL クエリーを送信し、アドホック増分スナップショット要求をシグナルテーブルに追加します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) ①
values ('ad-hoc-1', ②
       'execute-snapshot', ③
       '{"data-collections": ["schema1.table1", "schema2.table2"], ④
       "type":"incremental", ⑤
       "additional-condition":"color=blue"}'); ⑥
```

コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表6.3 シグナルテーブルに増分スナップショットシグナルを送信する SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリへのログメッセージを特定します。Debezium はこの文字列を使用しません。代わりに、スナップショット作成中に、Debezium は独自の ID 文字列をウォーターマークシグナルとして生成します。
3	execute-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるテーブル名の配列またはテーブル名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。
5	incremental	実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 値を指定しない場合には、コネクターは増分スナップショットを実行します。

項目	値	説明
6	additional-condition	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。 additional-condition パラメーターの詳細は、 additional-condition 付きのアドホック増分スナップショット を参照してください。

additional-condition 付きのアドホック増分スナップショット

スナップショットに、テーブル内のコンテンツのサブセットのみを含める場合は、スナップショットシグナルシグナルに **additional-condition** パラメーターを追加してシグナル要求を変更できます。

一般的なスナップショットの SQL クエリーは、以下の形式を取ります。

```
SELECT * FROM <tableName> ....
```

additional-condition パラメーターを追加して、以下の例のように **WHERE** 条件を SQL クエリーに追加します。

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```

以下の例は、シグナルテーブルに追加の条件を含むアドホック増分スナップショット要求を送信する SQL クエリーを示しています。

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

たとえば、以下の列が含まれる **products** テーブルがあるとします。

- **id** (プライマリーキー)
- **color**
- **quantity**

products テーブルの増分スナップショットに **color=blue** のデータ項目のみを含める場合は、次の SQL ステートメントを使用してスナップショットをトリガーできます。

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

additional-condition パラメーターを使用すると、列が2つ以上となる条件を指定することもできます。たとえば、前述の例の **products** テーブルを使用して、**color=blue** および **quantity>10** だけに一致するアイテムのみのデータが含まれる増分スナップショットをトリガーするクエリーを送信できます。

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-snapshot', '{"data-collections": ["schema1.products"],"type":"incremental", "additional-condition":"color=blue AND quantity>10"}');
```


以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 シグナルテーブルに送信する SQL クエリーでの type 値の指定は任意です。 値を指定しない場合には、コネクターは増分スナップショットを実行します。
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

6.1.1.4. 増分スナップショットの停止

ソースデータベースのテーブルにシグナルを送信して、増分スナップショットを停止することもできます。SQL **INSERT** クエリーを送信して、停止スナップショットシグナルをテーブルに送信します。

Debezium はシグナルテーブルの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのテーブルを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. SQL クエリーを送信して、シグナリングテーブルへのアドホックインクリメンタル スナップショットを停止します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
       'stop-snapshot', 3
       '{"data-collections": ["schema1.table1", "schema2.table2"], 4
       "type":"incremental"}'); 5
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表6.4 シグナリングテーブルに増分スナップショット停止信号を送信するための SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリーへのログメッセージを特定します。Debezium はこの文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドのオプションコンポーネントで、スナップショットから削除するテーブル名の配列またはテーブル名とマッチする正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクタのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

6.1.2. Debezium Oracle 変更イベントレコードを受信する Kafka トピックのデフォルト名

デフォルトでは、Oracle コネクタは、テーブルで発生するすべての **INSERT**、**UPDATE**、**DELETE** 操作の変更イベントを、そのテーブルに固有の単一の Apache Kafka トピックに書き込みます。コネクタは以下の規則を使用して変更イベントトピックに名前を付けます。

`topicPrefix.schemaName.tableName`

以下のリストは、デフォルト名のコンポーネントの定義を示しています。

topicPrefix

topic.prefix コネクタ設定プロパティで指定されたトピック接頭辞。

schemaName

操作が発生したスキーマの名前。

tableName

操作が発生したテーブルの名前。

たとえば、**fulfillment** がサーバー名、**inventory** がスキーマ名で、データベースに **orders**、**customers**、**products** という名前のテーブルが含まれる場合には、Debezium Oracle コネクタは、データベースのテーブルごとに1つ、以下の Kafka トピックにイベントを出力します。

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

コネクタは同様の命名規則を適用して、内部データベーススキーマの履歴トピック (**スキーマ変更トピック** と **トランザクションメタデータトピック**) にラベルを付けます。

デフォルトのトピック名が要件を満たさない場合は、カスタムトピック名を設定できます。カスタムトピック名を設定するには、論理トピックルーティング SMT に正規表現を指定します。論理トピックルーティング SMT を使用してトピックの命名をカスタマイズする方法は、**トピックルーティング** を参照してください。

6.1.3. Debezium Oracle コネクタによるデータベーススキーマの変更の公開方法

Debezium Oracle コネクタは、データベース内のキャプチャされたテーブルに適用される構造的な変更を記述するスキーマ変更イベントを生成するように設定することができます。コネクタは、スキーマ変更イベントを `<serverName>` という名前の Kafka トピックに書き込みます。ここで、**topicName** は **topic.prefix** 設定プロパティに指定した名前空間を指します。

Debezium は、新しいテーブルからデータをストリーミングするたびに、このトピックに新しいメッセージを出力します。

コネクタがスキーマ変更トピックに送信するメッセージには、ペイロードと、任意で変更イベントメッセージのスキーマが含まれます。スキーマ変更イベントメッセージのペイロードには、以下の要素が含まれます。

ddl

スキーマの変更につながる SQL **CREATE**、**ALTER**、または **DROP** ステートメントを提供します。

databaseName

ステートメントが適用されるデータベースの名前。**databaseName** の値は、メッセージキーとして機能します。

tableChanges

スキーマの変更後のテーブルスキーマ全体の構造化表現。**tableChanges** フィールドには、テーブルの各列のエントリなどのアレイが含まれます。構造化された表現は JSON または Avro 形式でデータを表示するため、コンシューマーは DDL パーサーを介して最初にメッセージを処理しなくてもメッセージを簡単に読み取りできます。

重要

デフォルトでは、コネクタは **ALL_TABLES** データベースビューを使用して、スキーマ履歴トピックに格納するテーブル名を識別します。そのビュー内で、コネクタは、データベースへの接続に使用するユーザーアカウントが使用できるテーブルからのみデータにアクセスできます。

スキーマ履歴トピックが異なるテーブルのサブセットを格納するように設定を変更できます。以下の方法のいずれかを使用して、トピックが保存するテーブルのセットを変更します。

- Debezium がデータベースへのアクセスに使用するアカウントのパーミッションを変更し、別のテーブルセットが **ALL_TABLES** ビューに表示されるようにします。
- コネクタプロパティ **`schema.history.internal.store.only.captured.tables.ddl`** を **true** に設定します。

重要

コネクタがテーブルをキャプチャするように設定されている場合、テーブルのスキーマ変更の履歴は、スキーマ変更トピックだけでなく、内部データベーススキーマの履歴トピックにも格納されます。内部データベーススキーマ履歴トピックはコネクタのみの使用を対象としており、使用するアプリケーションによる直接使用を目的としていません。スキーマ変更に関する通知が必要なアプリケーションが、スキーマ変更トピックからの情報のみを使用するようにしてください。

重要

データベーススキーマ履歴トピックをパーティションに分割しないでください。データベーススキーマ履歴トピックが正しく機能するには、コネクタが出力するイベントレコードの一貫したグローバル順序を維持する必要があります。

トピックがパーティション間で分割されないようにするには、以下のいずれかの方法を使用してトピックのパーティション数を設定します。

- データベーススキーマ履歴トピックを手動で作成する場合は、パーティション数を **1** に指定します。
- Apache Kafka ブローカーを使用してデータベーススキーマ履歴トピックを自動的に作成する場合に、トピックが作成されるので、**`Kafka num.partitions`** 設定オプションの値を **1** に設定します。

例: Oracle コネクタスキーマ変更トピックに発行されたメッセージ

以下の例は、JSON 形式の一般的なスキーマ変更メッセージを示しています。メッセージには、テーブルスキーマの論理表現が含まれます。

```

{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "2.1.4.Final",
      "connector": "oracle",
      "name": "server1",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "ORCLPDB1",
      "schema": "DEBEZIUM",
      "table": "CUSTOMERS",
      "txId": null,
      "scn": "1513734",
      "commit_scn": "1513754",
      "lcr_position": null,
      "rs_id": "001234.00012345.0124",
      "ssn": 1,
      "redo_thread": 1,
      "user_name": "user"
    },
    "ts_ms": 1588252618953, ①
    "databaseName": "ORCLPDB1", ②
    "schemaName": "DEBEZIUM", //
    "ddl": "CREATE TABLE \"DEBEZIUM\".\"CUSTOMERS\" \n ( \"ID\" NUMBER(9,0) NOT NULL
ENABLE, \n \"FIRST_NAME\" VARCHAR2(255), \n \"LAST_NAME\" VARCHAR2(255), \n
\"EMAIL\" VARCHAR2(255), \n PRIMARY KEY (\"ID\") ENABLE, \n SUPPLEMENTAL LOG
DATA (ALL) COLUMNS \n ) SEGMENT CREATION IMMEDIATE \n PCTFREE 10 PCTUSED 40
INITRANS 1 MAXTRANS 255 \n NOCOMPRESS LOGGING \n STORAGE (INITIAL 65536 NEXT
1048576 MINEXTENTS 1 MAXEXTENTS 2147483645 \n PCTINCREASE 0 FREELISTS 1
FREELIST GROUPS 1 \n BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT
CELL_FLASH_CACHE DEFAULT) \n TABLESPACE \"USERS\" ", ③
    "tableChanges": [ ④
      {
        "type": "CREATE", ⑤
        "id": "\"ORCLPDB1\".\"DEBEZIUM\".\"CUSTOMERS\"", ⑥
        "table": { ⑦
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ ⑧
            "ID"
          ],
        },
        "columns": [ ⑨
          {
            "name": "ID",
            "jdbcType": 2,
            "nativeType": null,
            "typeName": "NUMBER",
            "typeExpression": "NUMBER",
            "charsetName": null,
            "length": 9,
            "scale": 0,
            "position": 1,
            "optional": false,

```

```
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "FIRST_NAME",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR2",
    "typeExpression": "VARCHAR2",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 2,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "LAST_NAME",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR2",
    "typeExpression": "VARCHAR2",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 3,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "EMAIL",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "VARCHAR2",
    "typeExpression": "VARCHAR2",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 4,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  }
],
"attributes": [ 10
  {
    "customAttribute": "attributeValue"
  }
]
}
}
```

表6.5 スキーマ変更トピックに出力されたメッセージのフィールドの説明

項目	フィールド名	説明
1	ts_ms	<p>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。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。</p> <p>ソースオブジェクトの ts_ms は、データベースで変更が行われた時刻を示す。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>
2	databaseName schemaName	変更が含まれるデータベースとスキーマを識別します。
3	ddl	このフィールドには、スキーマの変更を行う DDL が含まれます。
4	tableChanges	DDL コマンドによって生成されるスキーマの変更が含まれる1つ以上の項目の配列。
5	type	<p>変更の種類を説明します。type は以下の値のいずれかに設定できます。</p> <p>CREATE テーブルの作成</p> <p>ALTER テーブルの変更</p> <p>DROP テーブルの削除</p>
6	id	作成、変更、または破棄されたテーブルの完全な識別子。テーブルの名前が変更されると、この識別子は<old>,<new> のテーブル名が連結されます。
7	table	適用された変更後のテーブルメタデータを表します。
8	primaryKeyColumnNames	テーブルのプライマリーキーを設定する列のリスト。
9	列	変更されたテーブルの各列のメタデータ。
10	attributes	各テーブル変更のカスタム属性メタデータ。

コネクタがスキーマ変更トピックに送信するメッセージでは、メッセージキーはスキーマの変更が含まれるデータベースの名前です。以下の例では、**payload** フィールドに **databaseName** キーが含まれます。

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "string",
        "optional": false,
        "field": "databaseName"
      }
    ],
    "optional": false,
    "name": "io.debezium.connector.oracle.SchemaChangeKey"
  },
  "payload": {
    "databaseName": "ORCLPDB1"
  }
}
```

6.1.4. トランザクション境界を表す Debezium Oracle コネクタによって生成されたイベント

Debezium は、トランザクションメタデータ境界を表し、データ変更イベントメッセージをエンリッチするイベントを生成できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクタのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクタをデプロイする前に発生するトランザクションのメタデータは利用できません。

データベーストランザクションは、キーワード **BEGIN** および **END** で囲まれたステートメントブロックによって表されます。Debezium は、すべてのトランザクションで **BEGIN** および **END** 区切り文字のトランザクション境界イベントを生成します。トランザクション境界イベントには以下のフィールドが含まれます。

status

BEGIN または **END**

id

一意のトランザクション識別子の文字列表現。

ts_ms

データソースでのトランザクション境界イベント (**BEGIN** または **END** イベント) の時間。もしデータソースが Debezium にイベント時間を提供しないなら、このフィールドは代わりに Debezium がイベントを処理する時間を表します。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

data_collection と **event_count** 要素のペアの配列。これは、コネクターがデータコレクションから発信された変更に対して出力するイベントの数を示します。

以下の例は、典型的なトランザクション境界メッセージを示しています。

例: Oracle コネクタートランザクション境界イベント

```
{
  "status": "BEGIN",
  "id": "5.6.641",
  "ts_ms": 1486500577125,
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "5.6.641",
  "ts_ms": 1486500577691,
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.CUSTOMER",
      "event_count": 1
    },
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.ORDER",
      "event_count": 1
    }
  ]
}
```

topic.transaction オプションで上書きされない限り、コネクターはトランザクションイベントを **<topic.prefix>.transaction** トピックに出力します。

6.1.4.1. Debezium Oracle コネクターがトランザクションメタデータで変更イベントメッセージを強化する方法

トランザクションメタデータを有効にすると、データメッセージ **Envelope** は新しい **transaction** フィールドでエンリッチされます。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下の例は、典型的なトランザクションのイベントメッセージを示しています。

```
{
  "before": null,
```

```
"after": {
  "pk": "2",
  "aa": "1"
},
"source": {
...
},
"op": "c",
"ts_ms": "1580390884335",
"transaction": {
  "id": "5.6.641",
  "total_order": "1",
  "data_collection_order": "1"
}
}
```

6.1.5. Debezium Oracle コネクターのイベントバッファリング使用方法

Oracle は、後でロールバックによって破棄された変更を含め、発生した順序で再実行ログにすべての変更を書き込みます。その結果、別のトランザクションからの同時変更はインターアップトアンドされます。コネクターが最初に変更ストリームを読み取ると、どの変更がコミットまたはロールバックされるかをすぐに判断できないため、変更イベントは内部バッファに一時的に保存されます。変更がコミットされると、コネクターは変更イベントをバッファから Kafka に書き込みます。コネクターはロールバックによって破棄される変更イベントを破棄します。

プロパティ `log.mining.buffer.type` を設定することにより、コネクターが使用するバッファリングメカニズムを設定できます。

ヒープ

デフォルトのバッファータイプは `memory` を使用して設定されます。デフォルトの `memory` 設定では、コネクターは JVM プロセスのヒープメモリーを使用してバッファイベントレコードを割り当て、管理します。`memory` バッファ設定を使用する場合は、Java プロセスに割り当てるメモリー量が、お使いの環境で長時間実行されるトランザクションや大規模トランザクションに対応することができることを確認してください。

6.1.6. Debezium Oracle コネクターが SCN 値のギャップを検出する方法

Debezium Oracle コネクターが LogMiner を使用するよう設定されると、システム変更番号 (SCN) に基づく開始範囲と終了範囲を使用して、Oracle から変更イベントを収集します。コネクターはこの範囲を自動的に管理し、コネクターがほぼリアルタイムで変更を流せるか、データベース内の大規模またはバルクトランザクションの量によって変更のバックログを処理しなければならないかに応じて、範囲を増減させます。

特定の状況下で、Oracle データベースは SCN 値を一定の割合で増加させるのではなく、異常に高い割合で SCN を前進させます。このような SCN 値のジャンプは、特定のインテグレーションがデータベースと対話する方法やホットバックアップなどのイベントの結果により発生する可能性があります。

Debezium Oracle コネクターは、以下の設定プロパティに依存して SCN ギャップを検出し、マイニング範囲を調整します。

`log.mining.scn.gap.detection.gap.size.min`

ギャップの最小サイズを指定します。

`log.mining.scn.gap.detection.time.interval.max.ms`

最大間隔を指定します。

コネクタは最初に、現在のマイニング範囲で現在の SCN と最大 SCN との間の変更数の違いを比較します。現在の SCN 値と最高 SCN 値の差が最小ギャップサイズより大きい場合、コネクタは SCN ギャップを潜在的に検出していることになる。ギャップが存在するかどうかを確認するために、コネクタは次に前のマイニング範囲の最後に現在の SCN および SCN のタイムスタンプを比較します。タイムスタンプの違いが最大間隔未満の場合、SCN ギャップの存在が確認されます。

SCN ギャップが発生すると、Debezium コネクタは現在のマイニングセッションの範囲のエンドポイントとして現在の SCN を自動的に使用します。これにより、SCN 値が予期せず増加するため、コネクタは変更を返す間で小規模な範囲を減らさずにリアルタイムイベントを迅速にキャッチできます。コネクタが SCN ギャップに応答して前述の手順を実行すると、`log.mining.batch.size.max` プロパティで指定された値を無視します。コネクタがマイニングセッションを終了し、リアルタイムのイベントに追いついた後、最大ログマイニングバッチサイズの実施を再開します。



警告

SCN ギャップ検出は、コネクタが動作していて、ほぼリアルタイムでイベントを処理しているときに、大きな SCN 増分が発生した場合にのみ有効です。

6.1.7. Debezium は、変更頻度の低いデータベースでオフセットをどのように管理するか？

Debezium Oracle コネクタは、コネクタオフセットでシステム変更番号を追跡するので、コネクタを再起動したときに、中断したところから開始することができます。これらのオフセットは、発行された各変更イベントの一部です。ただし、データベース変更の頻度が低い場合 (数時間または数日ごと)、オフセットが古くなり、トランザクションログでシステム変更番号が利用できなくなると、コネクタの再起動が正常に行われなくなる可能性があります。

Oracle への接続に非 CDB モードを使用するコネクタの場合、オフセットの同期を維持するために、コネクタが一定間隔でハートビートイベントを発信するように強制するために `heartbeat.interval.ms` を有効にすると、コネクタが定期的にハートビートイベントを発行するようになり、オフセットの同期が維持されます。

Oracle との接続に CDB モードを使用するコネクタの場合、Synchronization のメンテナンスはより複雑になります。`heartbeat.interval.ms` を設定する必要があるだけでなく `heartbeat.interval.ms` を設定する必要があります。CDB モードでは、コネクタは PDB 内の変更のみを追跡するため、両方のプロパティを指定する必要があります。プラグブルデータベース内から変更イベントをトリガーするための補足的なメカニズムが必要である。一定の間隔で、ハートビートアクションクエリーがコネクタに新しいテーブル行を挿入したり、プラグブルデータベースの既存の行を更新したりします。Debezium は、テーブルの変更を検知し、変更イベントを発行することで、変更処理の頻度が低いプラグブルデータベースでもオフセットの同期を確保します。



注記

コネクタのユーザーアカウントが所有していないテーブルで `heartbeat.action.query` を使用するには、コネクタのユーザーにそれらのテーブルに必要な `INSERT` または `UPDATE` クエリーを実行する権限を付与する必要があります。

6.2. DEBEZIUM ORACLE コネクタのデータ変更イベントの説明

Oracle コネクタが出力する全データ変更イベントにはキーと値があります。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。Debezium のトピック名を構築する方法は、[トピック名](#)を参照してください。



警告

Debezium Db2 コネクタは、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。つまり、論理サーバー名はアルファベット文字またはアンダースコア ([a-z,A-Z,_]) で始まり、論理サーバー名の残りの文字と、スキーマおよびテーブル名の残りの文字は英数字またはアンダースコア ([a-z,A-Z,0-9,_]) で始まる必要があります。コネクタは無効な文字をアンダースコアに自動的に置き換えます。

複数の論理サーバー名、スキーマ名、またはテーブル名の中で区別ができる文字のみが無効な場合に、アンダースコアに置き換えられると、命名で予期しない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](#) を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があります。トピックコンシューマーによる処理が困難になることがあります。変更可能なイベント構造の処理を容易にするため、Kafka Connect の各イベントは自己完結型となっています。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。



警告

SYS、**SYSTEM** ユーザーアカウントが加える変更は、コネクタではキャプチャーされません。

データ変更イベントの詳細は、以下を参照してください。

- [「Debezium Oracle コネクタ変更イベントのキー」](#)
- [「Debezium Oracle 変更イベントの値」](#)

6.2.1. Debezium Oracle コネクタ変更イベントのキー

変更されたテーブルごとに変更イベントキーは、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）の各コラムにフィールドが存在するように設定されます。

たとえば、**inventory** データベーススキーマに定義されている **customers** テーブルには、以下の変更イベントキーが含まれる場合があります。

```
CREATE TABLE customers (
  id NUMBER(9) GENERATED BY DEFAULT ON NULL AS IDENTITY (START WITH 1001) NOT
```

```

NULL PRIMARY KEY,
first_name VARCHAR2(255) NOT NULL,
last_name VARCHAR2(255) NOT NULL,
email VARCHAR2(255) NOT NULL UNIQUE
);

```

<topic.prefix>.transaction 設定プロパティの値が **server1** に設定されている場合は、データベースの **customers** テーブルで発生するすべての変更イベントの JSON 表現には以下のキー構造が使用されます。

```

{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "ID"
      }
    ],
    "optional": false,
    "name": "server1.INVENTORY.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。上記の例では、**payload** 値はオプションではなく、構造は **server1.DEBEZIUM.CUSTOMERS.Key** という名前のスキーマで定義され、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** フィールドの値から、**id** フィールドが1つでその値が **1004** の構造 (JSON では単なるオブジェクト) であることが分かります。

つまり、このキーは、**id** プライマリーキーの列にある値が **1004** の、**inventory.customers** テーブルの行 (名前が **server1** のコネクターからの出力) を記述していると解釈できます。

6.2.2. Debezium Oracle 変更イベントの値

変更イベントメッセージの値の構造は、メッセージの変更イベントの **メッセージキーの構造** をミラーリングし、スキーマセクションとペイロードセクションの両方が含まれます。

変更イベント値のペイロード

変更イベント値のペイロードセクションの **エンベロープ** 構造には、以下のフィールドが含まれます。

op

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。Oracle コネクターの変更イベント値のペイロードの **op** フィールドには、**c** (作成または挿入)、**u** (更新)、**d** (delete)、または **r** (スナップショットを示す read) のいずれかの値が含まれます。

before

任意のフィールド。存在する場合は、イベント発生 **前** の行の状態を記述します。この構造は、**server1.INVENTORY.CUSTOMERS.Value** Kafka Connect スキーマによって記述され、**server1** コネクターは **inventory.customers** テーブルのすべての行に使用します。

after

オプションのフィールドで、存在する場合は、変更が発生した **後** の行の状態が含まれます。構造は、**before** フィールドに使用される **server1.INVENTORY.CUSTOMERS.Value** Kafka Connect スキーマによって記述されます。

source

イベントのソースメタデータを記述する構造体を含む必須フィールド。Oracle コネクタの場合、構造には以下のフィールドが含まれます。

- Debezium のバージョン。
- コネクタ名。
- 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。
- トランザクション ID(スナップショットの含まない)。
- 変更の SCN。
- ソースデータベースのレコードがいつ変更されたかを示すタイムスタンプ(スナップショットの場合、タイムスタンプはスナップショットの発生時刻を示す)。
- 変更を加えたユーザー名

ヒント

commit_scn フィールドは任意で、変更イベントが参加するトランザクションコミットの SCN を記述します。

ts_ms

任意のフィールド。存在する場合は、コネクタがイベントを処理した時間 (Kafka Connect タスクを実行する JVM のシステムクロックがベース) が含まれます。

変更イベント値のスキーマ

イベントメッセージの値のスキーマ部分には、ペイロード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。

変更イベント値の詳細は、以下を参照してください。

- [作成 イベント](#)
- [更新 イベント](#)
- [削除 イベント](#)
- [切り捨て \(truncate\) イベント](#)

作成 イベント

次の例は、[イベントキーの変更](#) 例で説明した **customers** テーブルの **create** イベントの値を示しています。

```
{
  "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.DEBEZIUM.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.DEBEZIUM.CUSTOMERS.Value",  
    "field": "after"  
  }  
]
```

```
},
{
  "type": "struct",
  "fields": [
    {
      "type": "string",
      "optional": true,
      "field": "version"
    },
    {
      "type": "string",
      "optional": false,
      "field": "name"
    },
    {
      "type": "int64",
      "optional": true,
      "field": "ts_ms"
    },
    {
      "type": "string",
      "optional": true,
      "field": "txId"
    },
    {
      "type": "string",
      "optional": true,
      "field": "scn"
    },
    {
      "type": "string",
      "optional": true,
      "field": "commit_scn"
    },
    {
      "type": "string",
      "optional": true,
      "field": "rs_id"
    },
    {
      "type": "int32",
      "optional": true,
      "field": "ssn"
    },
    {
      "type": "int32",
      "optional": true,
      "field": "redo_thread"
    },
    {
      "type": "string",
      "optional": true,
      "field": "user_name"
    },
    {
      "type": "boolean",
```



```

        "optional": true,
        "field": "snapshot"
    }
  ],
  "optional": false,
  "name": "io.debezium.connector.oracle.Source",
  "field": "source"
},
{
  "type": "string",
  "optional": false,
  "field": "op"
},
{
  "type": "int64",
  "optional": true,
  "field": "ts_ms"
}
],
"optional": false,
"name": "server1.DEBEZIUM.CUSTOMERS.Envelope"
},
"payload": {
  "before": null,
  "after": {
    "ID": 1004,
    "FIRST_NAME": "Anne",
    "LAST_NAME": "Kretchmar",
    "EMAIL": "annek@noanswer.org"
  },
  "source": {
    "version": "2.1.4.Final",
    "name": "server1",
    "ts_ms": 1520085154000,
    "txId": "6.28.807",
    "scn": "2122185",
    "commit_scn": "2122185",
    "rs_id": "001234.00012345.0124",
    "ssn": 1,
    "redo_thread": 1,
    "user_name": "user",
    "snapshot": false
  },
  "op": "c",
  "ts_ms": 1532592105975
}
}
}

```

上記の例では、イベントが以下のスキーマを定義する方法に注目してください。

- エンベロープ (`server1.DEBEZIUM.CUSTOMERS.Envelope`)。
- ソース 構造 (`io.debezium.connector.oracle.Source`、Oracle コネクターに固有で、すべてのイベントで再利用)。
- **before** フィールドおよび **after** フィールドのテーブル固有のスキーマ。

ヒント

before フィールドおよび **after** フィールドのスキーマ名は `<logicalName>.<schemaName>.<tableName>.Value` の形式であるため、他のすべてのテーブルのスキーマからは完全に独立しています。その結果、[Avro コンバーター](#) を使用した場合、各論理ソースのテーブルの Avro スキーマは、それぞれ独自に進化し、独自の履歴を持つことになります。

このイベントの値の **payload** 部分には、イベントに関する情報が含まれます。行が作成された (**op=c**) ことを術士、および **after** フィールドの値に、行の **ID**、**FIRST_NAME**、**LAST_NAME**、および **EMAIL** 列に挿入された値が含まれていることを表しています。

ヒント

デフォルトでは、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。サイズが大きいのは、JSON 表現がメッセージのスキーマ部分とペイロード部分の両方を含んでいるためです。[Avro コンバーター](#) を使用すると、コネクタが Kafka トピックに記述するメッセージのサイズを小さくすることができます。

更新イベント

次の例は、前の **create** イベントと同じテーブルからコネクタが捕捉した **更新 変更** イベントを示しています。

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "ID": 1004,
      "FIRST_NAME": "Anne",
      "LAST_NAME": "Kretchmar",
      "EMAIL": "annek@noanswer.org"
    },
    "after": {
      "ID": 1004,
      "FIRST_NAME": "Anne",
      "LAST_NAME": "Kretchmar",
      "EMAIL": "anne@example.com"
    },
    "source": {
      "version": "2.1.4.Final",
      "name": "server1",
      "ts_ms": 1520085811000,
      "txId": "6.9.809",
      "scn": "2125544",
      "commit_scn": "2125544",
      "rs_id": "001234.00012345.0124",
      "ssn": 1,
      "redo_thread": 1,
      "user_name": "user",
      "snapshot": false
    },
    "op": "u",
    "ts_ms": 1532592713485
  }
}
```

ペイロードは `create` (挿入) イベントのペイロードと同じですが、以下の値は異なります。

- **op** フィールドの値が **u** であり、この行が更新により変更されたことを示す。
- **before** フィールドは、**update** データベースのコミット前に存在する値が含まれる行の以前の状態を表示します。
- **after** フィールドは行の更新状態を示しており、**EMAIL** の値は現在 **anne@example.com** に設定されています。
- **source** フィールドの構造は以前と同じフィールドを含みますが、コネクターが REDO ログ内の異なる位置からイベントをキャプチャしたため、値は異なります。
- **ts_ms** フィールドは、Debezium がいつイベントを処理したかを示すタイムスタンプを示す。

payload セクションでは、他に有用な情報を複数示しています。たとえば、**before** と **after** の構造を比較して、コミットの結果として行がどのように変更されたかを判断できます。ソース構造で、この変更の記録に関する情報がわかるので、追跡が可能です。また、このトピックや他のトピックの他のイベントと関連して、このイベントが発生した場合に、見識を得ることができます。これは、別のイベントと同じコミットの前後、または一部として発生していましたか？



注記

行のプライマリーキー/一意キーの列を更新すると、行のキーの値が変更されます。その結果、Debezium はこのような更新後に 3 つのイベントを出力します。

- **DELETE** イベント。
- 行のキーが古い [tombstone イベント](#)
- 行に新しいキーを提供する **INSERT** イベント。

delete イベント

次の例は、先の `create` と `update` の イベント例で示したテーブルの `delete` イベントを示しています。delete イベントの **schema** 部分は、これらのイベントの **schema** 部分と同一です。

```
{
  "schema": { ... },
  "payload": {
    "before": {
      "ID": 1004,
      "FIRST_NAME": "Anne",
      "LAST_NAME": "Kretchmar",
      "EMAIL": "anne@example.com"
    },
    "after": null,
    "source": {
      "version": "2.1.4.Final",
      "name": "server1",
      "ts_ms": 1520085153000,
      "txId": "6.28.807",
      "scn": "2122184",
      "commit_scn": "2122184",
      "rs_id": "001234.00012345.0124",
      "ssn": 1,

```

```

    "redo_thread": 1,
    "user_name": "user",
    "snapshot": false
  },
  "op": "d",
  "ts_ms": 1532592105960
}
}

```

イベントの **payload** 部分は、**create** または **update** イベントのペイロードと比較して、いくつかの違いを示しています。

- **op** フィールドの値は **d** であり、行が削除されたことを意味します。
- **before** フィールドは、データベースのコミットで削除された行の旧状態を示します。
- **after** フィールドの値が **null** 場合、その行はもはや存在しないことを意味します。
- **source** フィールドの構造には、**create** または **update** イベントに存在する多くのキーが含まれるが、**ts_ms**、**scn**、**txld** フィールドの値は異なっている。
- **ts_ms** は、Debezium がこのイベントを処理したタイミングを示すタイムスタンプを示します。

delete イベントは、コンシューマーがこの行の削除を処理するために必要な情報を提供する。

Oracle コネクタのイベントは、[Kafka ログコンパクション](#) と連携するように設計されており、すべてのキーで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。

行が削除されると、Kafka は同じキーを使用する以前のメッセージをすべて削除できるため、前の例で示された **delete** イベント値は、ログ圧縮でまだ機能します。同じキーを共有する **メッセージをすべて削除** するように Kafka に指示するには、メッセージの値を **null** に設定する必要があります。これを可能にするにはデフォルトで、Debezium Oracle コネクタは、値が **null** 以外で同じキーを持つ特別な **廃棄** イベントが含まれる **削除** イベントに従います。コネクタプロパティ **tombstones.on.delete** を設定すると、デフォルトの動作を変更できます。

切り捨て (truncate) イベント

切り捨て (truncate) 変更イベントは、テーブルが切り捨てられていることを伝えます。この場合のメッセージキーは **null** で、メッセージの値は以下のようになります。

```

{
  "schema": { ... },
  "payload": {
    "before": null,
    "after": null,
    "source": { ①
      "version": "2.1.4.Final",
      "connector": "oracle",
      "name": "oracle_server",
      "ts_ms": 1638974535000,
      "snapshot": "false",
      "db": "ORCLPDB1",
      "sequence": null,
      "schema": "DEBEZIUM",

```

```

"table": "TEST_TABLE",
"txId": "02000a0037030000",
"scn": "13234397",
"commit_scn": "13271102",
"lcr_position": null,
"rs_id": "001234.00012345.0124",
"ssn": 1,
"redo_thread": 1,
"user_name": "user"
},
"op": "t", ②
"ts_ms": 1638974558961, ③
"transaction": null
}
}

```

表6.6 切り捨て (truncate) イベント値フィールドの説明

項目	フィールド名	説明
1	source	<p>イベントのソースメタデータを記述する必須のフィールド。切り捨て (truncate) イベント値の source フィールド構造は、同じテーブルの作成、更新、および削除 イベントと同じで、以下のメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● 新しい行が含まれるデータベースおよびテーブル ● スキーマ名 ● イベントがスナップショットの一部である場合 (truncate イベントでは常にfalse) ● 操作が実行されたトランザクションの ID ● 操作の SCN ● データベースに変更が加えられた時点のタイムスタンプ ● 変更を行ったユーザー名
2	op	<p>操作の型を記述する必須の文字列。op フィールドの値はtで、このテーブルが切り捨てされたことを示します。</p>
3	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

切り捨て (`truncate`) イベントは、テーブル全体に加えられた変更を表し、メッセージキーを持たないため、コンシューマーが切り捨てられたイベントや変更イベントを受信する保証はありません (作成、更新など)。例えば、コンシューマーが異なるパーティションからイベントを読み込む場合、同じテーブルの `truncate` イベントを受信した後に、テーブルの `update` イベントを受信することがあります。順序は、トピックが単一のパーティションを使用する場合にのみ保証されます。

切り捨て (`truncate`) イベントをキャプチャーしない場合は、`skipped.operations` オプションを使用して除外します。

6.3. DEBEZIUM ORACLE コネクタによるデータ型のマッピング方法

Debezium Oracle コネクタは、テーブル行の値の変化を検出すると、その変化を表す `change` イベントを発行します。各変更イベントレコードは、元のテーブルと同じように構造化されており、イベントレコードには列値ごとにフィールドが含まれます。テーブル列のデータ型は、以下のセクションの表に示すように、コネクタが変更イベントフィールドで列の値をどのように表現するかを決定します。

テーブルの各列に対して、Debezium はソースデータ型に対応するイベントフィールドの **リテラル型**、場合によっては **セマンティック型** にマッピングします。

リテラル型

以下のカフカコネクトスキーマタイプのいずれかを使用して、値が文字通りどのように表現されるかを記述します。 **INT8**、 **INT16**、 **INT32**、 **INT64**、 **FLOAT32**、 **FLOAT64**、 **BOOLEAN**、 **STRING**、 **BYTES**、 **ARRAY**、 **MAP**、 および **STRUCT**。

セマンティック型

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。

デフォルトのデータ型変換が要件に合わない場合は、コネクタ用の **カスタムコンバータの作成** が可能です。

一部の Oracle ラージオブジェクト (CLOB、NCLOB、BLOB) および数値データ型については、デフォルトの設定プロパティを変更することにより、コネクタがタイプマッピングを実行する方法を操作することができます。Debezium プロパティがこれらのデータ型のマッピングをどのように制御するかの詳細については、 **Binary and Character LOB types** および **Numeric types** をご覧ください。

Debezium コネクタによる Oracle データ型のマッピング方法に関する詳細は、以下を参照してください。

- [文字タイプ](#)
- [バイナリーおよび文字の LOB 型](#)
- [数字型](#)
- [ブール値型](#)
- [時間型](#)
- [ROWID タイプ](#)
- [ユーザー定義のタイプ](#)
- [Oracle によって提供されたタイプ](#)
- [デフォルト値](#)

文字タイプ

以下の表は、コネクターによる基本の文字タイプへのマッピング方法を説明しています。

表6.7 Oracle 基本文字型のマッピング

Oracle 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) および注記
CHAR[(M)]	STRING	該当なし
NCHAR[(M)]	STRING	該当なし
NVARCHAR2[(M)]	STRING	該当なし
VARCHAR[(M)]	STRING	該当なし
VARCHAR2[(M)]	STRING	該当なし



バイナリーおよび文字の LOB 型

Debezium Oracle コネクターでの **BLOB**、**CLOB**、**NCLOB** の使用は、技術プレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行い、フィードバックを提供していただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、[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。

以下の表は、コネクターによるバイナリーおよび文字 LOB (Large Object) 型へのマッピング方法を説明しています。

表6.8 Oracle バイナリーおよび文字 LOB 型のマッピング

Oracle 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) および注記
BFILE	該当なし	このデータ型はサポートされていません。
BLOB	BYTES	binary.handling.mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。
CLOB	STRING	該当なし
LONG	該当なし	このデータタイプはサポートされていません。
LONG RAW	該当なし	このデータタイプはサポートされていません。

Oracle 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) および注記
NCLOB	STRING	該当なし
RAW	該当なし	このデータタイプはサポートされていません。

注記

Oracle は、**CLOB**、**NCLOB**、および **BLOB** データタイプの列値を、SQL ステートメントで明示的に設定または変更された場合にのみ供給します。その結果、変更イベントには、変更されていない **CLOB**、**NCLOB**、または **BLOB** 列の値が含まれることはありません。代わりに、コネクタプロパティ **unavailable.value.placeholder** で定義されているプレースホルダーが含まれます。

CLOB、**NCLOB**、または **BLOB** 列の値が更新されると、対応する更新変更イベントの **after** 要素に新しい値が追加されます。**before** 要素には使用できない値プレースホルダーが含まれます。

数字型

以下の表は、Debezium Oracle コネクタによる数値型のマッピング方法を説明しています。

注記

コネクタの **decimal.handling.mode** 設定プロパティの値を変更することで、コネクタが Oracle **DECIMAL**、**NUMBER**、**NUMERIC**、および **REAL** データ型をマッピングする方法を変更できます。このプロパティをデフォルト値の **precise** に設定すると、コネクタは、表に示すように、これらの Oracle データ型を Kafka Connect **org.apache.kafka.connect.data.Decimal** 論理型にマッピングします。プロパティの値を **double** または **string** に設定すると、コネクタは一部の Oracle データ型に別のマッピングを使用します。詳しくは、以下の表の **セマンティックタイプおよび注意** 事項の欄をご覧ください。

表6.9 Oracle 数値データ型のマッピング

Oracle 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) および注記
BINARY_FLOAT	FLOAT32	該当なし
BINARY_DOUBLE	FLOAT64	該当なし

Oracle 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) および注記
DECIMAL[(P, S)]	BYTES / INT8 / INT16 / INT32 / INT64	<p><code>org.apache.kafka.connect.data.Decimal</code> BYTES を使用する場合:</p> <p>NUMBER と同様に処理されます (DECIMAL の場合には S は 0 に初期設定されます)。</p> <p><code>decimal.handling.mode</code> プロパティが <code>double</code> に設定されている場合、コネクターは DECIMAL 値をスキーマタイプ <code>FLOAT64</code> の Java <code>double</code> 値として表現する。</p> <p><code>decimal.handling.mode</code> プロパティが <code>string</code> に設定されている場合、コネクターは DECIMAL 値をスキーマタイプ <code>STRING</code> でフォーマットされた文字列表現として表します。</p>
DOUBLE PRECISION	STRUCT	<p><code>io.debezium.data.VariableScaleDecimal</code></p> <p>転送された値のスケールが含まれる <code>INT32</code> 型の <code>scale</code> と、元の値がスケールされていない形式で含まれる <code>BYTES</code> 型の <code>value</code> の 2 つのフィールドがある構造が含まれます。</p>
FLOAT[(P)]	STRUCT	<p><code>io.debezium.data.VariableScaleDecimal</code></p> <p>転送された値のスケールが含まれる <code>INT32</code> 型の <code>scale</code> と、元の値がスケールされていない形式で含まれる <code>BYTES</code> 型の <code>value</code> の 2 つのフィールドがある構造が含まれます。</p>
INTEGER, INT	BYTES	<p><code>org.apache.kafka.connect.data.Decimal</code></p> <p>INTEGER は Oracle で <code>NUMBER(38,0)</code> にマップされるため、<code>INT</code> タイプよりも大きな値を保持することができます。</p>
NUMBER[(P[, *])]	STRUCT	<p><code>io.debezium.data.VariableScaleDecimal</code></p> <p>転送された値のスケールが含まれる <code>INT32</code> 型の <code>scale</code> と、元の値がスケールされていない形式で含まれる <code>BYTES</code> 型の <code>value</code> の 2 つのフィールドがある構造が含まれます。</p> <p><code>decimal.handling.mode</code> プロパティを <code>double</code> に設定すると、コネクターは NUMBER の値をスキーマタイプが <code>FLOAT64</code> の Java <code>double</code> 値として表します。</p> <p><code>decimal.handling.mode</code> プロパティが <code>string</code> に設定された場合、コネクターは NUMBER の値をスキーマ型 <code>STRING</code> でフォーマットされた文字列表現として表します。</p>

Oracle 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) および注記
NUMBER(P, S <= 0)	INT8 / INT16 / INT32 / INT64	<p>スケールが 0 の NUMBER 列は、整数を表します。負のスケールは Oracle での丸めを表します。たとえば、スケールが -2 の場合には、数百に丸められます。</p> <p>以下のように、精度とスケールリングに応じて、一致する Kafka Connect の整数タイプのいずれかが選択されます。</p> <ul style="list-style-type: none"> ● P - S < 3, INT8 ● P - S < 5, INT16 ● P - S < 10, INT32 ● P - S < 19, INT64 ● P - S >= 19, BYTES (org.apache.kafka.connect.data.Decimal) <p>decimal.handling.mode プロパティを double に設定すると、コネクタは NUMBER の値をスキーマタイプが FLOAT64 の Java double 値として表します。</p> <p>decimal.handling.mode プロパティが string に設定された場合、コネクタは NUMBER の値をスキーマ型 STRING でフォーマットされた文字列表現として表します。</p>
NUMBER(P, S > 0)	BYTES	org.apache.kafka.connect.data.Decimal
NUMERIC[(P, S)]	BYTES / INT8 / INT16 / INT32 / INT64	<p>org.apache.kafka.connect.data.Decimal BYTES を使用する場合:</p> <p>NUMBER と同様に処理されます (NUMERIC の場合には S は 0 に初期設定されます)。</p> <p>decimal.handling.mode プロパティを double に設定すると、コネクタは NUMERIC の値をスキーマタイプが FLOAT64 の Java double 値として表します。</p> <p>decimal.handling.mode プロパティが string に設定された場合、コネクタは NUMERIC の値をスキーマ型 STRING でフォーマットされた文字列表現として表します。</p>
SMALLINT	BYTES	<p>org.apache.kafka.connect.data.Decimal</p> <p>SMALLINT は Oracle で NUMBER(38,0) にマップされるため、INT タイプよりも大きな値を保持することができます。</p>

Oracle 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) および注記
REAL	STRUCT	<p>io.debezium.data.VariableScaleDecimal</p> <p>転送された値のスケールが含まれる INT32 型の scale と、元の値がスケールされていない形式で含まれる BYTES 型の value の 2 つのフィールドがある構造が含まれます。</p> <p>decimal.handling.mode プロパティが double に設定されている場合、コネクタは REAL 値をスキーマタイプ FLOAT64 の Java double 値として表現する。</p> <p>decimal.handling.mode プロパティが string に設定されている場合、コネクタは REAL 値をスキーマタイプ STRING でフォーマットされた文字列表現として表します。</p>

前述したように、Oracle では **NUMBER** 型において負のスケールが可能です。これが原因で、数値が **Decimal** として表示される場合に、Avro 形式への変換中に問題が発生する可能性があります。10 進数タイプにはスケール情報が含まれますが、[Avro 仕様](#) ではスケールの正の値しか使用できません。使用されるスキーマレジストリーによっては、Avro シリアル化に失敗する場合があります。この問題を回避するには、**NumberToZeroScaleConverter** を使用できます。これは、スケールが負で、十分に大きな数値 ($P - S \geq 19$) をスケール 0 の **Decimal** 型に変換します。以下のように設定できます。

```
converters=zero_scale
zero_scale.type=io.debezium.connector.oracle.converters.NumberToZeroScaleConverter
zero_scale.decimal.mode=precise
```

デフォルトでは、数値は **Decimal** 型 (**zero_scale.decimal.mode=precise**) に変換されますが、サポート対象の残りの 2 つの型 (**double** と **string**) もサポートすることですべてに対応します。

ブール値型

Oracle は、**BOOLEAN** データ型のネイティブサポートを提供しません。ただし、論理 **BOOLEAN** データ型の概念をシミュレートするために、特定のセマンティクスと他のデータ型を使用することが一般的です。

ソース列をブールデータ型に変換できるように、Debezium は **NumberOneToBooleanConverter** [カスタムコンバーター](#) を提供しており、以下のいずれかの方法で使用することができます。

- すべての **NUMBER(1)** 列を **BOOLEAN** タイプにマッピングします。
- 正規表現のコンマ区切りリストを使用して、列のサブセットを列挙します。このタイプの変換を使用するには、以下の例のように **selector** パラメーターを使用して **converters** 設定プロパティを設定する必要があります。

```
converters=boolean
boolean.type=io.debezium.connector.oracle.converters.NumberOneToBooleanConverter
boolean.selector=.*MYTABLE.FLAG,.*.IS_ARCHIVED
```

時間型

Oracle **INTERVAL**、**TIMESTAMP WITH TIME ZONE**、および **TIMESTAMP WITH LOCAL TIME ZONE** データ型以外では、コネクタが時間型を変換する方法は **time.precision.mode** 設定プロパティの値に依存します。

time.precision.mode 設定プロパティが **adaptive** (デフォルト) に設定された場合、コネクタ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。

Oracle 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) および注記
DATE	INT64	<p>io.debezium.time.Timestamp</p> <p>UNIX 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。</p>
INTERVAL DAY[(M)] TO SECOND	FLOAT64	<p>io.debezium.time.MicroDuration</p> <p>365.25 / 12.0 の計算式で月平均日数を算出した時間間隔の微小秒数です。</p> <p>io.debezium.time.Interval (interval.handling.mode が string に設定されている場合)</p> <p>P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S のパターンに従う区間値の文字列表現、例えば P1Y2M3DT4H5M6.78S など。</p>
INTERVAL YEAR[(M)] TO MONTH	FLOAT64	<p>io.debezium.time.MicroDuration</p> <p>365.25 / 12.0 の計算式で月平均日数を算出した時間間隔の微小秒数です。</p> <p>io.debezium.time.Interval (interval.handling.mode が string に設定されている場合)</p> <p>P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S のパターンに従う区間値の文字列表現、例えば P1Y2M3DT4H5M6.78S など。</p>
TIMESTAMP(0 - 3)	INT64	<p>io.debezium.time.Timestamp</p> <p>UNIX 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。</p>
TIMESTAMP, TIMESTAMP(4 - 6)	INT64	<p>io.debezium.time.MicroTimestamp</p> <p>UNIX 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。</p>
TIMESTAMP(7 - 9)	INT64	<p>io.debezium.time.NanoTimestamp</p> <p>UNIX エポックからのナノ秒数を表し、タイムゾーン情報は含まれません。</p>

Oracle 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) および注記
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。
TIMESTAMP WITH LOCAL TIME ZONE	STRING	io.debezium.time.ZonedTimestamp UTC のタイムスタンプの文字列表現。

time.precision.mode 設定プロパティが **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。Oracle がサポートする精度レベルは、Kafka Connect サポートの論理型を超過するため、**time.precision.mode** を **connect** に設定していて、データベース列の **fractional second precision** の値が 3 より大きい場合には **a loss of precision** という結果になります。

Oracle 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) および注記
DATE	INT32	org.apache.kafka.connect.data.Date UNIX エポックからの日数を表します。
INTERVAL DAY[(M)] TO SECOND	FLOAT64	io.debezium.time.MicroDuration 365.25 / 12.0 の計算式で月平均日数を算出した時間間隔の微小秒数です。 io.debezium.time.Interval (interval.handling.mode が string に設定されている場合) P<years>Y<months>M<days>DT<hours>H<minute>s<M<second>S のパターンに従う区間値の文字列表現、例えば P1Y2M3DT4H5M6.78S など。
INTERVAL YEAR[(M)] TO MONTH	FLOAT64	io.debezium.time.MicroDuration 365.25 / 12.0 の計算式で月平均日数を算出した時間間隔の微小秒数です。 io.debezium.time.Interval (interval.handling.mode が string に設定されている場合) P<years>Y<months>M<days>DT<hours>H<minute>s<M<second>S のパターンに従う区間値の文字列表現、例えば P1Y2M3DT4H5M6.78S など。

Oracle 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) および注記
TIMESTAMP(0 - 3)	INT64	org.apache.kafka.connect.data.Timestamp UNIX 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 - 6)	INT64	org.apache.kafka.connect.data.Timestamp UNIX 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(7 - 9)	INT64	org.apache.kafka.connect.data.Timestamp UNIX 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 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。
TIMESTAMP WITH LOCAL TIME ZONE	STRING	io.debezium.time.ZonedTimestamp UTC のタイムスタンプの文字列表現。

ROWID タイプ

次の表は、コネクタが ROWID (行アドレス) データ型をどのようにマッピングするかを説明したものです。

表6.10 Oracle ROWID データタイプのマッピング

Oracle 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) および注記
ROWID	STRING	該当なし
UROWID	該当なし	このデータ型はサポートされていません。

ユーザー定義のタイプ

オラクルでは、組み込みのデータ型では要件を満たせない場合に、カスタムデータ型を定義して柔軟性を持たせることができます。オブジェクト型、REF データ型、Varrays、Nested Tables などのユーザー定義型があります。現時点では、Debezium Oracle コネクタをこれらのユーザー定義タイプで使用することはできません。

Oracle によって提供されたタイプ

Oracle は、組み込み型や ANSI でサポートされている型では不十分な場合に、新しい型を定義するために使用できる SQL ベースのインタフェースを提供しています。Oracle は、**任意の**、XML、または

Spatial 型など、幅広い目的に対応するために一般的に使用されるデータ型をいくつか提供していません。現時点では、Debezium Oracle コネクタはこれらのデータ型では使用できません。

デフォルト値

データベーススキーマの列にデフォルト値が指定されている場合、Oracle コネクタはこの値に対応する Kafka レコードフィールドのスキーマに伝搬させようと試みます。ほとんどの一般的なデータタイプがサポートされています。

- 文字型 (**CHAR**、**NCHAR**、**VARCHAR**、**VARCHAR2**、**NVARCHAR**、**NVARCHAR2**)
- 数値型 (**INTEGER**、**NUMERIC**、など)
- 時間型 (**DATE**、**TIMESTAMP**、**INTERVAL** など)。

一時的なタイプが **TO_TIMESTAMP** や **TO_DATE** などの関数呼び出しを使用してデフォルト値を表す場合、コネクタは関数を評価するために追加のデータベース呼び出しを行うことでデフォルト値を解決します。例えば、**DATE** 列が **TO_DATE('2021-01-02', 'YYYY-MM-DD')** というデフォルト値で定義されている場合、その列のデフォルト値はその日付の UNIX エポックからの日数、この場合は **18629** となります。

一時的な型がデフォルト値を表すために **SYSDATE** 定数を使用する場合、コネクタは、列が **NOT NULL** または **NULL** として定義されているかどうかに基づいてこれを解決します。カラムが **NULL** 可能な場合、デフォルト値は設定されません。しかし、カラムが **NULL** 可能でない場合、デフォルト値は **0** (**DATE** または **TIMESTAMP(n)** データ型の場合) または **1970-01-01T00:00:00Z** (**TIMESTAMP WITH TIME ZONE** または **TIMESTAMP WITH LOCAL TIME ZONE** データ型の場合) のいずれかに解決されます。デフォルトの値のタイプは数値です。ただし、カラムが **TIMESTAMP WITH TIME ZONE** または **TIMESTAMP WITH LOCAL TIME ZONE** の場合は文字列として出力されます。

6.4. DEBEZIUM と連携させるための ORACLE の設定

Debezium Oracle コネクタと連携するように Oracle を設定するには、以下の手順が必要です。以下の手順では、コンテナデータベースと少なくとも1つのプラグ可能なデータベースでマルチテナンシーの設定を使用することを前提としています。マルチテナント設定を使用しない場合は、以下の手順を調整する必要がある場合があります。

Debezium コネクタと使用するために Oracle を設定する場合の詳細は、以下を参照してください。

- [「Oracle インストールタイプとの Debezium Oracle コネクタの互換性」](#)
- [「変更イベントをキャプチャーする際に Debezium Oracle コネクタが除外するスキーマ」](#)
- [「Debezium で使用する Oracle データベースの準備」](#)
- [「データディクショナリーに対応するように Oracle redo ログのサイズを変更」](#)
- [「Debezium Oracle コネクタの Oracle ユーザーの作成」](#)
- [「Oracle standby データベースのサポート」](#)

6.4.1. Oracle インストールタイプとの Debezium Oracle コネクタの互換性

Oracle データベースは、スタンドアロンインスタンスまたは Oracle Real Application Cluster (RAC) を使用してインストールできます。Debezium Oracle コネクタはどちらのタイプのインストールとも互換性があります。

6.4.2. 変更イベントをキャプチャーする際に Debezium Oracle コネクタが除外するスキーマ

Debezium Oracle コネクタがテーブルをキャプチャーすると、以下のスキーマからテーブルが自動的に除外されます。

- **appqosys**
- **audsys**
- **ctxsys**
- **dvsys**
- **dbsfwuser**
- **dbsnmp**
- **qsmadmin_internal**
- **lbacsys**
- **mdsys**
- **ojvmsys**
- **olapsys**
- **orddata**
- **ordsys**
- **outln**
- **sys**
- システム
- **wmsys**
- **xdb**

コネクタがテーブルからの変更をキャプチャできるようにするには、そのテーブルが前述のリストにない名前のスキーマを使用している必要があります。

6.4.3. 変更イベントをキャプチャーする際に Debezium Oracle コネクタが除外するテーブル

Debezium Oracle コネクタがテーブルをキャプチャーすると、以下のルールに一致するテーブルが自動的に除外されます。

- **CMP[3|4]\$[0-9]+** に一致する圧縮アドバイザーテーブル。
- パターン **SYS_IOT_OVER_%** に一致するインデックスで整理された表。
- **MDRT_%**、**MDRS_%**、または **MDXT_%** パターンと一致する空間テーブル。

- ネストされたテーブル

コネクターが前述のルールの一つに一致する名前のテーブルをキャプチャーできるようにするには、テーブルの名前を変更する必要があります。

6.4.4. Debezium で使用する Oracle データベースの準備

Oracle LogMiner に必要な設定

```
ORACLE_SID=ORACLCDB dbz_oracle sqlplus /nolog

CONNECT sys/top_secret AS SYSDBA
alter system set db_recovery_file_dest_size = 10G;
alter system set db_recovery_file_dest = '/opt/oracle/oradata/recovery_area' scope=spfile;
shutdown immediate
startup mount
alter database archivelog;
alter database open;
-- Should now "Database log mode: Archive Mode"
archive log list

exit;
```

Oracle AWS RDS では、上記のコマンドを実行することも、sysdba としてログインすることもできません。AWS には、LogMiner 設定に使用可能な代替のコマンドが含まれます。これらのコマンドを実行する前に、Oracle AWS RDS インスタンスでバックアップが有効になっていることを確認してください。

Oracle でバックアップが有効になっていることを確認するには、最初に次のコマンドを実行します。LOG_MODE は ARCHIVELOG となるはずですが、そうでない場合は、Oracle AWS RDS インスタンスの再起動が必要になる場合があります。

Oracle AWS RDS LogMiner に必要な設定

```
SQL> SELECT LOG_MODE FROM V$DATABASE;

LOG_MODE
-----
ARCHIVELOG
```

LOG_MODE が ARCHIVELOG に設定されたら、コマンドを実行して LogMiner 設定を完了します。最初のコマンドはデータベースを archivelogs に設定し、2 番目のコマンドは補足ロギングを追加します。

Oracle AWS RDS LogMiner に必要な設定

```
exec rdsadmin.rdsadmin_util.set_configuration('archivelog retention hours',24);

exec rdsadmin.rdsadmin_util.alter_supplemental_logging('ADD');
```

Debezium がデータベース行の変更 **前** の状態をキャプチャできるようにするには、キャプチャしたテーブルまたはデータベース全体の補足ロギングも有効にする必要があります。次の例は、1 つの **inventory.customers** テーブルのすべての列に対して補足的なログを設定する方法を示しています。

```
ALTER TABLE inventory.customers ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

すべてのテーブル列の補助ロギングを有効にすると、Oracle redo ログのボリュームが増えます。ログのサイズに過剰な増加を防ぐには、前述の設定を選択的に適用します。

補完用のロギングは最小限のデータベースレベルで有効にする必要があります、以下のように設定できません。

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

6.4.5. データディクショナリーに対応するように Oracle redo ログのサイズを変更

データベースの設定によっては、REDO ログのサイズや数が、許容できるパフォーマンスを得るために十分でない場合があります。Debezium Oracle コネクタを設定する前に、REDO ログの容量がデータベースをサポートするのに十分であることを確認してください。

データベースの REDO ログの容量は、そのデータディクショナリーを保存するのに十分でなければなりません。一般的に、データディクショナリーのサイズは、データベースのテーブルや列の数に応じて大きくなります。REDO ログの容量が十分でない場合、データベースと Debezium コネクタの両方でパフォーマンスの問題が発生する可能性があります。

データベースのログ容量を増やす必要があるかどうかは、データベース管理者に相談してください。

6.4.6. Debezium Oracle コネクタの Oracle ユーザーの作成

Debezium Oracle コネクタが変更イベントをキャプチャーするには、特定のパーミッションを持つ Oracle LogMiner ユーザーとして実行する必要があります。以下の例は、マルチテナントデータベースモデルでコネクタの Oracle ユーザーアカウントを作成する SQL を示しています。



警告

コネクタは、自分の Oracle ユーザーアカウントによって行われたデータベースの変更をキャプチャします。ただし、**SYS** や **SYSTEM** のユーザーアカウントで行われた変更は捕捉できません。

コネクタの LogMiner ユーザーの作成

```
sqlplus sys/top_secret@//localhost:1521/ORCLCDB as sysdba
CREATE TABLESPACE logminer_tbs DATAFILE '/opt/oracle/oradata/ORCLCDB/logminer_tbs.dbf'
SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
exit;
```

```
sqlplus sys/top_secret@//localhost:1521/ORCLPDB1 as sysdba
CREATE TABLESPACE logminer_tbs DATAFILE
'/opt/oracle/oradata/ORCLCDB/ORCLPDB1/logminer_tbs.dbf'
SIZE 25M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
exit;
```

```
sqlplus sys/top_secret@//localhost:1521/ORCLCDB as sysdba
```

```

CREATE USER c##dbzuser IDENTIFIED BY dbz
  DEFAULT TABLESPACE logminer_tbs
  QUOTA UNLIMITED ON logminer_tbs
  CONTAINER=ALL;

GRANT CREATE SESSION TO c##dbzuser CONTAINER=ALL; 1
GRANT SET CONTAINER TO c##dbzuser CONTAINER=ALL; 2
GRANT SELECT ON V_$DATABASE TO c##dbzuser CONTAINER=ALL; 3
GRANT FLASHBACK ANY TABLE TO c##dbzuser CONTAINER=ALL; 4
GRANT SELECT ANY TABLE TO c##dbzuser CONTAINER=ALL; 5
GRANT SELECT_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL; 6
GRANT EXECUTE_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL; 7
GRANT SELECT ANY TRANSACTION TO c##dbzuser CONTAINER=ALL; 8
GRANT LOGMINING TO c##dbzuser CONTAINER=ALL; 9

GRANT CREATE TABLE TO c##dbzuser CONTAINER=ALL; 10
GRANT LOCK ANY TABLE TO c##dbzuser CONTAINER=ALL; 11
GRANT CREATE SEQUENCE TO c##dbzuser CONTAINER=ALL; 12

GRANT EXECUTE ON DBMS_LOGMNR TO c##dbzuser CONTAINER=ALL; 13
GRANT EXECUTE ON DBMS_LOGMNR_D TO c##dbzuser CONTAINER=ALL; 14

GRANT SELECT ON V_$LOG TO c##dbzuser CONTAINER=ALL; 15
GRANT SELECT ON V_$LOG_HISTORY TO c##dbzuser CONTAINER=ALL; 16
GRANT SELECT ON V_$LOGMNR_LOGS TO c##dbzuser CONTAINER=ALL; 17
GRANT SELECT ON V_$LOGMNR_CONTENTS TO c##dbzuser CONTAINER=ALL; 18
GRANT SELECT ON V_$LOGMNR_PARAMETERS TO c##dbzuser CONTAINER=ALL; 19
GRANT SELECT ON V_$LOGFILE TO c##dbzuser CONTAINER=ALL; 20
GRANT SELECT ON V_$ARCHIVED_LOG TO c##dbzuser CONTAINER=ALL; 21
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO c##dbzuser CONTAINER=ALL; 22
GRANT SELECT ON V_$TRANSACTION TO c##dbzuser CONTAINER=ALL; 23

exit;

```

表6.11 パーミッション/付与の説明

項目	ロール名	説明
1	CREATE SESSION	コネクターが Oracle に接続できるようにします。
2	SET CONTAINER	コネクターがプラグ可能なデータベース間の切り替えを可能にします。これは、Oracle インストールでコンテナデータベースのサポート (CDB) が有効になっている場合にのみ必要です。
3	SELECT ON V_\$DATABASE	コネクターによる V\$DATABASE テーブルの読み取りが可能になります。

項目	ロール名	説明
4	FLASHBACK ANY TABLE	コネクターがデータの初期スナップショットを実行する方法であるフラッシュバッククエリーを実行できるようにします。
5	SELECT ANY TABLE	コネクターで任意のテーブルを読み込めるようにする。
6	SELECT_CATALOG_ROLE	Oracle LogMiner セッションで必要とされるデータディクショナリーをコネクターで読み込めるようにします。
7	EXECUTE_CATALOG_ROLE	コネクターがデータディクショナリーを Oracle redo ログに書き込むことを可能にします。これは、スキーマの変更を追跡するために必要なものです。
8	SELECT ANY TRANSACTION	スナップショットプロセスで、任意のトランザクションに対してフラッシュバックスナップショットクエリーを実行できるようにします。 FLASHBACK ANY TABLE が付与されている場合、これも付与されるべきです。
9	LOGMINING	ロールこのロールは、Oracle LogMiner とそのパッケージへの完全なアクセスを付与する方法として、新しいバージョンの Oracle で追加されました。このロールのない古いバージョンの Oracle では、この付与を無視できます。
10	CREATE TABLE	コネクターがそのデフォルトのテーブルスペースにフラッシュテーブルを作成することを有効にします。フラッシュテーブルにより、LGWR 内部バッファのディスクへのフラッシュをコネクターが明示的に制御することができます。
11	LOCK ANY TABLE	スキーマスナップショットの実行中にコネクターがテーブルをロックできるようにします。スナップショットロックが設定により明示的に無効化されている場合、このグラントは安全に無視することができます。
12	CREATE SEQUENCE	コネクターがそのデフォルトのテーブルスペースにシーケンスを作成することを有効にします。

項目	ロール名	説明
13	EXECUTE ON DBMS_LOGMNR	DBMS_LOGMNR パッケージのメソッドをコネクタで実行できるようにします。これは Oracle LogMiner との対話に必要です。Oracle の新しいバージョンでは、 LOGMINING ロールによってこの権限が与えられますが、古いバージョンでは、明示的に付与する必要があります。
14	EXECUTE ON DBMS_LOGMNR_D	DBMS_LOGMNR_D パッケージのメソッドをコネクタで実行できるようにします。これは Oracle LogMiner との対話に必要です。Oracle の新しいバージョンでは、 LOGMINING ロールによってこの権限が与えられますが、古いバージョンでは、明示的に付与する必要があります。
15 から 23	SELECT ON V_\$...	コネクタがこれらのテーブルを読み取れることを可能にします。Oracle LogMiner セッションを準備するために、コネクタは Oracle redo およびアーカイブログ、および現在のトランザクションの状態に関する情報を読み取れる必要があります。これらの付与がないと、コネクタは操作できません。

6.4.7. Oracle standby データベースのサポート

Debezium Oracle コネクタは Oracle 物理または論理スタンバイデータベースとは使用できません。

6.5. DEBEZIUM ORACLE コネクタのデプロイメント

以下の方法のいずれかを使用して Debezium Oracle コネクタをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタプラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。



重要

Debezium Oracle コネクタのアーカイブには、ライセンスの関係で、Oracle データベースに接続するために必要な Oracle JDBC ドライバーが含まれていません。コネクタがデータベースにアクセスできるようにするには、コネクタ環境にドライバーを追加する必要があります。詳細は、[Obtaining the Oracle JDBC driver](#) を参照してください。

関連情報

- [「Debezium Oracle コネクタ設定プロパティの説明」](#)

6.5.1. Oracle JDBC ドライバーの取得

Debezium が Oracle データベースに接続するために必要な Oracle JDBC ドライバーファイルは、ライセンスの関係で Debezium Oracle コネクタアーカイブに含まれていません。ドライバーは、Maven

Central からダウンロード可能です。使用するデプロイメント方法に応じて、Kafka Connect カスタムリソースまたはコネクタイメージの構築に使用する Dockerfile にコマンドを追加して、ドライバーを取得することができます。

- AMQ Streams を使用して Kafka Connect イメージにコネクタを追加する場合は、「[AMQ Streams を使用した Debezium Oracle コネクタのデプロイ](#)」に示すように、**KafkaConnect** カスタムリソースの **builds.plugins.artifact.url** にドライバーの Maven Central の場所を追加してください。
- Dockerfile を使用してコネクタ用のコンテナイメージを構築する場合、Dockerfile に **curl** コマンドを挿入して、Maven Central から必要なドライバーファイルをダウンロードするための URL を指定します。詳細には、[Dockerfile からカスタム Kafka Connect コンテナイメージを構築して Debezium Oracle コネクタをデプロイ](#) するを参照してください。

6.5.2. AMQ Streams を使用した Debezium Oracle コネクタのデプロイメント

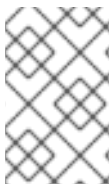
Debezium 1.7 以降、Debezium コネクタのデプロイに推奨される方法は、AMQ Streams を使用してコネクタプラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクタアーティファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクタがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクタを開始します。

Kafka Connect イメージのビルド仕様では、デプロイ可能なコネクタを指定できます。各コネクタプラグインに対して、デプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーティファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をデプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクタを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。
- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

6.5.3. AMQ Streams を使用した Debezium Oracle コネクタのデプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクタをデプロイするには、最

初にコネクタ用の Kafka Connect イメージをビルドする必要がありました。コネクタを OpenShift にデプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクタプラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクタ定義を含む **KafkaConnect** カスタムリソースの入力パラメータを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリまたは別の設定済みの HTTP サーバーから必要なアーティファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのデプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクタを起動します。

前提条件

- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスターが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合

- [ImageStream](#) リソースを新規コンテナイメージを保存するためにクラスターにデプロイします。クラスターの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスターにログインします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例6.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium Oracle コネクタアーカイブ。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクタで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプト SMT アーカイブと Debezium コネクタで使用する関連言語の依存関係。SMT アーカイブおよび言語の依存関係は任意のコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場
合にのみ、これらのコンポーネントを追加します。
- Oracle JDBC ドライバー。Oracle データベースに接続するために必要ですが、コネク
タアーカイブには含まれていません。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
  type: imagestream ❹
  image: debezium-streams-connect:latest
  plugins: ❺
  - name: debezium-connector-oracle
    artifacts:
      - type: zip ❻
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-oracle/2.1.4.Final-redhat-00001/debezium-connector-oracle-2.1.4.Final-redhat-00001-plugin.zip ❼
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3.0.Final-redhat-❸/apicurio-registry-distro-connect-converter-2.3.0.Final-redhat-❸.zip ❽
      - type: zip
        url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip ❾
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar ❿
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
      - type: jar
        url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json/3.0.11/groovy-json-3.0.11.jar
```



```

- type: jar
  url:
https://repo1.maven.org/maven2/com/oracle/database/jdbc/ojdbc8/21.1.0.0/ojdbc8-
21.1.0.0.jar

bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

...

```

表6.12 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクタを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクタのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタプラグインに対して、コネクタと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクタアーカイブは、 .zip ファイル形式で提供されます。JDBC ドライバファイルは .jar 形式です。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。
7	artifacts.url の値は、コネクタアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。Debezium コネクタアーティファクトは Red Hat リポジトリで入手できます。OpenShift クラスターは指定されたサーバーにアクセスする必要があります。

項目	説明
8	(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクターが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。
9	(オプション) Debezium コネクターで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベースルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 40px; height: 40px; background-color: black; margin-right: 10px;"></div> <div> <p>重要</p> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタープラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクターが失敗します。</p> </div> </div> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>
11	Maven Central における Oracle JDBC ドライバーの場所を指定します。必要なドライバーは Debezium Oracle コネクターアーカイブには含まれていません。

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスタを起動します。設定に一覧表示されているコネクターアーティファクトはクラスタで利用できます。

4. **KafkaConnector** リソースを作成し、デプロイする各コネクターのインスタンスを定義します。
たとえば、以下の **KafkaConnector** CR を作成し、**oracle-inventory-connector.yaml** として保存します。

例6.2 Debezium コネクターの KafkaConnector カスタムリソースを定義する oracle-inventory-connector.yaml ファイル

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-oracle ①
spec:
  class: io.debezium.connector.oracle.OracleConnector ②
  tasksMax: 1 ③
  config: ④
    schema.history.internal.kafka.bootstrap.servers: debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092
    schema.history.internal.kafka.topic: schema-changes.inventory
    database.hostname: oracle.debezium-oracle.svc.cluster.local ⑤
    database.port: 1521 ⑥
    database.user: debezium ⑦
    database.password: dbz ⑧
    database.dbname: mydatabase ⑨
    topic.prefix: inventory-connector-oracle ⑩
    table.include.list: PUBLIC.INVENTORY ⑪
  ...

```

表6.13 コネクタ設定の説明

項目	説明
1	Kafka Connect クラスタに登録するコネクタの名前。
2	コネクタクラスの名前。
3	同時に動作できるタスクの数。
4	コネクタの設定。
5	ホストデータベースインスタンスのアドレス。
6	データベースインスタンスのポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名前。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。

項目	説明
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターのトピック接頭辞。 指定する名前は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクタから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名前はクラスターのコネクタ間で一意である必要があります。 コネクタを Avro コネクタ と統合する場合、この名前空間は関連する Kafka Connect スキーマの名前や、対応する Avro スキーマの名前空間でも使用されます。
11	コネクタが変更イベントをキャプチャーするテーブルの一覧。

5. 以下のコマンドを実行してコネクタリソースを作成します。

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium Oracle デプロイメントを検証する](#) 準備が整いました。

6.5.4. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium Oracle コネクタのデプロイ

Debezium Oracle コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、以下のカスタムリソース (CR) を作成する必要があります。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium Oracle コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用するのと同じ OpenShift インスタンスに適用します。

前提条件

- Oracle Database が稼働し、[Oracle を設定して Debezium コネクタと連携する](#) 手順が完了済みである必要があります。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。

- Podman または Docker がインストールされている。
- Debezium コネクターを実行するコンテナを追加する予定のコンテナレジストリー (quay.io や docker.io など) でコンテナを作成および管理するアカウントとパーミッションを持っている。
- Kafka Connect サーバーは、Oracle 用の必要な JDBC ドライバーをダウンロードするために、Maven Central にアクセスすることができます。また、ドライバーのローカルコピー、またはローカルの Maven リポジトリーや他の HTTP サーバーから利用可能なものを使用することもできます。
詳細は、[Obtaining the Oracle JDBC driver](#) を参照してください。

手順

1. Kafka Connect の Debezium Oracle コンテナを作成します。
 - a. registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12 をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-oracle.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium 2
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
oracle/2.1.4.Final-redhat-00001/debezium-connector-oracle-2.1.4.Final-redhat-00001-
plugin.zip \
&& unzip debezium-connector-oracle-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-oracle-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://repo1.maven.org/maven2/com/oracle/ojdbc/ojdbc8/21.1.0.0/ojdbc8-
21.1.0.0.jar
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-oracle.yaml** という名前の Docker ファイルを作成します。

- b. 前のステップで作成した **debezium-container-for-oracle.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

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

```
docker build -t debezium-container-for-oracle:latest .
```

上記のコマンドは、**debezium-container-for-oracle** という名前のコンテナイメージを構築します。

- c. カスタムイメージを quay.io などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

```
podman push <myregistry.io>/debezium-container-for-oracle:latest
```

```
docker push <myregistry.io>/debezium-container-for-oracle:latest
```

- d. 新しい Debezium Oracle KafkaConnect カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  image: debezium-container-for-oracle ❷
...
```

項目	説明
1	KafkaConnector リソースはこの Kafka Connect クラスタでコネクタを設定するために使用されることを、 metadata.annotations は Cluster Operator に示します。
2	spec.image は Debezium コネクタを実行するために作成したイメージの名前を指定します。設定された場合、このプロパティによって Cluster Operator の STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE 変数がオーバーライドされます。

- e. 以下のコマンドを入力して、**KafkaConnect** CR を OpenShift Kafka Connect 環境に適用します。

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

このコマンドは、Debezium コネクタを実行するために作成したイメージの名前を指定する Kafka Connect インスタンスを追加します。

2. Debezium Oracle コネクタインスタンスを設定する **KafkaConnector** カスタムリソースを作成します。

通常、コネクタに使用できる設定プロパティを使用して、**.yaml** ファイルに Debezium Db2 コネクタを設定します。コネクタ設定は、Debezium に対して、スキーマおよびテーブルのサブセットにイベントを生成するよう指示する可能性があり、または機密性の高い、大きすぎる、または不必要な指定の列で Debezium が値を無視、マスク、または切り捨てするようにプロパティを設定する可能性もあります。

以下の例では、ポート **1521** で Oracle ホスト IP アドレスに接続する Debezium コネクタを設定します。このホストには **ORCLCDB** という名前のデータベースがあり、**server1** はサーバーの論理名です。

Oracle inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector-oracle 1
  labels:
    strimzi.io/cluster: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: 'true'
spec:
  class: io.debezium.connector.oracle.OracleConnector 2
  config:
    database.hostname: <oracle_ip_address> 3
    database.port: 1521 4
    database.user: c##dbzuser 5
    database.password: dbz 6
    database.dbname: ORCLCDB 7
    database.pdb.name : ORCLPDB1, 8
    topic.prefix: inventory-connector-oracle 9
    schema.history.internal.kafka.bootstrap.servers: kafka:9092 10
    schema.history.internal.kafka.topic: schema-changes.inventory 11
```

表6.14 コネクタ設定の説明

項目	説明
1	Kafka Connect サービスに登録する場合のコネクタの名前。
2	この Oracle コネクタクラスの名称。
3	Oracle インスタンスのアドレス。
4	Oracle インスタンスのポート番号。
5	コネクタのユーザーの作成 で指定した Oracle ユーザーの名称。
6	コネクタのユーザーの作成 で指定した Oracle 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称。

項目	説明
8	コネクターが変更をキャプチャーする Oracle のプラグ可能なデータベースの名前。コンテナデータベース (CDB) のインストールのみで使用されます。
9	topic prefix は、コネクターが変更をキャプチャーする Oracle データベースサーバーの名前空間を識別して提供します。
10	DDL ステートメントをデータベーススキーマの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スキーマの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。

- Kafka Connect でコネクターインスタンスを作成します。たとえば、**KafkaConnector** リソースを **inventory-connector.yaml** ファイルに保存した場合は、以下のコマンドを実行します。

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

上記のコマンドは **inventory-connector** を登録し、コネクターは **KafkaConnector** CR に定義されている **server1** データベースに対して実行を開始します。

Debezium Oracle コネクターに設定できる設定プロパティの完全リストは [Oracle コネクタープロパティ](#) を参照してください。

結果

コネクターが起動すると、コネクターが設定された Oracle データベースの **整合性スナップショット** が実行されます。その後、コネクターは行レベルの操作のデータ変更イベントの生成を開始し、変更イベントレコードを Kafka トピックにストリーミングします。

6.5.5. コンテナデータベースとノンコンテナデータベースの設定

Oracle Database は以下の展開タイプをサポートしています。

コンテナデータベース (CDB)

複数の PDB (Multi Pluggable Database) を格納できるデータベース。データベースクライアントは、標準的な非 CDB データベースであるかのように、各 PDB に接続します。

ノンコンテナデータベース (非 CDB)

プラグブルデータベースの作成には対応していない、標準的な Oracle のデータベース。

6.5.6. Debezium Oracle コネクターが実行していることの確認

コネクターがエラーなしで正常に起動すると、コネクターがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウンストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクターが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクターのステータスを確認します。

- コネクタがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクタが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクタが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクタの名前をクリックします (例: **inventory-connector-oracle**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-oracle -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例6.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-oracle
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector
```

...

```
Status:
Conditions:
  Last Transition Time: 2021-12-08T17:41:34.897153Z
```

```

Status:          True
Type:            Ready
Connector Status:
Connector:
  State:         RUNNING
  worker_id:    10.131.1.124:8083
Name:           inventory-connector-oracle
Tasks:
  Id:           0
  State:        RUNNING
  worker_id:    10.131.1.124:8083
  Type:         source
Observed Generation: 1
Tasks Max:      1
Topics:
  inventory-connector-oracle.inventory
  inventory-connector-oracle.inventory.addresses
  inventory-connector-oracle.inventory.customers
  inventory-connector-oracle.inventory.geom
  inventory-connector-oracle.inventory.orders
  inventory-connector-oracle.inventory.products
  inventory-connector-oracle.inventory.products_on_hand
Events: <none>

```

2. コネクタによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. **KafkaTopics** リストから確認するトピックの名前をクリックします (例: **inventory-connector-oracle.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例6.4 KafkaTopic リソースのステータス

NAME	REPLICATION FACTOR	READY	CLUSTER
connect-cluster-configs	1	True	debezium-kafka-cluster
connect-cluster-offsets			debezium-kafka-cluster

```

1          True
connect-cluster-status                          debezium-kafka-cluster 5
1          True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fad1c6a
debezium-kafka-cluster 50          1          True
inventory-connector-oracle--a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster 1          1          True
inventory-connector-oracle.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480      debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b      debezium-kafka-cluster 1
1          True
inventory-connector-oracle.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5      debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d      debezium-kafka-cluster
1          1          True
inventory-connector-oracle.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef      debezium-kafka-cluster 1
1          True
inventory-connector-oracle.inventory.products_on_hand---
8649e0f17ffcc9212e266e31a7aeaa4585e5c6b5     debezium-kafka-cluster 1
1          True
schema-changes.inventory                       debezium-kafka-cluster
1          1          True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55     debezium-
kafka-cluster 1          1          True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b     debezium-kafka-cluster 1 1
True

```

3. トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。

```

oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>

```

以下に例を示します。

```

oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-oracle.inventory.products_on_hand

```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory-connector-oracle.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例6.5 Debezium 変更イベントの内容

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "product_id",
        "optional": false,
        "name": "inventory-connector-oracle.inventory.products_on_hand.Key",
        "payload": {
          "product_id": 101
        }
      }
    ]
  },
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "product_id",
            "optional": true,
            "name": "inventory-connector-oracle.inventory.products_on_hand.Value",
            "field": "before"
          },
          {
            "type": "struct",
            "fields": [
              {
                "type": "int32",
                "optional": false,
                "field": "product_id",
                "optional": true,
                "name": "inventory-connector-oracle.inventory.products_on_hand.Value",
                "field": "after"
              }
            ]
          },
          {
            "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": "string",
            "optional": true,
            "name": "io.debezium.data.Enum",
            "version": 1,
            "parameters": {
              "allowed": "true,last,false",
              "default": "false",
              "field": "snapshot"
            }
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "sequence"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          },
          {
            "type": "string",
            "optional": true,
            "field": "gtid"
          },
          {
            "type": "string",
            "optional": false,
            "field": "file"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "pos"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "row"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "query"
          },
          {
            "optional": false,
            "name": "io.debezium.connector.oracle.Source",
            "field": "source",
            "type": "string",
            "optional": false,
            "field": "op"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
          },
          {
            "type": "struct",
            "fields": [
              {
                "type": "string",
                "optional": false,
                "field": "id"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "total_order"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "data_collection_order"
              },
              {
                "optional": true,
                "field": "transaction"
              },
              {
                "optional": false,
                "name": "inventory-connector-oracle.inventory.products_on_hand.Envelope",
                "payload": {
                  "before": null,
                  "after": {
                    "product_id": 101,
                    "quantity": 3,
                    "source": {
                      "version": "2.1.4.Final-redhat-00001",
                      "connector": "oracle",
                      "name": "inventory-connector-oracle",
                      "ts_ms": 1638985247805,
                      "snapshot": "true",
                      "db": "inventory",
                      "sequence": null,
                      "table": "products_on_hand",
                      "server_id": 0,
                      "gtid": null,
                      "file": "oracle-bin.000003",
                      "pos": 156,
                      "row": 0,
                      "thread": null,
                      "query": null,
                      "op": "r",
                      "ts_ms": 1638985247805,
                      "transaction": null
                    }
                  }
                }
              }
            ]
          }
        ]
      }
    ]
  }
}
```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

6.6. DEBEZIUM ORACLE コネクタースettingsプロパティの説明

Debezium Oracle コネクタースettingsには、アプリケーションに適したコネクタースettingsを実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。プロパティに関する情報は、以下のように設定されています。

- [必要な Debezium Oracle コネクタースettingsプロパティ](#)

- Debezium がデータベース履歴トピックから読み取るイベントを処理する方法を制御する [データベース履歴コネクタ設定プロパティ](#)。
 - [パススルーデータベーススキーマ履歴プロパティ](#)
- データベースドライバーの動作を制御する [パススルーデータベースドライバープロパティ](#)。

必要な Debezium Oracle コネクタ設定プロパティ

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

プロパティ	デフォルト	説明
name	デフォルトなし	コネクタの一意名。同じ名前で再登録を試みると失敗します。(このプロパティはすべての Kafka Connect コネクタに必要です)
connector.class	デフォルトなし	コネクタの Java クラスの名前。Oracle コネクタには、常に io.debezium.connector.oracle.OracleConnector の値を使用します。

converters	デフォルトなし	<p>コネクタが使用できる カスタムコンバーター インスタンスのシンボリック名のコンマ区切りリストを列挙します。</p> <p>たとえば、boolean です。</p> <p>このプロパティは、コネクタがカスタムコンバータを使用できるようにするために必要です。</p> <p>コネクタに設定するコンバータごとに、コンバータインターフェイスを実装するクラスの完全修飾名を指定する .type プロパティも追加する必要があります。.type プロパティでは、以下の形式を使用します。</p> <p><converterSymbolicName>.type</p> <p>以下に例を示します。</p> <pre>boolean.type: io.debezium.connector.oracle.converters. NumberOneToBooleanConverter</pre> <p>設定されたコンバータの動作をさらに制御したい場合は、1つ以上の設定パラメータを追加して、コンバータに値を渡すことができます。追加の設定パラメータとコンバータを関連付けるには、パラメータ名の前にコンバータのシンボリック名を付けます。</p> <p>たとえば、boolean コンバータが処理する列のサブセットを指定する selector パラメータを定義するには、次のプロパティを追加します。</p> <pre>boolean.selector: .*MYTABLE.FLAG,.*.IS_ARCHIVED</pre>
tasks.max	1	<p>このコネクタに作成するタスクの最大数。Oracle コネクタ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。</p>
database.hostname	デフォルトなし	<p>Oracle データベースサーバーの IP アドレスまたはホスト名。</p>
database.port	デフォルトなし	<p>Oracle データベースサーバーの整数のポート番号。</p>
database.user	デフォルトなし	<p>コネクタが Oracle データベースサーバーへの接続に使用する Oracle ユーザーアカウントの名前。</p>

database.password	デフォルトなし	Oracle 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	デフォルトなし	接続先のデータベースの名前。CDB + PDB モデルを使用する場合は、CDB 名である必要があります。
database.url	デフォルトなし	raw データベースの JDBC URL を指定します。このプロパティを使用すると、そのデータベース接続を柔軟に定義できます。有効な値は、raw TNS 名および RAC 接続文字列などです。
database.pdb.name	デフォルトなし	接続先の Oracle のプラグ可能なデータベースの名前。このプロパティは、コンテナデータベース (CDB) のインストールでのみ使用してください。
topic.prefix	デフォルトなし	<p>コネクターが変更をキャプチャーする Oracle データベースサーバーの名前空間を識別して提供するトピック接頭辞。設定した値は、コネクターが出力するすべての Kafka トピック名の接頭辞として使用されます。Debezium 環境のすべてのコネクターで一意のトピック接頭辞を指定します。英数字、ハイフン、ドットおよびアンダースコアの文字が有効です。</p> <div data-bbox="884 1200 1426 1794" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <p>警告</p>  <p>このプロパティの値を変更しないでください。名前の変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。また、コネクターはデータベーススキーマ履歴トピックを復元できません。</p> </div>

<code>database.connection.adapter</code>	<code>logminer</code>	コネクタがデータベースの変更をストリーミングする際に使用するアダプター実装。 logminer (デフォルト) を設定することができます。コネクタは、ネイティブの Oracle LogMiner API を使用します。
--	-----------------------	--

<p>snapshot.mode</p>	<p>Initial</p>	<p>このコネクターがキャプチャーされたテーブルのスナップショットを取得するために使用するモードを指定します。以下の値を設定できます。</p> <p>always</p> <p>スナップショットには、キャプチャーされたテーブルの構造およびデータが含まれます。表現この値を指定して、各コネクターの開始時にキャプチャーされたテーブルからデータの表現をすべてトピックに入力します。</p> <p>Initial</p> <p>スナップショットには、キャプチャーされたテーブルの構造およびデータが含まれます。この値を指定して、キャプチャーされたテーブルからのデータの完全な表現を使用して、トピックを設定します。スナップショットが正常に完了した場合、次のコネクター開始時にスナップショットは再度実行されません。</p> <p>initial_only</p> <p>スナップショットには、キャプチャーされたテーブルの構造およびデータが含まれます。コネクターは最初のスナップショットを実行し、その後の変更を処理せずに停止します。</p> <p>schema_only</p> <p>スナップショットには、キャプチャーされたテーブルの構造のみが含まれます。コネクターに、スナップショット作成後に発生した変更のみのデータをキャプチャーさせる場合には、この値を指定します。</p> <p>schema_only_recovery</p> <p>これは、すでに変更を取り込んでしまったコネクターのリカバリー設定です。この設定により、コネクターを再起動すると、破損または損失したデータベーススキーマ履歴トピックのリカバリーが可能になります。これを定期的に設定して、予想外に増加しているデータベーススキーマ履歴トピックをクリーンアップすることができます。データベーススキーマ履歴トピックは無期限に保持する必要があります。このモードは、コネクターがシャットダウンされた時点とスナップショットが作成された時点からスキーマの変更が行われていないことが保証されていない場合にのみ安全です。</p> <p>スナップショットが完了すると、snapshot.mode が initial_only に設定されている場合を除き、コネクターはデータベースの REDO ログから変更イベントを読み続けます。</p> <p>詳しくは、snapshot.mode オプションの表をご覧ください。</p>
-----------------------------	----------------	---

snapshot.locking.mode	shared	<p>コネクタがテーブルロックを保持するかどうか、また保持する時間をコントロールします。テーブルロックは、コネクタがスナップショットを実行している間、特定の種類の変更テーブル操作が発生するのを防ぎます。以下の値を設定できます。</p> <p>shared</p> <p>テーブルへの同時アクセスを可能にしますが、どのセッションも排他的なテーブルロックを取得できないようにします。コネクタは、テーブルスキーマをキャプチャする際に ROW SHARE レベルのロックを取得します。</p> <p>none</p> <p>スナップショット中にコネクタがテーブルロックを取得するのを防ぎます。この設定は、スナップショットの作成中にスキーマの変更が発生しない場合にのみ使用します。</p>
snapshot.include.collection.list	table.include.list に指定したすべてのテーブル	<p>スナップショットに含めるテーブルの完全修飾名 (<schemaName>.<tableName>) と一致する正規表現のコンマ区切りリスト (任意)。POSIX 正規表現のみが有効です。指定する項目は、コネクタの table.include.list プロパティで名前を付ける必要があります。このプロパティは、コネクタの snapshot.mode プロパティが never 以外の値に設定されている場合にのみ有効です。このプロパティは増分スナップショットの動作には影響しません。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p>

<p>snapshot.select.statement.overrides</p>	<p>デフォルトなし</p>	<p>スナップショットに追加するテーブル行を指定します。スナップショットにテーブルの行のサブセットのみを含める場合は、プロパティを使用します。このプロパティはスナップショットにのみ影響します。コネクターがログから読み取るイベントには影響しません。</p> <p>プロパティには、<schemaName>、<tableName> の形式で完全修飾テーブル名のコンマ区切りリストが含まれます。たとえば、</p> <p>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</p> <p>をリスト内の各テーブルに対して、スナップショットを作成する場合には、その他の設定プロパティを追加して、コネクターがテーブルで実行するように SELECT ステートメントを指定します。指定した SELECT ステートメントは、スナップショットに追加するテーブル行のサブセットを決定します。この SELECT 文のプロパティの名前を指定するには、</p> <p>snapshot.select.statement.overrides.<schemaName>.<tableName> の形式を使用します</p> <p>(例: snapshot.select.statement.overrides.customers.orders)</p> <p>スナップショットにソフト削除以外のレコードのみを含める場合は、soft-delete 列 (delete_flag) を含む customers.orders テーブルから、以下のプロパティを追加します。</p> <pre>"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre> <p>作成されるスナップショットでは、コネクターには delete_flag = 0 のレコードのみが含まれます。</p>
---	----------------	---

schema.include.list	デフォルトなし	<p>変更をキャプチャーする対象とするスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。POSIX 正規表現のみが有効です。 schema.include.list に含まれていないスキーマ名は、変更をキャプチャーする対象から除外されます。デフォルトでは、システム以外のスキーマはすべて変更がキャプチャーされます。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、 schema.exclude.list プロパティも設定しないでください。</p>
include.schema.comments	false	<p>コネクターがメタデータオブジェクトでテーブルおよび列のコメントを解析して公開するかどうかを指定するブール値。このオプションを有効にすると、メモリー使用量に影響を及ぼします。論理スキーマオブジェクトの数およびサイズは、Debezium コネクターによって消費されるメモリーの量に大きく影響し、それぞれに大きな文字列データを追加すると、非常に高価になる可能性があります。</p>
schema.exclude.list	デフォルトなし	<p>変更をキャプチャーする対象としないスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。POSIX 正規表現のみが有効です。システムスキーマ以外で、 schema.exclude.list に名前が含まれていないスキーマの変更がキャプチャーされます。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、 `schema.include.list` プロパティを設定しないでください。</p>

table.include.list	デフォルトなし	<p>キャプチャー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)。POSIX 正規表現のみが有効です。このプロパティーが設定されている場合、コネクターは指定されたテーブルからのみ変更をキャプチャします。各テーブルの識別子は、</p> <p><schema_name>.<table_name></p> <p>の形式を使用します。デフォルトでは、コネクターはキャプチャーされたデータベースごとにすべての非システムテーブルを監視します。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティーを設定に含める場合は、table.exclude.list プロパティーも設定しないでください。</p>
table.exclude.list	デフォルトなし	<p>監視から除外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)。POSIX 正規表現のみが有効です。コネクターは除外リストに指定されていないテーブルからの変更をキャプチャーします。</p> <p><schemaName>.<tableName> 形式を使用して、各テーブルの識別子を指定します。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティーを設定に含める場合は、table.include.list プロパティーも設定しないでください。</p>

<p>column.include.list</p>	<p>デフォルトなし</p>	<p>変更イベントメッセージの値に含ま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)。POSIX 正規表現のみが有効です。列の完全修飾名は、</p> <p><Schema_name>.<table_name>.<column_name></p> <p>の形式を使用します。プライマリーキーの列は、このプロパティを使用してその値を明示的に含めなくても、常にイベントのキーに含まれます。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。このプロパティを設定に含める場合は、column.exclude.list プロパティを設定しないでください。</p>
<p>column.exclude.list</p>	<p>デフォルトなし</p>	<p>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)。POSIX 正規表現のみが有効です。完全修飾の列名は、</p> <p><schema_name>.<table_name>.<column_name></p> <p>の形式を使用します。プライマリーキーの列は、このプロパティを使用してその値を明示的に除外した場合でも、イベントのキーには常に含まれます。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。このプロパティを設定に含める場合は、column.include.list プロパティを設定しないでください。</p>

<p><code>column.mask.hash.hashAlgorithm.with.salt.salt;</code> <code>column.mask.hash.v2.hashAlgorithm.with.salt.salt</code></p>	<p>該当なし</p>	<p>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)。列の完全修飾名の形式は <code><schemaName>.<tableName>.<columnName></code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>作成された変更イベントレコードでは、指定された列の値は仮名に置き換えられます。</p> <p>仮名は、指定された <code>hashAlgorithm</code> と <code>salt</code> を適用すると得られるハッシュ化された値で設定されます。使用されるハッシュ関数に基づいて、参照整合性は保持され、列値は仮名に置き換えられます。サポートされるハッシュ関数は、Java Cryptography Architecture Standard Algorithm Name Documentation の MessageDigest セクション に説明されています。</p> <p>以下の例では、CzQMA0cB5K が無作為に選択された salt になります。</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>必要な場合は、仮名は自動的に列の長さに短縮されます。コネクター設定には、異なるハッシュアルゴリズムと salt を指定する複数のプロパティーを含めることができます。</p> <p>使用される <code>hashAlgorithm</code>、選択された <code>salt</code>、および実際のデータセットによっては、結果のデータセットが完全にマスクされない場合があります。</p> <p>値が異なる場所やシステムでハッシュ化されている場合は、ハッシュ化ストラテジーバージョン 2 を使用する必要があります。</p>
---	-------------	--

binary.handling.mode	bytes	バイナリー (blob) 列を変更イベントで表す方法を指定します。 bytes はバイナリーデータをバイト配列として表します (デフォルト)。 base64 はバイナリーデータを base64 でエンコードされた文字列として表します。 base64-url-safe は base64-url-safe でエンコードされた文字列としてバイナリーデータを表現します。 hex はバイナリーデータを 16 進エンコード (base16) 文字列として表します。
schema.name.adjustment.mode	none	コネクタで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能: <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。
decimal.handling.mode	precise	コネクタが NUMBER 、 DECIMAL および NUMERIC 列の浮動小数点値を処理する方法を指定します。以下のオプションのいずれかを使用できます。 <p>precise (デフォルト)</p> <p>バイナリー形式の変更イベントで表現される java.math.BigDecimal の値を使用して正確に値を表します。</p> <p>double</p> <p>double 値を使用して値を表します。double 値を使用することは簡単ですが、精度が失われる可能性があります。</p> <p>string</p> <p>フォーマットされた文字列としてエンコードされます。string オプションを使用すると、消費は簡単になりますが、実際のタイプのセマンティクスの情報が失われる可能性があります。詳細は、数字型 を参照してください。</p>
interval.handling.mode	numeric	numeric は、マイクロ秒単位の概算値で 間隔 を表します。 <p>string</p> <p>は、P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S の文字列パターン表現を使用して間隔を正確に表します。例: P1Y2M3DT4H5M6.78S。</p>

event.processing.failure.handling.mode	fail	<p>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。以下のオプションのいずれかを使用できます。</p> <p>fail</p> <p>例外 (問題のあるイベントのオフセットを示す) を伝播することでコネクターが停止します。</p> <p>warn</p> <p>問題のあるイベントがスキップされるようにします。その後、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。</p> <p>skip</p> <p>問題のあるイベントがスキップされるようにします。</p>
max.batch.size	2048	このコネクターの反復処理中に処理す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。
max.queue.size	8192	<p>ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクターが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクターがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。max.queue.size の値を、max.batch.size の値よりも大きくなるように設定します。</p>
max.queue.size.in.bytes	0 (無効)	<p>ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。</p> <p>max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、max.queue.size=1000、max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。</p>

<code>poll.interval.ms</code>	500 (0.5 秒)	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が待機する時間 (ミリ秒単位) を指定する正の整数値。
<code>tombstones.on.delete</code>	true	<p>削除 イベントの後に廃棄 (tombstone) イベントを行うかどうかを制御します。以下の値が可能です。</p> <p>true</p> <p>削除操作ごとに、コネクタは、削除 イベントと後続の廃棄 (tombstone) イベントを出力します。</p> <p>false</p> <p>削除操作ごとに、コネクタは 削除 イベントのみを出力します。</p> <p>ソースレコードを削除すると、廃棄イベント (デフォルトの動作) により、Kafka が <code>log compaction</code> が有効なトピックで削除した列のキーを共有するイベントをすべて完全に削除できるようになります。</p>

<p>message.key.columns</p>	<p>デフォルトなし</p>	<p>指定のテーブルの Kafka トピックに公開する変更イベントレコードのカスタムメッセージキーを形成するためにコネクタが使用する列を指定する式のリスト。</p> <p>デフォルトでは、Debezium はテーブルのプライマリーキー列を、出力するレコードのメッセージキーとして使用します。デフォルトの代わりに、またはプライマリーキーのないテーブルのキーを指定するには、1つ以上の列をもとにカスタムメッセージキーを設定できます。テーブルにカスタムメッセージキーを設定するには、テーブルを列挙した後、メッセージキーとして使用する列を列挙します。各リストエントリは、</p> <p><fullyQualifiedTableName>:<keyColumn>,<keyColumn></p> <p>の形式を使用します。テーブルのキーを複数の列名に基づいて設定するには、列名の間にはコンマを挿入します。</p> <p>各完全修飾テーブル名は、</p> <p><schemaName>.<tableName> の形式の正規表現です。</p> <p>プロパティには複数のテーブルのエントリを含めることができます。セミコロンを使用して、リスト内のテーブルエントリを区切ります。</p> <p>以下の例は、テーブル inventory.customers および purchase.orders:</p> <p>inventory.customers:pk1,pk2; (.*)purchaseorders:pk3,pk4</p> <p>のメッセージキーを設定します。テーブル inventory.customer の場合、列 pk1 と pk2 がメッセージキーとして指定されます。どのスキーマの purchaseorders テーブルでも、pk3 と pk4 のカラムがメッセージキーとして使用されます。</p> <p>カスタムメッセージキーの作成に使用する列の数に制限はありません。ただし、一意の鍵を指定するために必要な最小数を使用することが推奨されます。</p>
-----------------------------------	----------------	--

<p>column.truncate.to.length.chars</p>	<p>デフォルトなし</p>	<p>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)。一連の列の値をコネクタでマスクする場合 (たとえば、列に機密データが含まれている場合) は、このプロパティを設定します。 length を正の整数に設定して、指定された列のデータをプロパティ名の長さで指定されたアスタリスク (*) 文字数で置き換えます。指定した列のデータを空の文字列に置き換えるには、長さを 0 (ゼロ) に設定します。</p> <p>列の完全修飾名は、 <schemaName>.<tableName>.<columnName> の形式に従います。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>
<p>column.mask.with.length.chars</p>	<p>デフォルトなし</p>	<p>文字をアスタリスク (*) に置き換えることで、変更イベントメッセージの列名をマスクする正規表現のコンマ区切りリスト (任意)。プロパティ名に置き換える文字の数を指定します (例: column.mask.with.8.chars)。長さは正の整数またはゼロに指定します。次に、マスクを適用する各文字ベースの列名のリストに正規表現を追加します。</p> <p><schemaName>.<tableName>.<columnName> の形式を使用して、列の完全修飾名を指定します。</p> <p>コネクタ設定には、異なる長さを指定する複数のプロパティを含めることができます。</p>

<p>column.propagate.source.type</p>	<p>デフォルトなし</p>	<p>列のメタデータを表す追加パラメーターをコネクターに発行させたい列の完全修飾名に一致する、オプションのコンマ区切りの正規表現のリスト。このプロパティが設定されている場合、コネクターは次のフィールドをイベントレコードのスキーマに追加します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクターがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は <tableName>.<columnName>、または <schemaName>.<tableName>.<columnName> のいずれかの形式を取ります。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現 として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p>
--	----------------	--

<p>datatype.propagate.source.type</p>	<p>デフォルトなし</p>	<p>データベース内の列に対して定義されているデータ型の完全修飾名を指定する正規表現のオプションのコンマ区切りリスト。このプロパティが設定されている場合、データ型が一致する列に対して、コネクタはスキーマに次の追加フィールドを含むイベントレコードを発行します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクタがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、<tableName>.<typeName>、または <schemaName>.<tableName>.<typeName> のいずれかの形式を取ります。データ型の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データ型の名前文字列全体に対して照合されます。式は、型名に存在する可能性のある部分文字列と一致しません。</p> <p>Oracle 固有のデータ型名の一覧は、Oracle データ型マッピング を参照してください。</p>
--	----------------	--

heartbeat.interval.ms	0	<p>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間隔(ミリ秒単位)を指定します。</p> <p>このプロパティを使用して、コネクターがソースデータベースから変更イベントを受信し続けるかどうかを決定します。</p> <p>長期間にわたり、キャプチャーしたテーブルで変更イベントが発生しない場合に、このプロパティを設定すると便利です。</p> <p>このような場合には、コネクターは redo ログの読み取りを続行しますが、変更イベントメッセージは出力されないため、Kafka トピックのオフセットは変更されません。コネクターはデータベースから読み取る最新のシステム変更番号 (SCN) をフラッシュしないため、データベースは必要以上のログファイルを保持する可能性があります。コネクターが再起動すると、保持期間が延長され、コネクターは一部の変更イベントを重複して送信する可能性があります。</p> <p>デフォルト値 0 が設定されていると、コネクターでハートビートメッセージが送信されません。</p>
heartbeat.action.query	デフォルトなし	<p>コネクターがハートビートメッセージを送信するときにコネクターがソースデータベースで実行するクエリーを指定します。</p> <p>たとえば、</p> <pre>INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')</pre> <p>です。コネクターは ハートビートメッセージ を出力した後にクエリーを実行します。</p> <p>このプロパティを設定し、ハートビートテーブルを作成してハートビートメッセージを受信することで、Debezium が高トラフィックデータベースと同じホスト上にある低トラフィックデータベースのオフセットの同期に失敗 する状況を解決することができます。コネクターは設定されたテーブルにレコードを挿入した後、低トラフィックデータベースから変更を受信し、データベースの SCN 変更を認識することができるので、オフセットをブローカーと同期させることができます。</p>

snapshot.delay.ms	デフォルトなし	スナップショットを作成する前に、コネクタが起動してから待機する間隔をミリ秒単位で指定します。 このプロパティを使用して、クラスターで複数の接続を開始するときに (コネクタのリバランスの原因となる可能性がある) スナップショットが中断されないようにします。
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。コネクタは、指定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。
query.fetch.size	デフォルトなし	指定のクエリーのデータベースのラウンドトリップごとにフェッチされる行の数を指定します。デフォルトは、JDBC ドライバーのデフォルトのフェッチサイズです。
sanitize.field.names	コネクタ設定が、Avro を使用するように key.converter または value.converter パラメータ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を正規化するかどうかを指定します。詳しい情報は、 Avro naming を参照してください。
provide.transaction.meta data	false	true に設定すると、Debezium はトランザクション境界でイベントを生成し、トランザクションメタデータでデータイベントエンベロープをエンリッチします。 詳細は、 トランザクションメタデータ を参照してください。

log.mining.strategy	redo_log_catalog	<p>テーブルと列 ID を名前に解決するために Oracle LogMiner が特定のデータディクショナリーをビルドおよび使用する方法を制御するマイニングストラテジーを指定します。</p> <p>redo_log_catalog:: データディクショナリーをオンラインの redo ログに書き込みます。これにより、時間の経過と共により多くのアーカイブログが生成されるようになります。これにより、キャプチャーされたテーブルに対する DDL の変更を追跡することもできます。そのため、スキーマが頻繁に変更される場合、これが理想的な変更です。</p> <p>online_catalog:: データベースの現在のデータディクショナリーを使用してオブジェクト ID を解決し、オンラインの redo ログに追加情報を書き込みません。これにより、Log Miner は大幅に速く採掘できるようになりましたが、DDL の変更を追跡できないという代償を払いました。キャプチャしたテーブルのスキーマが頻繁に変更されない、または全く変更されない場合は、この方法が最適です。</p>
log.mining.buffer.type	memory	<p>バッファータイプは、コネクターがトランザクションデータのバッファリングをどのように管理するかを制御します。</p> <p>memory- JVM プロセスのヒープを使用してすべてのトランザクションデータをバッファリングします。コネクターで長時間のトランザクションや大規模なトランザクションの処理を想定していない場合は、このオプションを選択します。このオプションを有効にすると、再起動時にバッファの状態が保持されません。リスタート後は、現在のオフセットの SCN 値からバッファを再作成します。</p>
log.mining.session.max.ms	0	<p>新しいセッションが使用される前に LogMiner セッションをアクティブに保つことができる最大期間 (ミリ秒単位)。</p> <p>低容量のシステムの場合、同じセッションを長期間使用すると、LogMiner セッションが PGA メモリーを過剰に消費することがあります。デフォルトの動作は、ログスイッチが検出されたときのみ、新しい LogMiner セッションを使用することです。この値を 0 より大きく設定することで、LogMiner セッションが PGA メモリーの割り当て解除と再割り当てのために停止および開始される前にアクティブにできる最大ミリ秒数を指定します。</p>

<code>log.mining.batch.size.min</code>	1000	このコネクタが redo/archive ログから読み込もうとする最小 SCN 間隔サイズ。また、必要に応じてコネクタのスループットを調整するために、アクティブバッチサイズをこの量だけ増減させます。
<code>log.mining.batch.size.max</code>	100000	このコネクタが REDO/ARCHIVE ログから読み取るときに使用する最大 SCN インターバルサイズです。
<code>log.mining.batch.size.default</code>	20000	コネクタが REDO/ARCHIVE ログからデータを読み取る際に使用する開始 SCN 間隔サイズ。これは、バッチサイズを調整するための手段としても機能します。現在の SCN とバッチの開始/終了 SCN の差がこの値よりも大きい場合、バッチサイズが増減されます。
<code>log.mining.sleep.time.min.ms</code>	0	redo/archive ログからデータを読み込んだ後、再びデータの読み込みを開始するまでのコネクタのスリープ時間の最小値です。値はミリ秒単位です。
<code>log.mining.sleep.time.max.ms</code>	3000	redo/archive ログからデータを読み込んだ後、再びデータの読み込みを開始するまでのコネクタのスリープ時間の最大値。値はミリ秒単位です。
<code>log.mining.sleep.time.default.ms</code>	1000	redo/archive ログからデータを読み込んだ後、再びデータの読み込みを開始するまでのコネクタのスリープ時間の開始値。値はミリ秒単位です。
<code>log.mining.sleep.time.increment.ms</code>	200	logminer からデータを読み取る際に、コネクタが最適なスリープ時間を調整するために使用する時間の最大値を上下させる。値はミリ秒単位です。
<code>log.mining.archive.log.hours</code>	0	SYSDATE からアーカイブログを採掘するまでの過去の時間数です。デフォルトの設定 (0) を使用すると、コネクタはすべてのアーカイブログを粉碎します。

<p>log.mining.archive.log.only.mode</p>	<p>false</p>	<p>コネクターが変更をアーカイブログだけから挽くのか、オンライン REDO ログとアーカイブログを組み合わせで挽くのか (デフォルト) を制御します。</p> <p>redo ログは円形のバッファを使用しており、どの時点でもアーカイブすることができます。オンライン redo ログが頻繁にアーカイブされる環境では、LogMiner のセッションが失敗することがあります。redo ログとは対照的に、アーカイブログは信頼性が保証されています。このオプションを true に設定すると、コネクターはアーカイブログのみをマイニングします。コネクターがアーカイブログのみをマイニングするように設定すると、オペレーションがコミットされてからコネクターが関連する変更イベントを発するまでの待ち時間が長くなる可能性があります。遅延の程度は、データベースがオンラインの redo ログをアーカイブするように設定されている頻度によって異なります。</p>
<p>log.mining.archive.log.only.scn.poll.interval.ms</p>	<p>10000</p>	<p>開始システムの変更番号がアーカイブログにあるかどうかを判断するためのポーリングの間に、コネクターがスリープするミリ秒数です。log.mining.archive.log.only.mode が有効でない場合は、この設定は使用されません。</p>
<p>log.mining.transaction.retention.hours</p>	<p>0</p>	<p>正の整数値で、redo ログの切り替えの間に長時間実行されるトランザクションを保持する時間を指定します。0 に設定すると、コミットまたはロールバックが検出されるまで、トランザクションが保持されます。</p> <p>LogMiner アダプターは、実行中のすべてのトランザクションのインメモリーバッファを維持します。トランザクションの一部となるすべての DML 操作は、コミットまたはロールバックが検出されるまでバッファされるため、そのバッファがオーバーフローしないように長時間実行されるトランザクションを回避する必要があります。設定されたこの値を超えるトランザクションは完全に破棄され、コネクターはトランザクションに含まれていた操作のメッセージを出力しません。</p>

log.mining.archive.destination.name	デフォルトなし	<p>LogMiner でアーカイブログをマイニングする際に使用する、設定された Oracle のアーカイブ先を指定します。</p> <p>デフォルトの動作では、ローカルで設定された最初の有効なデスティネーションが自動的に選択されます。しかし、LOG_ARCHIVE_DEST_5 のように、デスティネーション名を指定すれば、特定のデスティネーションを使用することができます。</p>
log.mining.username.exclude.list	デフォルトなし	<p>LogMiner クエリーから除外するデータベースユーザーのリスト。特定のユーザーが行った変更を常にキャプチャプロセスから除外したい場合は、このプロパティを設定すると便利です。</p>
log.mining.scn.gap.detection.gap.size.min	1000000	<p>SCN ギャップがあるかどうかを判断するために、コネクタが現在の SCN 値と前回の SCN 値の差と比較する値を指定します。SCN 値の差が指定された値より大きく、時間差が log.mining.scn.gap.detection.time.interval.max.ms より小さい場合、SCN ギャップが検出され、コネクタは設定された最大バッチよりも大きいマイニングウィンドウ。</p>
log.mining.scn.gap.detection.time.interval.max.ms	20000	<p>SCN ギャップがあるかどうかを判断するために、コネクタが現在の SCN タイムスタンプと前回の SCN タイムスタンプの差と比較する値をミリ秒単位で指定します。タイムスタンプの差が指定された値よりも小さく、SCN デルタが指定された値よりも大きい場合、SCN ギャップが検出され、設定された最大バッチよりも大きいマイニングウィンドウを使用します。 log.mining.scn.gap.detection.gap.size.min より大きい場合、SCN ギャップが検出され、コネクタは設定された最大バッチよりも大きいマイニングウィンドウを使用します。</p>

<p>lob.enabled</p>	<p>false</p>	<p>ラージオブジェクト (CLOB または BLOB) の列値を変更イベントで出力するかどうかを制御します。</p> <p>デフォルトでは、変更イベントには大きなオブジェクト列がありますが、列には値が含まれていません。大規模なオブジェクトの列タイプやペイロードの処理管理には、ある程度のオーバーヘッドがあります。大きなオブジェクトの値をキャプチャして、変更イベントでシリアル化するには、このオプションを true に設定します。</p> <div data-bbox="882 611 991 775" style="display: inline-block; vertical-align: middle;">  </div> <p style="margin-left: 20px;">注記</p> <p style="margin-left: 20px;">ラージオブジェクトデータタイプの使用は、技術プレビューの機能です。</p>
<p>unavailable.value.placeholder</p>	<p>__debezium_unavailable_value</p>	<p>コネクターが提供する定数を指定して、元の値がデータベースによって提供されておらず、また変更されていない値であることを示します。</p>

<p>rac.nodes</p>	<p>デフォルトなし</p>	<p>Oracle Real Application Clusters (RAC) ノードのホスト名またはアドレスをコマンドで区切って入力してください。このフィールドは、Oracle RAC の展開との互換性を有効にするために必要です。</p> <p>RAC ノードのリストを以下のいずれかの方法で指定します。</p> <ul style="list-style-type: none"> ● database.port の値を指定します。また、rac.nodes リストの各アドレスに対して、指定されたポート値を使用します。以下に例を示します。 <pre>database.port=1521 rac.nodes=192.168.1.100,192.168.1.101</pre> <ul style="list-style-type: none"> ● database.port の値を指定します。また、リストの1つまたは複数のエントリーのデフォルトポートを上書きします。このリストには、デフォルトの database.port 値を使用するエントリーと、独自のポート値を定義するエントリーを含めることができます。以下に例を示します。 <pre>database.port=1521 rac.nodes=192.168.1.100,192.168.1.101:1522</pre> <p>database.url プロパティを使用してデータベースの生の JDBC URL を提供する場合、database.port の値を定義する代わりに、各 RAC ノードのエントリーでポート値を明示的に指定する必要があります。</p>
<p>skipped.operations</p>	<p>t</p>	<p>ストリーミング中にコネクタがスキップする操作タイプをコマンドで区切ったリスト。以下のタイプの操作をスキップするようにコネクタを設定することができます。</p> <ul style="list-style-type: none"> ● c (挿入/作成) ● u (更新) ● d (削除) ● t (省略) <p>デフォルトでは、省略操作のみがスキップされます。</p>

signal.data.collection	デフォルト値なし	シグナルをコネクターへの送信に使用されるデータコレクションの完全修飾名。このプロパティを Oracle プラグインデータベース (PDB) で使用する場合は、その値にはルートデータベースの名前を設定します。コレクション名の指定には、 <databaseName>.<schemaName>.<tableName> の形式を使用します。
incremental.snapshot.chunk.size	1024	増分スナップショットのチャンクの実行中にコネクターがメモリーを取得して読み取る行の最大数。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファーにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。
topic.naming.strategy	io.debezium.schema.SchemaTopicNamingStrategy	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する TopicNamingStrategy クラスの名前。デフォルトは SchemaTopicNamingStrategy 。
topic.delimiter	.	トピック名の区切り文字を指定します。デフォルトは . です。
topic.cache.size	10000	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。
topic.heartbeat.prefix	__debezium-heartbeat	コネクターがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 topic.prefix.topic.transaction です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は __debezium-heartbeat.fulfillment になります。

<code>topic.transaction</code>	<code>transaction</code>	<p>コネクターがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、</p> <p><code>topic.prefix.topic.transaction</code></p> <p>です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は fulfillment.transaction になります。</p>
--------------------------------	--------------------------	---

Debezium Oracle コネクターデータベーススキーマ履歴の設定プロパティ

Debezium には、コネクターがスキーマ履歴トピックと対話する方法を制御する `schema.history.internal.*` プロパティのセットが含まれています。

以下の表は、Debezium コネクターを設定するための `schema.history.internal` プロパティについて説明しています。

表6.15 コネクターデータベーススキーマ履歴の設定プロパティ

プロパティ	デフォルト	説明
<code>schema.history.internal.kafka.a.topic</code>	デフォルトなし	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。
<code>schema.history.internal.kafka.a.bootstrap.servers</code>	デフォルトなし	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。各ペアは、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。
<code>schema.history.internal.kafka.a.recovery.poll.interval.ms</code>	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。
<code>schema.history.internal.kafka.a.query.timeout.ms</code>	3000	Kafka 管理クライアントを使用してクラスター情報を取得する際に、コネクターが待機すべき最大ミリ秒数を指定する整数値です。
<code>schema.history.internal.kafka.a.create.timeout.ms</code>	30000	Kafka 管理クライアントを使用して kafka 履歴トピックを作成する間、コネクターが待機する最大ミリ秒数を指定する整数値。
<code>schema.history.internal.kafka.a.recovery.attempts</code>	100	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、 <code>recovery.attempts</code> × <code>recovery.poll.interval.ms</code> です。

プロパティ	デフォルト	説明
schema.history.internal.skip.unparseable.ddl	false	コネクターが不正または不明なデータベースのステートメントを無視するかどうか、または人が問題を修正するために処理を停止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 の処理中にデータの損失や分割を引き起こす可能性があるため、必ず注意して使用する必要があります。
schema.history.internal.store.only.captured.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かどうかを指定するブール値 true は、変更が Debezium によってキャプチャーされるテーブルに関連する DDL ステートメントのみを記録します。変更がキャプチャーされたテーブルを変更すると、不足しているデータが必要になる可能性があるため、 true に設定するには注意が必要です。 安全なデフォルトは false です。

プロデューサーおよびコンシューマクライアントを設定するためのパススルーデータベーススキーマ履歴プロパティ

Debezium は、Kafka プロデューサーを使用して、データベーススキーマ履歴トピックにスキーマの変更を書き込みます。同様に、コネクターが起動すると、データベーススキーマ履歴トピックから読み取る Kafka コンシューマーに依存します。 **schema.history.internal.producer.*** および **schema.history.internal.consumer.*** 接頭辞で始まるパススルー設定プロパティのセットに値を割り当てて、Kafka プロデューサーおよびコンシューマクライアントの設定を定義します。パススループロデューサーおよびコンシューマデータベーススキーマ履歴プロパティは、以下の例のように Kafka ブローカーとのこれらのクライアントの接続をセキュアにする方法など、さまざまな動作を制御します。

```

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

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

```

Debezium は、プロパティを Kafka クライアントに渡す前に、プロパティ名から接頭辞を削除します。

[Kafka プロデューサー設定プロパティ](#) および [Kafka コンシューマー設定プロパティ](#) の詳細は、Kafka のドキュメントを参照してください。

Debezium Oracle コネクターパススルーデータベースドライバースettingsプロパティ

Debezium コネクターでは、データベースドライバースettingsのパススルー設定が可能です。パススルーデータベースプロパティは接頭辞 **driver.*** で始まります。たとえば、コネクターは **driver.foofoo=false** などのプロパティを JDBC URL に渡します。

[データベーススキーマ履歴クライアントのパススループロパティ](#) の場合のように、Debezium はプロパティから接頭辞を削除してからデータベースドライバースettingsに渡します。

6.7. DEBEZIUM ORACLE コネクターのパフォーマンスの監視

Debezium Oracle コネクターは、Apache Zookeeper、Apache Kafka、および Kafka Connect に含まれる JMX メトリクスの組み込みサポートに加えて、3 種類のメトリクスを提供します。

- 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](#)。
- 変更イベントの処理時にコネクターを監視するための、[ストリーミングメトリクス](#)。
- 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](#)。

JMX 経由でこれらのメトリクスを公開する方法の詳細は、[監視に関するドキュメント](#) を参照してください。

6.7.1. Debezium SQL Server コネクターのスナップショットメトリクス

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

スナップショット操作がアクティブでない場合や、最後のコネクターの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	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 include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。

属性	タイプ	説明
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	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。
QueueRemainingCapacity	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。
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。

属性	タイプ	説明
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

コネクタは、増分スナップショットの実行時に、以下の追加のスナップショットメトリクスも提供します。

属性	タイプ	説明
ChunkId	string	現在のスナップショットチャンクの識別子。
ChunkFrom	string	現在のチャンクを定義するプライマリーキーセットの下限。
ChunkTo	string	現在のチャンクを定義するプライマリーキーセットの上限。
TableFrom	string	現在スナップショットされているテーブルのプライマリーキーセットの下限。
TableTo	string	現在スナップショットされているテーブルのプライマリーキーセットの上限。

6.7.2. Debezium Oracle コネクタのストリーミングメトリクス

MBean は `debezium.oracle:type=connector-metrics,context=streaming,server=<oracle.server.name>` です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。
TotalNumberOfCreateEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクターが見た更新イベントの合計数。
TotalNumberOfDeleteEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。

属性	タイプ	説明
Connected	boolean	コネクタ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との間の期間 (ミリ秒単位)。この値は、データベースサーバーとコネクタが稼働しているマシンのクロック間の差異に対応します。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは max.queue.size.in.bytes が正の長さの値に設定されている場合に利用可能です。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

Debezium Oracle コネクタは、以下のストリーミングメトリクスも追加で提供します。

表6.16 追加のストリーミングメトリクスの説明

属性	タイプ	説明
CurrentScn	string	処理された最新のシステム変更番号です。
OldestScn	string	トランザクションバッファ内の最も古いシステム変更番号。
CommittedScn	string	トランザクションバッファからの最後のコミットされたシステム変更番号。

属性	タイプ	説明
OffsetScn	string	現在、コネクターのオフセットに書き込まれているシステム変更番号。
CurrentRedoLogFileName	string[]	現在採掘されているログファイルの配列。
MinimumMinedLogCount	long	任意の Log Miner セッションに指定されたログの最小数です。
MaximumMinedLogCount	long	任意の LogMiner セッションに指定されたログの最大数。
RedoLogStatus	string[]	filename status 形式のマイニングされた各ログファイルの現在の状態の配列。
SwitchCounter	int	最終日について、データベースがログスイッチを実行した回数。
LastCapturedDmlCount	long	最後の LogMiner セッションクエリーで確認される DML 操作の数。
MaxCapturedDmlInBatch	long	単一の LogMiner セッションクエリーの処理中に確認される DML 操作の最大数。
TotalCapturedDmlCount	long	確認された DML 操作の合計数。
FetchingQueryCount	long	実行された LogMiner セッションクエリー (別名バッチ) の合計数。
LastDurationOfFetchQueryInMilliseconds	long	最後の LogMiner セッションクエリーのフェッチ時間 (ミリ秒単位)。
MaxDurationOfFetchQueryInMilliseconds	long	任意の LogMiner セッションクエリーのフェッチの最大時間 (ミリ秒単位)。
LastBatchProcessingTimeInMilliseconds	long	最後の LogMiner クエリーバッチ処理の時間 (ミリ秒単位)。

属性	タイプ	説明
TotalParseTimeInMilliseconds	long	DML イベント SQL ステートメントの解析に費やした時間 (ミリ秒単位)。
LastMiningSessionStartTimeInMilliseconds	long	最後の LogMiner セッションを開始する期間 (ミリ秒単位)。
MaxMiningSessionStartTimeInMilliseconds	long	LogMiner セッションを開始する最長期間 (ミリ秒単位)。
TotalMiningSessionStartTimeInMilliseconds	long	コネクターが LogMiner セッションを開始するのに費やす合計期間 (ミリ秒単位)。
MinBatchProcessingTimeInMilliseconds	long	単一の LogMiner セッションからの結果を処理するのに費やされた最小時間 (ミリ秒単位)。
MaxBatchProcessingTimeInMilliseconds	long	単一の LogMiner セッションからの結果を処理するのに費やされた最大時間 (ミリ秒単位)。
TotalProcessingTimeInMilliseconds	long	LogMiner セッションからの結果を処理するのに費やされた合計時間 (ミリ秒単位)。
TotalResultSetNextTimeInMilliseconds	long	ログマイニングビューからの処理する次の行を取得する JDBC ドライバーによって費やされた合計期間 (ミリ秒単位)。
TotalProcessedRows	long	すべてのセッションでログマイニングビューから処理される行の合計数。
BatchSize	int	データベースのラウンドトリップごとにログのマイニングクエリーによって取得されるエンタリーの数。
MillisecondToSleepBetweenMiningQuery	long	ログマイニングビューから結果の別のバッチを取得する前にコネクターがスリープ状態になる期間 (ミリ秒単位)。

属性	タイプ	説明
MaxBatchProcessingThroughput	long	ログマイニングビューから処理される行/秒の最大数。
AverageBatchProcessingThroughput	long	ログマイニングから処理される行/秒の平均数。
LastBatchProcessingThroughput	long	最後のバッチでログマイニングビューから処理された平均行数/秒。
NetworkConnectionProblemsCounter	long	検出された接続問題の数。
HoursToKeepTransactionInBuffer	int	トランザクションがコミットやロールバックされずにコネクターのインメモリーバッファに保持されてから破棄されるまでの時間数。 log.mining.transaction.retention を参照してください。
NumberOfActiveTransactions	long	トランザクションバッファの現在のアクティブなトランザクションの数。
NumberOfCommittedTransactions	long	トランザクションバッファのコミットされたトランザクションの数。
NumberOfRolledBackTransactions	long	トランザクションバッファのロールバックされたトランザクションの数。
CommitThroughput	long	トランザクションバッファのコミットされた1秒あたりのトランザクションの平均数。
RegisteredDmlCount	long	トランザクションバッファに登録された DML 操作の数。
LagFromSourceInMilliseconds	long	トランザクションログに変更が発生した時刻とそれがトランザクションバッファに追加された時刻の差 (ミリ秒単位)。

属性	タイプ	説明
MaxLagFromSourceInMilliseconds	long	トランザクションログに変更が発生した時刻とそれがトランザクションバッファに追加された時刻の差の最大値 (ミリ秒単位)。
MinLagFromSourceInMilliseconds	long	トランザクションログに変更が発生した時刻とそれがトランザクションバッファに追加された時刻の差の最小値 (ミリ秒単位)。
AbandonedTransactionIds	string[]	古いためにトランザクションバッファから削除された、最も新しい放棄されたトランザクション識別子の配列。 log.mining.transaction.retention.hours を参照してください。
RolledBackTransactionIds	string[]	マイニングされトランザクションバッファにロールバックされたトランザクション識別子の配列。
LastCommitDurationInMilliseconds	long	最後のトランザクションバッファコミット操作の期間 (ミリ秒単位)。
MaxCommitDurationInMilliseconds	long	最長のトランザクションバッファコミット操作の期間 (ミリ秒単位)。
ErrorCount	int	検出されたエラーの数。
WarningCount	int	検出された警告の数。
ScnFreezeCount	int	システム変更番号の繰り上げチェックが行われ、変更されなかった回数。高い値は、長時間稼働するトランザクションが進行中で、コネクタのオフセットに最近処理されたシステム変更番号をフラッシュするのを妨げていることを示す場合があります。最適な条件であれば、 0 に近い値、もしくは等しい値になるはずです。

属性	タイプ	説明
UnparsableDdlCount	int	検出されたものの、DDL パーサーで解析できなかった DDL レコードの数です。これは常に、 0 となります。しかし、解析不能な DDL をスキップすることを許可した場合、このメトリックを使用して、コネクターのログに警告が書き込まれたかどうかを判断することができます。
MiningSessionUserGlobalAreaMemoryInBytes	long	現在のマイニングセッションのユーザーグローバルエリア (UGA) のメモリー消費量 (単位: バイト)。
MiningSessionUserGlobalAreaMaxMemoryInBytes	long	すべてのマイニングセッションでの最大のユーザーグローバルエリア (UGA) のメモリー消費量 (バイト)。
MiningSessionProcessGlobalAreaMemoryInBytes	long	現在のマイニングセッションのプロセスグローバルエリア (PGA) のメモリー消費量 (単位: バイト)。
MiningSessionProcessGlobalAreaMaxMemoryInBytes	long	全マイニングセッションのプロセスグローバルエリア (PGA) の最大メモリー消費量 (バイト)。

6.7.3. Debezium Oracle コネクターのスキーマ履歴メトリクス

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

以下の表は、利用可能なスキーマ履歴メトリクスの一覧です。

属性	タイプ	説明
Status	string	データベーススキーマ履歴の状態を示す STOPPED 、 RECOVERING (ストレージから履歴を復元)、または RUNNING のいずれか。

属性	タイプ	説明
<code>RecoveryStartTime</code>	<code>long</code>	リカバリーが開始された時点のエポック秒の時間。
<code>ChangesRecovered</code>	<code>long</code>	リカバリーフェーズ中に読み取られた変更の数。
<code>ChangesApplied</code>	<code>long</code>	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。
<code>MillisecondsSinceLastRecoveredChange</code>	<code>long</code>	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)。
<code>MillisecondsSinceLastAppliedChange</code>	<code>long</code>	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)。
<code>LastRecoveredChange</code>	<code>string</code>	履歴ストアから復元された最後の変更の文字列表現。
<code>LastAppliedChange</code>	<code>string</code>	最後に適用された変更の文字列表現。

6.8. ORACLE コネクターのよくある質問

Oracle 11g はサポートされますか？

Oracle 11g はサポート対象外ですが、ベストエフォートベースで Oracle 11g との後方互換性を確保しています。Red Hat は、コミュニティを頼りに、Oracle 11g との互換性に関する懸念点を伝え、リグレーションが特定された場合にバグ修正を提供します。

Oracle LogMiner は非推奨となりましたか？

いいえ、Oracle は、Oracle 12c で Oracle LogMiner を使用した継続的なマイニングオプションのみを非推奨にし、Oracle 19c からそのオプションを削除しました。Debezium Oracle コネクターは、このオプションに依存せずに機能するため、新しいバージョンの Oracle で問題なく安全に使用できます。

オフセットの位置を変更するにはどうすればよいですか？

Debezium Oracle コネクターは、`scn` という名前のフィールドと `commit_scn` という別の名前の 2 つの重要な値をオフセットに保持します。`scn` フィールドは、コネクターが変更をキャプチャーするとき使用する `law-watermark` の開始位置を表す文字列です。

- コネクターオフセットを含むトピックの名前を見つけます。これは、`offset.storage.topic` 設定プロパティとして指定された値に基づいて設定されます。
- コネクターの最後のオフセット、格納先のキーを見つけ、そのオフセットの保存に使用するパーティションを特定します。これは、Kafka ブローカーのインストールによって提供される `kafkacat` ユーティリティスクリプトを使用して実行できます。以下に例を示します。

```
kafkacat -b localhost -C -t my_connect_offsets -f 'Partition(%p) %k %s\n'
Partition(11) ["inventory-connector",{"server":"server1"}] {"scn":"324567897",
"commit_scn":"324567897: 0x2832343233323:1"}
```

inventory-connector のキーは ["inventory-connector",{"server":"server1"}]、パーティションは **11** で、最後のオフセットはキーの後にくるコンテンツです。

3. 以前のオフセットに戻すには、コネクタを停止し、次のコマンドを発行する必要があります。

```
echo ["inventory-connector",{"server":"server1"}]
{"scn":"3245675000","commit_scn":"324567500"} | \
kafkacat -P -b localhost -t my_connect_offsets -K \ | -p 11
```

これにより、指定のキーとオフセット値を、**my_connect_offsets** トピックのパーティション **11** に書き込みます。この例では、コネクタは **324567897** ではなく SCN **3245675000** に戻します。

コネクタで、特定のオフセット SCN が含まれるログが見つけれない場合はどうなりますか？

Debezium コネクタは、コネクタオフセットに low-watermark および high-watermark SCN 値を維持します。low-watermark SCN は開始位置を表し、コネクタが正常に起動するために利用可能なオンライン redo または archive ログに存在する必要があります。コネクタがこのオフセット SCN を検出できないと報告した場合は、まだ利用可能なログに SCN が含まれていないため、コネクタが中断した場所から変更をマイニングできないことを示しています。

この場合は、2つのオプションがあります。1つ目として、コネクタの履歴トピックとオフセットを削除し、コネクタを再起動して、提案どおりに新しいスナップショットを作成します。こうすることで、すべてのトピックコンシューマーでデータ損失が発生しないようにします。2つ目として、オフセットを手動で操作し、redo または archive ログで利用可能な位置に SCN を進めます。これにより、古い SCN 値と新しく提供された SCN 値の間で発生した変更がなくなり、トピックに書き込まれなくなります。これは、推奨されません。

さまざまなマイニングストラテジーの違いは何ですか？

Debezium Oracle コネクタには、**log.mining.strategy** の2つのオプションがあります。デフォルトは **redo_in_catalog** で、この設定では、ログスイッチが検出されるたびに Oracle データディクショナリーを REDO ログに書き込むようにコネクタに指示されます。このデータディクショナリーは、Oracle LogMiner が redo および archive ログを解析するときにスキーマの変更を効果的に追跡するために必要です。このオプションでは、通常よりも多くのアーカイブログが生成されますが、データ変更のキャプチャーに影響を与えずに、キャプチャーされるテーブルをリアルタイムで操作できます。通常、このオプションはより多くの Oracle データベースメモリーを必要とし、各ログスイッチの後、Oracle LogMiner セッションとプロセスを開始するまでに少し時間がかかります。

別のオプション **online_catalog** では、データディクショナリーを redo ログには書き込まれません。代わりに、Oracle LogMiner は、テーブルの構造の現在の状態を含むオンラインデータディクショナリーを常に使用します。つまり、テーブルの構造が変更され、オンラインデータディクショナリーと一致しなくなった場合やテーブルの構造が変更された場合は、Oracle LogMiner がテーブルまたは列の名前を解決できなくなります。キャプチャーされるテーブルに対して頻繁にスキーマの変更が適用される場合は、このマイニングストラテジーのオプションを使用しないでください。テーブルのログから変更がすべてキャプチャーされ、コネクタの停止、スキーマの変更適用、コネクタの再起動を行い、テーブルのデータ変更を再開するように、すべてのデータ変更をスキーマの変更でロックステップすることが重要です。このオプションでは、必要な Oracle データベースメモリーが少なく済み、LogMiner プロセスによってデータディクショナリーをロードまたは準備する必要がないため、通常、Oracle LogMiner セッションの起動ははるかに早くなります。

SYS または SYSTEM ユーザーが行った変更がキャプチャーされないのはなぜですか？

Oracle データベースは、**SYS** および **SYSTEM** ユーザーアカウントを使用して、変更データキャプチャーに重要ではない redo ログで多数の内部操作を実行します。Debezium Oracle コネクターが Oracle LogMiner から変更を読み取ると、これら 2 つのユーザーアカウントによる変更は自動的に除外されます。このような理由から、これら 2 つのユーザーアカウントのいずれかを使用していて変更イベントが表示されない場合、これらのユーザーによる変更がキャプチャーされません。指定されたシステム以外のユーザーアカウントを使用して、全変更を実行してキャプチャーする必要があります。

コネクターが AWS での変更のキャプチャーを停止しているように見えるのはなぜですか？

タイムアウト [AWS Gateway Load Balancer](#) で 350 秒の修正されたアイドルタイムアウトにより、完了までに 350 秒を超える JDBC 呼び出しは無期限にハングする可能性があります。

Oracle LogMiner API の呼び出しが完了するまでに 350 秒以上かかる状況では、タイムアウトが発生し、AWS Gateway Load Balancer がハングアップすることがあります。例えば、大量のデータを処理する LogMiner セッションと Oracle の定期的なチェックポイントタスクが同時に実行された場合、このようなタイムアウトが発生する可能性があります。

AWS Gateway Load Balancer でタイムアウトが発生しないように、Kafka Connect 環境から root または superuser で次の手順を実行して、キープアライブパケットを有効にします。

1. ターミナルから、以下のコマンドを実行します。

```
sysctl -w net.ipv4.tcp_keepalive_time=60
```

2. `/etc/sysctl.conf` を編集し、以下のように以下の変数の値を設定します。

```
net.ipv4.tcp_keepalive_time=60
```

3. Oracle コネクターが `database.hostname` ではなく `database.url` プロパティを使用するように再設定し、以下の例のように Oracle 接続文字列記述子 (**ENABLE=broken**) を追加します。

```
database.url=jdbc:oracle:thin:username/password!@(DESCRIPTION=(ENABLE=broken)
(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(Host=hostname)(Port=port)))
(CONNECT_DATA=(SERVICE_NAME=serviceName)))
```

前述の手順では、TCP ネットワークスタックが 60 秒ごとにキープアライブパケットを送信するように設定します。その結果、LogMiner API への JDBC 呼び出しが完了するまで 350 秒を超える時間がかかる場合、AWS Gateway ロードバランサーはタイムアウトしません。これにより、コネクターはデータベースのトランザクションログから変更の読み取りを続行します。

ORA-01555 の原因と対処方法は？

Debezium Oracle コネクターは、最初のスナップショットフェーズの実行時にフラッシュバッククエリーを使用します。フラッシュバッククエリーは、データベースの **UNDO_RETENTION** データベースパラメーターによって維持されるフラッシュバック領域に依存する特別なタイプのクエリーで、指定の SCN での特定の時点におけるテーブルの内容に基づいてクエリーの結果を返します。デフォルトでは、Oracle は通常、データベース管理者が増減の調節をしない限り、redo または flashback 領域を約 15 分間保持します。大きなテーブルをキャプチャーする設定の場合、最初のスナップショットを実行するのに 15 分以上かかるか、設定した **UNDO_RETENTION** の期間分かかる場合があります、最終的に次の例外が発生します。

```
ORA-01555: snapshot too old: rollback segment number 12345 with name
"_SYSSMU11_1234567890$" too small
```

この例外を扱う方法として、まずデータベース管理者と連携し、**UNDO_RETENTION** データベースパラメーターを一時的に増やすことができるかどうかを確認します。Oracle データベースの再起動は必要ありません。したがって、データベースの可用性に影響を与えることなく、オンラインで実行できます。ただし、これを変更しても、テーブルスペースに、必要な UNDO データを格納する領域が十分でない場合、上記の例外か、「snapshot too old」の例外が発生する可能性があります。

この例外を処理する 2 つ目の方法として、**snapshot.mode** を **schema_only** に設定し、最初のスナップショットではなく、増分スナップショットに依存します。増分スナップショットはフラッシュバッククエリーに依存しないため、ORA-01555 例外の対象ではありません。

ORA-04036 の原因と対処方法は？

データベースの変更が頻繁に発生すると、Debezium Oracle コネクターは ORA-04036 例外を報告する可能性があります。Oracle LogMiner セッションが開始され、ログの切り替えが検出されるまで再利用されます。セッションは、Oracle LogMiner で最適なパフォーマンス使用率を達成できるように再利用されますが、マイニングセッションが長時間実行されると、PGA メモリーが過剰に使用され、最終的に次のような例外が発生する可能性があります。

ORA-04036: PGA memory used by the instance exceeds PGA_AGGREGATE_LIMIT

この例外は、Oracle が redo ログを切り替える頻度、または Debezium Oracle コネクターがマイニングセッションを再利用できる期間を指定することで回避できます。Debezium Oracle コネクターには、設定オプション **log.mining.session.max.ms** があり、現在の Oracle LogMiner セッションを終了して新しいセッションを開始する前に再利用できる期間を制御します。これにより、データベースで許可されている PGA メモリーを超えることなく、データベースリソースを管理できます。

ORA-01882 の原因と対処方法は？

Debezium Oracle コネクターは、Oracle データベースへの接続時に以下の例外を報告する可能性があります。

ORA-01882: timezone region not found

これは、JDBC ドライバーでタイムゾーン情報を正しく解決できない場合に発生します。このドライバー関連の問題を解決するには、地域を使用してタイムゾーンの詳細を解決しないようにドライバーに指示する必要があります。これには、**driver.oracle.jdbc.timezoneAsRegion=false** を使用してドライバーパススループロパティを指定します。

ORA-25191 の原因と対処方法は？

Debezium Oracle コネクターは、インデックス設定テーブル (IOT) が Oracle LogMiner でサポートされていないため、IOT を自動的に無視します。ただし、ORA-25191 例外が出力された場合は、そのようなマッピングに固有のまれなケースが原因である可能性があり、自動的に除外するには追加のルールが必要になる場合があります。ORA-25191 例外の例を以下に示します。

ORA-25191: cannot reference overflow table of an index-organized table

ORA-25191 例外が出力された場合は、テーブルとそのマッピング、他の親テーブルに関連する内容など、Jira で問題を起票してください。回避策として、包含/除外設定オプションを調整して、コネクターがそのようなテーブルにアクセスできないようにします。

第7章 POSTGRESQL の DEBEZIUM コネクタ

Debezium の PostgreSQL コネクタは、PostgreSQL データベースのスキーマで行レベルの変更をキャプチャーします。このコネクタと互換性のある MongoDB のバージョンについては、[Debezium Supported Configurations page](#) を参照してください。

PostgreSQL サーバまたはクラスターに初めて接続すると、コネクタはすべてのスキーマの整合性スナップショットを作成します。スナップショットの完了後、コネクタはデータベースのコンテンツを挿入、更新、および削除する行レベルの変更を継続的にキャプチャーします。これらの行レベルの変更は、PostgreSQL データベースにコミットされています。コネクタはデータの変更イベントレコードを生成し、それらを Kafka トピックにストリーミングします。各テーブルのデフォルトの動作では、コネクタは生成されたすべてのイベントをそのテーブルの個別の Kafka トピックにストリーミングします。アプリケーションとサービスは、そのトピックからのデータ変更イベントレコードを使用します。

Debezium PostgreSQL コネクタを使用するための情報および手順は、以下のように設定されています。

- [「Debezium PostgreSQL コネクタの概要」](#)
- [「Debezium PostgreSQL コネクタの仕組み」](#)
- [「Debezium PostgreSQL コネクタのデータ変更イベントの説明」](#)
- [「Debezium PostgreSQL コネクタによるデータ型のマッピング方法」](#)
- [「Debezium コネクタを実行するための PostgreSQL の設定」](#)
- [「Debezium PostgreSQL コネクタのデプロイメント」](#)
- [「Debezium PostgreSQL コネクタのパフォーマンスの監視」](#)
- [「Debezium PostgreSQL コネクタによる障害および問題の処理方法」](#)

7.1. DEBEZIUM POSTGRESQL コネクタの概要

PostgreSQL の [論理デコード](#) 機能は、バージョン 9.4 で導入されました。これは、トランザクションログにコミットされた変更の抽出を可能にし、[出力プラグイン](#) を用い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。出力プラグインを使用すると、クライアントは変更を使用できます。

PostgreSQL コネクタには、連携してデータベースの変更を読み取りおよび処理する 2 つの主要部分が含まれています。

- **pgoutput** は、PostgreSQL 10+ の標準的な論理デコード出力プラグインです。これは、この Debezium リリースでサポートされている唯一の論理デコード出力プラグインです。このプラグインは PostgreSQL コミュニティにより維持され、PostgreSQL 自体によって [論理レプリケーション](#) に使用されます。このプラグインは常に存在するため、追加のライブラリーをインストールする必要はありません。Debezium コネクタは、raw レプリケーションイベントストリームを直接変更イベントに変換します。
- PostgreSQL の [ストリーミングレプリケーションプロトコル](#) および PostgreSQL [JDBC ドライバー](#) を使用して、論理デコード出力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)。

コネクタは、キャプチャーされた各行レベルの挿入、更新、および削除操作の **変更イベント** を生成

し、個別の Kafka トピックの各テーブルに対する変更イベントレコードを送信します。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、これらのトピックから受け取るすべての行レベルイベントに対応できます。

通常、PostgreSQL は一定期間後にログ先行書き込み (WAL、write-ahead log) をページします。つまり、コネクタ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、コネクタはすべてのデータの整合性のあるビューで開始し、スナップショットの作成中に加えられた変更は省略されません。

コネクタはフォールトトレラントです。コネクタは変更を読み取り、イベントを生成するため、各イベントの WAL の位置を記録します。コネクタ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)、コネクタは再起動後に最後に停止した場所から 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。スナップショット中にコネクタが停止した場合、コネクタは再起動時に新しいスナップショットを開始します。

重要

コネクタは PostgreSQL の論理デコード機能に依存および反映します。これには、以下の制限があります。

- 論理デコードは DDL の変更をサポートしません。よって、コネクタは DDL の変更イベントをコンシューマーに報告できません。
- 論理デコードの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。PostgreSQL サーバーのクラスターがある場合、コネクタはアクティブな **primary** サーバー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。**primary** サーバーが失敗するか降格されると、コネクタは停止します。**primary** サーバーの回復後に、コネクタを再起動できます。別の PostgreSQL サーバーが **primary** に昇格された場合は、コネクタの設定を調整してからコネクタを再起動します。

[問題が発生した場合の動作](#) では、問題の発生時にコネクタがどのように対応するかが説明されています。

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ています。1バイト文字エンコーディングでは、拡張 ASCII コード文字が含まれる文字列を正しく処理できません。

7.2. DEBEZIUM POSTGRESQL コネクタの仕組み

Debezium PostgreSQL コネクタを最適に設定および実行するには、コネクタによるスナップショットの実行方法、変更イベントのストリーム方法、Kafka トピック名の決定方法、およびメタデータの使用方法を理解すると便利です。

詳細は以下を参照してください。

- [「Debezium PostgreSQL コネクタによるデータベーススナップショットの実行方法」](#)
- [「Debezium PostgreSQL コネクタによる変更イベントレコードのストリーミング方法」](#)

- 「[Debezium PostgreSQL の変更イベントレコードを受信する Kafka トピックのデフォルト名](#)」
- 「[トランザクション境界を表す Debezium PostgreSQL コネクタによって生成されたイベント](#)」

7.2.1. PostgreSQL コネクタのセキュリティー

Debezium コネクタを使用して PostgreSQL データベースから変更をストリーミングするには、コネクタは特定の権限がデータベースで必要になります。必要な権限を付与する方法の1つとして、ユーザーに **superuser** 権限を付与する方法がありますが、これにより PostgreSQL データが不正アクセスによって公開される可能性があります。Debezium ユーザーに過剰な権限を付与するのではなく、特定の特権を付与する専用の Debezium レプリケーションユーザーを作成することが推奨されます。

Debezium PostgreSQL ユーザーの権限設定の詳細は、[パーミッションの設定](#) を参照してください。PostgreSQL の論理レプリケーションセキュリティーの詳細は、[PostgreSQL のドキュメント](#) を参照してください。

7.2.2. Debezium PostgreSQL コネクタによるデータベーススナップショットの実行方法

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に設定されています。つまり、PostgreSQL コネクタは WAL ののみを読み取ってもデータベースの履歴全体を確認できません。そのため、コネクタ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。スナップショットを実行するためのデフォルト動作は、以下の手順で設定されます。この動作を変更するには、[snapshot.mode コネクタ設定プロパティ](#) を **initial** 以外の値に設定します。

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での後続の読み取りがデータの単一バージョンに対して行われるようにします。他のクライアントによる後続の **INSERT**、**UPDATE**、および **DELETE** 操作によるデータの変更は、このトランザクションでは確認できません。
2. 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。
3. 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。
4. トランザクションをコミットします。
5. コネクタ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。

コネクタに障害が発生した場合、コネクタのリバランスが発生した場合、または1の後で5の完了前に停止した場合、コネクタは再起動後に新しいスナップショットを開始します。コネクタが最初のスナップショットを完了すると、PostgreSQL コネクタは手順2で読み取る位置からストリーミングを続行します。これにより、コネクタが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が再び停止した場合、コネクタは再起動後に最後に停止した位置から変更のストリーミングを続行します。

表7.1 [snapshot.mode](#) コネクタ設定プロパティのオプション

オプション	説明
-------	----

オプション	説明
always	<p>コネクターは起動時に常にスナップショットを実行します。スナップショットが完了した後、コネクターは上記の手順の 3. から変更のストリーミングを続行します。このモードは、以下のような状況で使用すると便利です。</p> <ul style="list-style-type: none"> 一部の WAL セグメントが削除され、利用できなくなったことを認識している。 クラスタの障害後に、新しいプライマリーが昇格された。always スナップショットモードを使用すると、新しいプライマリーが昇格された後、コネクターが新しいプライマリーで再起動するまでに加えられた変更をコネクターが見逃さないようにすることができます。
never	<p>コネクターはスナップショットを実行しません。このようにコネクターを設定したすると、起動時の動作は次のようになります。Kafka オフセットトピックに以前保存された LSN がある場合、コネクターはその位置から変更をストリーミングを続行します。保存された LSN がない場合、コネクターはサーバーで PostgreSQL の論理レプリケーションスロットが作成された時点で変更のストリーミングを開始します。never スナップショットモードは、対象のすべてのデータが WAL に反映されている場合にのみ便利です。</p>
initial_only	<p>コネクターはデータベースのスナップショットを実行し、変更イベントレコード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、スナップショットの完了時に停止します。</p>
exported	<p>非推奨、全てのモードがロックレスになります。</p>

7.2.2.1. アドホックスナップショット

デフォルトでは、コネクターは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクターではスナップショットプロセスは繰り返し処理されません。コネクターがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクターを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。テーブルデータを再キャプチャーするメカニズムを提供するため、Debezium にはアドホックスナップショットを実行するオプションがあります。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクター設定は、異なるテーブルセットをキャプチャーするように変更されます。
- Kafka トピックを削除して、再構築する必要があります。
- 設定エラーや他の問題が原因で、データの破損が発生します。

アドホックと呼ばれるスナップショットを開始することで、以前にスナップショットをキャプチャーしたテーブルのスナップショットを再実行できます。アドホックスナップショットには、[シグナルテーブル](#)を使用する必要があります。シグナルリクエストを Debezium シグナルテーブルに送信して、アドホックスナップショットを開始します。

既存のテーブルのアドホックスナップショットを開始すると、コネクタはテーブルにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、[トピックの自動作成](#)が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックのスナップショットシグナルは、スナップショットに追加するテーブルを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のテーブルのサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のテーブルの内容のサブセットをキャプチャできます。

execute-snapshot メッセージをシグナルテーブルに送信してキャプチャーするテーブルを指定します。以下の表で説明されているように、**run-snapshot** シグナルのタイプを **incremental** に設定し、スナップショットに追加するテーブルの名前を指定します。

表7.2 アドホックの **execute-snapshot** シグナルレコードの例

フィールド	デフォルト	値
type	incremental	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 incremental スナップショットのみです。
data-collections	該当なし	スナップショットされるテーブルの完全修飾名にマッチする正規表現を含む配列。名前の形式は signal.data.collection 設定オプションと同じです。
additional-condition	該当なし	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

execute-snapshot シグナルタイプのエントリーをシグナルテーブルに追加して、アドホックスナップショットを開始します。コネクタがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各テーブルの開始ポイントおよびエンドポイントとして使用します。テーブルのエントリー数と設定されたチャンクサイズに基づいて、Debezium はテーブルをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、**execute-snapshot** アクションタイプは [増分スナップショット](#) のみをトリガーします。詳細は、[スナップショットの増分](#)を参照してください。

7.2.2.2. 増分スナップショット

スナップショットを柔軟に管理するため、Debezium には **増分スナップショット** と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、[Debezium コネクタにシグナルを送信するための](#) Debezium メカニズムに依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各テーブルを段階的にキャプチャーします。スナップショットがキャプチャーするテーブルと、[各チャンクのサイズ](#)を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

増分スナップショットが進むと、Debezium はウォーターマークを使用して進捗を追跡し、キャプチャーする各テーブル行のレコードを管理します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクターはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセスが再開すると、スナップショットは最初からテーブルをキャプチャーするのではなく、停止した時点から開始します。
- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。たとえば、コネクター設定を変更してテーブルを `table.include.list` プロパティーに追加した後スナップショットを再実行します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各テーブルをプライマリーキー別に分類して、[設定されたチャンクサイズ](#)に基づいてテーブルをチャンクに分割します。チャンクごとに作業し、テーブルの行ごとにチャンクでキャプチャーします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットの作成が進むにつれ、他のプロセスがデータベースへのアクセスを継続し、テーブルレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT**、**UPDATE**、**DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できます。つまり、ストリーミングプロセスは、スナップショットがその行の **READ** イベントが含まれるチャンクをキャプチャーする前に、テーブルの行を変更するイベントを生成する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリーミングされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて入ってきた **READ** イベントと、同じテーブルの行を変更するストリーミングイベント間の競合の解決を容易にするために、Debezium は **スナップショットウィンドウ** と呼ばれるものを使用します。スナップショットウィンドウは、増分スナップショットが指定のテーブルチャンクのデータをキャプチャーしている途中に、間隔を決定します。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は 2 種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャーするスナップショットレコードは、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをテーブルの Kafka トピックに出力します。

コネクタは各スナップショットチャンクにプロセスを繰り返します。



警告

PostgreSQL の Debezium コネクタでは、増分スナップショットの実行中のスキーマの変更はサポートしません。増分スナップショットの開始 **前** にスキーマの変更が行われ、シグナルが送信された **後** にスキーマの変更が行われた場合は、スキーマの変更を正しく処理するために、パススルーの設定オプション **database.autosave** が **conservative** に設定されます。

7.2.2.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、[アドホックスナップショットシグナル](#) をソースデータベースのシグナルテーブルに送信することです。

シグナルを SQL **INSERT** クエリーとしてシグナルテーブルに送信します。

Debezium がシグナルテーブルの変更を検出すると、シグナルを読み取り、要求されたスナップショット操作を実行します。

送信するクエリーはスナップショットに追加するテーブルを指定し、必要に応じてスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに追加するテーブルを指定するには、テーブルをリストする **data-collections** 配列またはテーブルの照合に使用する正規表現の配列を指定します。以下に例を示します。

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

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるテーブルの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのテーブルを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **My.Table** のテーブルを含めるには、**"public"."My.Table"** の形式を使用します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

1. SQL クエリーを送信し、アドホック増分スナップショット要求をシグナルテーブルに追加します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
'execute-snapshot', 3
'{"data-collections": ["schema1.table1", "schema2.table2"], 4
"type":"incremental"}, 5
"additional-condition":"color=blue"}'); 6
```

コマンドの **id**、**type**、および **data** パラメーターの値は、**シグナルテーブルのフィールド** に対応します。

以下の表では、この例のパラメーターについて説明しています。

表7.3 シグナルテーブルに増分スナップショットシグナルを送信する SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。

項目	値	説明
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリーへのログメッセージを特定します。Debezium はこの文字列を使用しません。代わりに、スナップショット作成中に、Debezium は独自の ID 文字列をウォーターマークシグナルとして生成します。
3	execute-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるテーブル名の配列またはテーブル名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクタのシグナリングテーブル名を指定するのと同じ形式を使用します。
5	incremental	実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 値を指定しない場合には、コネクタは増分スナップショットを実行します。
6	additional-condition	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。 additional-condition パラメーターの詳細は、 additional-condition 付きのアドホック増分スナップショット を参照してください。

additional-condition 付きのアドホック増分スナップショット

スナップショットに、テーブル内のコンテンツのサブセットのみを含める場合は、スナップショットシグナルシグナルに **additional-condition** パラメーターを追加してシグナル要求を変更できます。

一般的なスナップショットの SQL クエリーは、以下の形式を取ります。

```
SELECT * FROM <tableName> ....
```

additional-condition パラメーターを追加して、以下の例のように **WHERE** 条件を SQL クエリーに追加します。

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```

以下の例は、シグナルテーブルに追加の条件を含むアドホック増分スナップショット要求を送信する SQL クエリーを示しています。

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


たとえば、以下の列が含まれる **products** テーブルがあるとします。

- **id** (プライマリーキー)
- **color**
- **quantity**

products テーブルの増分スナップショットに **color=blue** のデータ項目のみを含める場合は、次の SQL ステートメントを使用してスナップショットをトリガーできます。

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

additional-condition パラメーターを使用すると、列が2つ以上となる条件を指定することもできます。たとえば、前述の例の **products** テーブルを使用して、**color=blue** および **quantity>10** だけに一致するアイテムのみのデータが含まれる増分スナップショットをトリガーするクエリーを送信できます。

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-snapshot',
{"data-collections": ["schema1.products"],"type":"incremental", "additional-condition":"color=blue AND
quantity>10"});
```

以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。現在、唯一の有効なオプションはデフォルト値 incremental だけです。シグナルテーブルに送信する SQL クエリーでの type 値の指定は任意です。値を指定しない場合には、コネクターは増分スナップショットを実行します。

項目	フィールド名	説明
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

7.2.2.4. 増分スナップショットの停止

ソースデータベースのテーブルにシグナルを送信して、増分スナップショットを停止することもできます。SQL **INSERT** クエリーを送信して、停止スナップショットシグナルをテーブルに送信します。

Debezium はシグナルテーブルの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのテーブルを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティで指定されます。

手順

- SQL クエリーを送信して、シグナリングテーブルへのアドホックインクリメンタルスナップショットを停止します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
      'stop-snapshot', 3
      '{"data-collections": ["schema1.table1", "schema2.table2"], 4
      "type": "incremental"}'); 5
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表7.4 シグナリングテーブルに増分スナップショット停止信号を送信するための SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリへのログメッセージを特定します。Debezium はこの文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドのオプションコンポーネントで、スナップショットから削除するテーブル名の配列またはテーブル名とマッチする正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

7.2.3. Debezium PostgreSQL コネクターによる変更イベントレコードのストリーミング方法

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大半の時間を費やします。このメカニズムは、[PostgreSQL のレプリケーションプロトコル](#) に依存します。このプロトコルにより、クライアントはログシーケンス番号 (LSN) と呼ばれる特定の場所で変更がサーバーのトランザクションログにコミットされる際に、サーバーから変更を受信することができます。

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、出力ストリームに書き込みます。その後、クライアントは変更を使用できます。

Debezium PostgreSQL コネクターは PostgreSQL クライアントとして動作します。コネクターが変更を受信すると、イベントを Debezium の **create**、**update**、または **delete** イベントに変換します。これには、イベントの LSN が含まれます。PostgreSQL コネクターは、同じプロセスで実行されている Kafka Connect フレームワークにレコードのこれらの変更イベントを転送します。Kafka Connect プロセスは、変更イベントレコードを適切な Kafka トピックに生成された順序で非同期的に書き込みます。

Kafka Connect は定期的に最新の **オフセット** を別の Kafka トピックに記録します。オフセットは、各イベントに含まれるソース固有の位置情報を示します。PostgreSQL コネクタでは、各変更イベントに記録された LSN がオフセットです。

Kafka Connect が正常にシャットダウンすると、コネクタを停止し、すべてのイベントレコードを Kafka にフラッシュして、各コネクタから受け取った最後のオフセットを記録します。Kafka Connect の再起動時に、各コネクタの最後に記録されたオフセットを読み取り、最後に記録されたオフセットで各コネクタを起動します。コネクタを再起動すると、PostgreSQL サーバーにリクエストを送信し、その位置の直後に開始されるイベントを送信します。

注記

PostgreSQL コネクタは、論理デコード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。ただし、コネクタはプライマリーキーが設定される列に関する情報を取得しません。コネクタは JDBC メタデータ (サイドチャネル) からこの情報を取得します。テーブルのプライマリーキー定義が変更される場合 (プライマリーキー列の追加、削除、または名前変更によって)、変更される場合、JDBC からのプライマリーキー情報が論理デコードプラグインが生成する変更イベントと同期されないごくわずかな期間が発生します。このごくわずかな期間に、キーの構造が不整合な状態でメッセージが作成される可能性があります。不整合にならないようにするには、以下のようにプライマリーキーの構造を更新します。

1. 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します。
2. Debezium に残りのイベントをすべて処理させます。
3. Debezium を停止します。
4. 関連するテーブルのプライマリーキー定義を更新します。
5. データベースまたはアプリケーションを読み取り/書き込みモードにします。
6. Debezium を再起動します。

PostgreSQL 10+ 論理デコードサポート (pgoutput)

PostgreSQL 10+ の時点で、PostgreSQL でネイティブにサポートされる **pgoutput** と呼ばれる論理レプリケーションストリームモードがあります。つまり、Debezium PostgreSQL コネクタは追加のプラグインを必要とせずにその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。

7.2.4. Debezium PostgreSQL の変更イベントレコードを受信する Kafka トピックのデフォルト名

デフォルトでは、PostgreSQL コネクタは、テーブルで発生するすべての **INSERT**、**UPDATE**、**DELETE** 操作の変更イベントを、そのテーブルに固有の単一の Apache Kafka トピックに書き込みます。コネクタは以下の規則を使用して変更イベントトピックに名前を付けます。

`topicPrefix.schemaName.tableName`

以下のリストは、デフォルト名のコンポーネントの定義を示しています。

`topicPrefix`

`topic.prefix` コネクタ設定プロパティで指定されたトピック接頭辞。

schemaName

変更イベントが発生したデータベーススキーマの名前。

tableName

変更イベントが発生したデータベーステーブルの名前。

例えば、**postgres** データベースと、**products**、**products_on_hand**、**customers**、**orders** の4つのテーブルを含む **inventory** スキーマを持つ PostgreSQL インスタレーションの変更をキャプチャするコネクターの設定において、**fulfillment** が論理的なサーバー名であるとします。コネクターは以下の4つの Kafka トピックにレコードをストリーミングします。

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

テーブルは特定のスキーマの一部ではなく、デフォルトの **public** PostgreSQL スキーマで作成されたとします。Kafka トピックの名前は以下になります。

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

コネクターは、同様の命名規則を適用して、[トランザクションメタデータのトピック](#)をラベル付けします。

デフォルトのトピック名が要件を満たさない場合は、カスタムトピック名を設定できます。カスタムトピック名を設定するには、論理トピックルーティング SMT に正規表現を指定します。論理トピックルーティング SMT を使用してトピックの命名をカスタマイズする方法は、[トピックルーティング](#)を参照してください。

7.2.5. トランザクション境界を表す Debezium PostgreSQL コネクターによって生成されたイベント

Debezium は、トランザクション境界を表し、データ変更イベントメッセージをエンリッチするイベントを生成できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクターのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクターをデプロイする前に発生するトランザクションのメタデータは利用できません。

Debezium はすべてのトランザクションの **BEGIN** および **END** に対して、以下のフィールドが含まれるイベントを生成します。

status

BEGIN または **END**

id

Postgres トランザクション ID 自体と、コロンで区切られた特定の操作の LSN で設定される一意の トランザクション識別子の文字列表現。形式は **txID:LSN** です。

ts_ms

データソースでのトランザクション境界イベント (**BEGIN** または **END** イベント) の時間。もしデータソースが Debezium にイベント時間を提供しないなら、このフィールドは代わりに Debezium が イベントを処理する時間を表します。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

data_collection と **event_count** 要素のペアの配列。これは、コネクターがデータコレクションから発信された変更に対して出力するイベントの数を示します。

例

```
{
  "status": "BEGIN",
  "id": "571:53195829",
  "ts_ms": 1486500577125,
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "571:53195832",
  "ts_ms": 1486500577691,
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "s1.a",
      "event_count": 1
    },
    {
      "data_collection": "s2.a",
      "event_count": 1
    }
  ]
}
```

topic.transaction オプションで上書きされない限り、トランザクションイベントは **<topic.prefix>.transaction** という名前のトピックに書き込まれます。

変更データイベントのエンリッチメント

トランザクションメタデータを有効にすると、データメッセージ **Envelope** は新しい **transaction** フィールドでエンリッチされます。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下は、メッセージの例になります。

```
{
  "before": null,
  "after": {
    "pk": "2",
    "aa": "1"
  },
  "source": {
    ...
  },
  "op": "c",
  "ts_ms": "1580390884335",
  "transaction": {
    "id": "571:53195832",
    "total_order": "1",
    "data_collection_order": "1"
  }
}
```

7.3. DEBEZIUM POSTGRESQL コネクターのデータ変更イベントの説明

Debezium PostgreSQL コネクターは、行レベルの **INSERT**、**UPDATE**、および **DELETE** 操作ごとにデータ変更イベントを生成します。各イベントにはキーと値が含まれます。キーと値の構造は、変更されたテーブルによって異なります。

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** を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があります。コンシューマーによる処理が困難になることがあります。これに対応するために、各イベントにはコンテンツのスキーマが含まれます。スキーマレジストリーを使用している場合は、コンシューマーがレジストリーからスキーマを取得するために使用できるスキーマ ID が含まれます。これにより、各イベントが自己完結型になります。

以下のスケルトン JSON は、変更イベントの基本となる 4 つの部分を示しています。ただし、アプリケーションで使用するために選択した Kafka Connect コンバーターの設定方法によって、変更イベントのこれら 4 部分の表現が決定されます。**schema** フィールドは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。同様に、イベントキーおよびイベントペイロードは、変更イベントが生成されるようにコンバーターを設定した場合のみ変更イベントに含まれます。JSON コンバーターを使用し、変更イベントの基本となる 4 つの部分すべてを生成するように設定すると、変更イベントの構造は次のようになります。

```
{
  "schema": { ❶
    ...
  },
  "payload": { ❷
    ...
  },
  "schema": { ❸
```

```

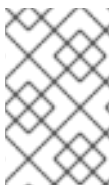
...
},
"payload": { 4
...
},
}

```

表7.5 変更イベントの基本内容の概要

項目	フィールド名	説明
1	schema	最初の schema フィールドはイベントキーの一部です。イベントキーの payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、最初の schema フィールドは、変更されたテーブルのプライマリーキーの構造、またはテーブルにプライマリーキーがない場合は変更されたテーブルの一意キーの構造を記述します。 message.key.columns コネクタ設定プロパティを設定すると、テーブルのプライマリーキーをオーバーライドできます。この場合、最初の schema フィールドはそのプロパティによって識別されるキーの構造を記述します。
2	payload	最初の payload フィールドはイベントキーの一部です。前述の schema フィールドによって記述された構造を持ち、変更された行のキーが含まれます。
3	schema	2つ目の schema フィールドはイベント値の一部です。イベント値の payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、2つ目の schema は変更された行の構造を記述します。通常、このスキーマには入れ子になったスキーマが含まれます。
4	payload	2つ目の payload フィールドはイベント値の一部です。前述の schema フィールドによって記述された構造を持ち、変更された行の実際のデータが含まれます。

デフォルトの動作では、コネクタによって、変更イベントレコードが **イベントの元のテーブル名前が同じトピック** にストリーミングされます。



注記

Kafka 0.10 以降では、任意でイベントキーおよび値を **タイムスタンプ** とともに記録できます。このタイムスタンプはメッセージが作成された (プロデューサーによって記録) 時間または Kafka によってログに買い込まれた時間を示します。



警告

PostgreSQL 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。つまり、論理サーバー名はアルファベットまたはアンダースコア (a-z、A-Z、または `_`) で始まる必要があります。論理サーバー名の残りの各文字と、スキーマ名とテーブル名の各文字は、アルファベット、数字、またはアンダースコア (a-z、A-Z、0-9、または `_`) でなければなりません。無効な文字がある場合は、アンダースコアに置き換えられます。

論理サーバー名、スキーマ名、またはテーブル名に無効な文字が含まれ、名前を区別する唯一の文字が無効であると、無効な文字はすべてアンダースコアに置き換えられるため、予期せぬ競合が発生する可能性があります。

詳細は以下を参照してください。

- [「Debezium PostgreSQL の変更イベントのキー」](#)
- [「Debezium PostgreSQL 変更イベントの値」](#)

7.3.1. Debezium PostgreSQL の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は、イベントが作成された時点でテーブルのプライマリーキーの各列のフィールドが含まれる構造を持ちます。また、テーブルの **REPLICA IDENTITY** が **FULL** または **USING INDEX** に設定されている場合は、各ユニークキー制約のフィールドがあります。

public データベーススキーマに定義されている **customers** テーブルと、そのテーブルの変更イベントキーの例を見てみましょう。

テーブルの例

```
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)
);
```

変更イベントキーの例

topic.prefix コネクター設定プロパティに **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持ち、JSON では以下のようになります。

```
{
  "schema": { 1
    "type": "struct",
    "name": "PostgreSQL_server.public.customers.Key", 2
    "optional": false, 3
    "fields": [ 4
```

```

    {
      "name": "id",
      "index": "0",
      "schema": {
        "type": "INT32",
        "optional": "false"
      }
    }
  ],
  "payload": { 5
    "id": "1"
  },
}

```

表7.6 変更イベントキーの説明

項目	フィールド名	説明
1	schema	キーのスキーマ部分は、キーの payload 部分の内容を記述する Kafka Connect スキーマを指定します。
2	PostgreSQL_server.inventory.customers.Key	<p>キーのペイロードの構造を定義するスキーマの名前。このスキーマは、変更されたテーブルのプライマリーキーの構造を記述します。キースキーマ名の形式は connector-name.database-name.table-name.Key です。この例では、以下のようになります。</p> <ul style="list-style-type: none"> ● PostgreSQL_server はこのイベントを生成したコネクタの名前です。 ● inventory は変更されたテーブルが含まれるデータベースです。 ● customers は更新されたテーブルです。
3	任意	イベントキーの payload フィールドに値が含まれる必要があるかどうかを示します。この例では、キーのペイロードに値が必要です。テーブルにプライマリーキーがない場合は、キーの payload フィールドの値は任意です。
4	fields	各フィールドの名前、インデックス、およびスキーマなど、 payload で想定される各フィールドを指定します。
5	payload	この変更イベントが生成された行のキーが含まれます。この例では、キーには値が1の id フィールドが1つ含まれます。



注記

column.exclude.list および **column.include.list** コネクタ設定プロパティは、テーブル列のサブセットのみをキャプチャできるようにしますが、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。プライマリーキーや一意キーの制約がないテーブルの行は一意に識別できません。

7.3.2. Debezium PostgreSQL 変更イベントの値

変更イベントの値はキーよりも若干複雑です。キーと同様に、値には **schema** セクションと **payload** セクションがあります。**schema** セクションには、入れ子のフィールドを含む、**Envelope** セクションの **payload** 構造を記述するスキーマが含まれています。データを作成、更新、または削除する操作のすべての変更イベントには、Envelope 構造を持つ値 payload があります。

変更イベントキーの例を紹介するために使用した、同じサンプル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)
);
```

この表への変更に対する変更イベントの値は、**REPLICA IDENTITY** 設定およびイベントの目的である操作により異なります。

詳細は、以下を参照してください。

- [Replica identity](#)
- [作成 イベント](#)
- [更新 イベント](#)
- [プライマリーキーの更新](#)
- [削除 イベント](#)
- [廃棄 \(tombstone\) イベント](#)

Replica identity

REPLICA IDENTITY は **UPDATE** および **DELETE** イベントの論理デコードプラグインで利用可能な情報量を決定する PostgreSQL 固有のテーブルレベルの設定です。具体的には、**REPLICA IDENTITY** の設定は、**UPDATE** または **DELETE** イベントが発生するたびに、関係するテーブル列の以前の値に利用可能な情報 (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** - テーブルにプライマリーキーがある場合に、**UPDATE** および **DELETE** イベントにテーブルのプライマリーキー列の以前の値が含まれることがデフォルトの動作になります。**UPDATE** イベントでは、値が変更されたプライマリーキー列のみが存在します。

テーブルにプライマリーキーがない場合、コネクタはそのテーブルの **UPDATE** または **DELETE** イベントを出力しません。プライマリーキーのないテーブルの場合、コネクタは **作成** イベントのみを出力します。通常、プライマリーキーのないテーブルは、テーブルの最後にメッセージを追加するために使用されます。そのため、**UPDATE** および **DELETE** イベントは便利ではありません。

- **NOTHING: UPDATE** および **DELETE** 操作の出力された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** 操作の出力されたイベントには、テーブルの列すべての以前の値が含まれます。
- **INDEX index-name: UPDATE** および **DELETE** 操作の発生したイベントには、指定されたインデックスに含まれる列の以前の値が含まれます。**UPDATE** イベントには、更新された値を持つインデックス化された列も含まれます。

作成 イベント

以下の例は、**customers** テーブルにデータを作成する操作に対して、コネクタによって生成される変更イベントの値の部分を示しています。

```
{
  "schema": { ❶
    "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", ❷
      "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", ❸
    "field": "source"
},
{
    "type": "string",
    "optional": false,
    "field": "op"
},
{
    "type": "int64",
    "optional": true,
    "field": "ts_ms"
}
],
"optional": false,
"name": "PostgreSQL_server.inventory.customers.Envelope" ❹
},
"payload": { ❺
    "before": null, ❻
    "after": { ❼
        "id": 1,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
    }
},

```

```

"source": { 8
  "version": "2.1.4.Final",
  "connector": "postgresql",
  "name": "PostgreSQL_server",
  "ts_ms": 1559033904863,
  "snapshot": true,
  "db": "postgres",
  "sequence": "[\"24023119\", \"24023128\"]",
  "schema": "public",
  "table": "customers",
  "txId": 555,
  "lsn": 24023128,
  "xmin": null
},
"op": "c", 9
"ts_ms": 1559033904863 10
}
}

```

表7.7 作成 イベント値フィールドの説明

項目	フィールド名	説明
1	schema	値のペイロードの構造を記述する、値のスキーマ。変更イベントの値スキーマは、コネクターが特定のテーブルに生成するすべての変更イベントで同じになります。
2	name	スキーマ セクションで、各 name フィールドは、値のペイロードのフィールドに対するスキーマを指定します。 PostgreSQL_server.inventory.customers.Value は、 before と after ペイロードのスキーマです。このスキーマは customers テーブルに固有です。 before および after フィールドのスキーマ名は logicalName.tableName.Value の形式で、スキーマ名がデータベースで一意になるようにします。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 スキーマは、それぞれ独自に進化し、独自の履歴を持つことになります。
3	name	io.debezium.connector.postgresql.Source は、ペイロードの source フィールドのスキーマです。このスキーマは、PostgreSQL コネクターに固有のもので、コネクターは生成するすべてのイベントにこれを使用します。
4	name	PostgreSQL_server.inventory.customers.Envelope は、ペイロードの全体的な構造のスキーマで、 PostgreSQL_server はコネクター名、 inventory はデータベース、 customers はテーブルを指します。

項目	フィールド名	説明
5	payload	<p>値の実際のデータ。これは、変更イベントが提供する情報です。</p> <p>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。これは、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です。しかし、Avro コンバーターを使用すると、コネクタが Kafka トピックにストリーミングするメッセージのサイズを大幅に小さくすることができます。</p>
6	before	<p>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。この例のように、op フィールドが create (作成) の c である場合、この変更イベントは新しい内容に対するものであるため、before は null になります。</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>注記</p> <p>このフィールドを利用できるかどうかは、各テーブルの REPLICA IDENTITY 設定によって異なります。</p> </div>
7	after	<p>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。この例では、after フィールドには、新しい行の id、first_name、last_name、および email 列の値が含まれます。</p>
8	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、イベントの発生元、イベントの発生順序、およびイベントが同じトランザクションの一部であるかどうかなど、このイベントと他のイベントを比較するために使用できる情報が含まれています。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ型および名前 ● 新しい行が含まれるデータベースおよびテーブル ● 追加のオフセット情報を文字列化した JSON 配列。最初の値は常に最後にコミットされた LSN で、2 番目の値は常に現在の LSN です。いずれの値も null である可能性があります。 ● スキーマ名 ● イベントがスナップショットの一部であるか ● 操作が実行されたトランザクションの ID ● データベースログの操作のオフセット ● データベースに変更が加えられた時点のタイムスタンプ

項目	フィールド名	説明
9	op	<p>コネクターによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、cは操作によって行が作成されたことを示しています。有効な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (読み取り、スナップショットのみに適用) ● t = truncate ● m = message
10	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

更新イベント

サンプル **customers** テーブルにある更新の変更イベントの値には、そのテーブルの **作成** イベントと同じスキーマがあります。同様に、イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。以下は、コネクターによって **customers** テーブルでの更新に生成されるイベントの変更イベント値の例になります。

```
{
  "schema": { ... },
  "payload": {
    "before": { ❶
      "id": 1
    },
    "after": { ❷
      "id": 1,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { ❸
      "version": "2.1.4.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": false,
      "db": "postgres",
      "schema": "public",
    }
  }
}
```

```

    "table": "customers",
    "txId": 556,
    "lsn": 24023128,
    "xmin": null
  },
  "op": "u", ④
  "ts_ms": 1465584025523 ⑤
}
}

```

表7.8 更新 イベント値フィールドの説明

項目	フィールド名	説明
1	before	データベースをコミットする前に行にあった値が含まれる任意のフィールド。この例では、テーブルの REPLICA IDENTITY 設定がデフォルトでは DEFAULT であるため、プライマリーキー列 id のみが存在します。+更新イベントに、行のすべてのコラムの以前の値が含まれるようにするには、 ALTER TABLE customers REPLICA IDENTITY FULL を実行し、 customers テーブルを変更する必要があります。
2	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。この例では、 first_name 値は Anne Marie です。
3	source	イベントのソースメタデータを記述する必須のフィールド。 source フィールド構造には 作成 イベントと同じフィールドがありますが、一部の値が異なります。ソースメタデータには以下が含まれています。 <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● 新しい行が含まれるデータベースおよびテーブル ● スキーマ名 ● イベントがスナップショットの一部である場合 (update イベントの場合は常に false) ● 操作が実行されたトランザクションの ID ● データベースログの操作のオフセット ● データベースに変更が加えられた時点のタイムスタンプ
4	op	操作の型を記述する必須の文字列。 更新 イベントの値では、 op フィールドの値は u で、更新によってこの行が変更したことを示します。

項目	フィールド名	説明
5	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>



注記

行のプライマリーキー/一意キーの列を更新すると、行のキーの値が変更されます。キーが変更されると、3つのイベントが Debezium によって出力されます。3つのイベントとは、**DELETE** イベント、行の古いキーを持つ **廃棄 (tombstone)**、およびそれに続く行の新しいキーを持つイベントです。詳細は次のセクションで説明します。

プライマリーキーの更新

行のプライマリーキーフィールドを変更する **UPDATE** 操作は、プライマリーキーの変更と呼ばれます。プライマリーキーの変更では、**UPDATE** イベントレコードの代わりにコネクターが古いキーの **DELETE** イベントレコードと、新しい (更新された) キーの **CREATE** イベントレコードを出力します。これらのイベントには通常の構造と内容があり、イベントごとにプライマリーキーの変更に関連するメッセージヘッダーがあります。

- **DELETE** イベントレコードには、メッセージヘッダーとして **__debezium.newkey** が含まれます。このヘッダーの値は、更新された行の新しいプライマリーキーです。
- **CREATE** イベントレコードには、メッセージヘッダーとして **__debezium.oldkey** が含まれます。このヘッダーの値は、更新された行にあった以前の (古い) プライマリーキーです。

delete イベント

削除 変更イベントの値は、同じテーブルの **作成** および **更新** イベントと同じ **schema** の部分になります。サンプル **customers** テーブルの **削除** イベントの **payload** 部分は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "before": { ❶
      "id": 1
    },
    "after": null, ❷
    "source": { ❸
      "version": "2.1.4.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": false,
      "db": "postgres",
    }
  }
}
```

```

    "schema": "public",
    "table": "customers",
    "txId": 556,
    "lsn": 46523128,
    "xmin": null
  },
  "op": "d", 4
  "ts_ms": 1465581902461 5
}
}

```

表7.9 削除 イベント値フィールドの説明

項目	フィールド名	説明
1	before	<p>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。delete イベント値の before フィールドには、データベースのコミットで削除される前に行にあった値が含まれます。</p> <p>この例では、テーブルの REPLICA IDENTITY 設定が DEFAULT であるため、before フィールドにはプライマリーキー列のみが含まれます。</p>
2	after	<p>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。削除 イベント値の after フィールドは null で、行が存在しないことを示します。</p>
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。削除 イベント値の source フィールド構造は、同じテーブルの 作成 および 更新 イベントと同じになります。多くの source フィールドの値も同じです。削除 イベント値では、ts_ms および lsn フィールドの値や、その他の値が変更された可能性があります。ただし、削除 イベント値の source フィールドは、同じメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● 削除された行が含まれていたデータベースとテーブル ● スキーマ名 ● イベントがスナップショットの一部であるか (常に 削除 イベントは false) ● 操作が実行されたトランザクションの ID ● データベースログの操作のオフセット ● データベースに変更が加えられた時点のタイムスタンプ
4	op	<p>操作の型を記述する必須の文字列。op フィールドの値は d で、行が削除されたことを示します。</p>

項目	フィールド名	説明
5	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

削除 変更イベントレコードは、この行の削除を処理するために必要な情報を持つコンシューマーを提供します。



警告

プライマリーキーを持たないテーブルに対して生成された **削除** イベントをコンシューマーが処理できるようにするには、テーブルの **REPLICA IDENTITY** を **FULL** に設定します。テーブルに主キーがなく、テーブルの **REPLICA IDENTITY** が **DEFAULT** または **NOTHING** に設定されている場合、**削除** イベントの **before** フィールドはありません。

PostgreSQL コネクターイベントは、[Kafka の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。ログコンパクションにより、少なくとも各キーの最新のメッセージが保持される限り、一部の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。

廃棄 (tombstone) イベント

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、**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、Kafka が同じキーを持つすべてのメッセージを削除するには、メッセージの値が **null** である必要があります。これを可能にするには、PostgreSQL コネクターは、値が **null** 値以外の同じキーを持つ特別な **廃棄** イベントが含まれる **削除** イベントに従います。

切り捨て (truncate) イベント

切り捨て (truncate) 変更イベントは、テーブルが切り捨てられていることを伝えます。この場合のメッセージキーは **null** で、メッセージの値は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "source": { ❶
      "version": "2.1.4.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
```

```

    "snapshot": false,
    "db": "postgres",
    "schema": "public",
    "table": "customers",
    "txId": 556,
    "lsn": 46523128,
    "xmin": null
  },
  "op": "t", ②
  "ts_ms": 1559033904961 ③
}
}

```

表7.10 切り捨て (truncate) イベント値フィールドの説明

項目	フィールド名	説明
1	source	<p>イベントのソースメタデータを記述する必須のフィールド。切り捨て (truncate) イベント値の source フィールド構造は、同じテーブルの作成、更新、および削除 イベントと同じで、以下のメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● 新しい行が含まれるデータベースおよびテーブル ● スキーマ名 ● イベントがスナップショットの一部であるか (常に 削除 イベントは false) ● 操作が実行されたトランザクションの ID ● データベースログの操作のオフセット ● データベースに変更が加えられた時点のタイムスタンプ
2	op	<p>操作の型を記述する必須の文字列。op フィールドの値は t で、このテーブルが切り捨てされたことを示します。</p>
3	ts_ms	<p>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。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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

1つの **TRUNCATE** ステートメントが複数のテーブルに適用された場合、切り捨てられたテーブルごとに1つの切り捨て (truncate) 変更イベントレコードが出力されます。

切り捨て (truncate) イベントは、テーブル全体に加えた変更を表し、メッセージキーを持たないの

で、単一のパーティションを持つトピックを使用しない限り、テーブルに関する変更イベント (**作成**、**更新** など) とそのテーブルの **切り捨て (truncate)** イベントの順番は保証されません。たとえば、これらのイベントが異なるパーティションから読み取られる場合、コンシューマーは **更新** イベントを **切り捨て (truncate)** イベントの後でのみ受け取る可能性があります。



メッセージイベント

このイベントタイプは、Postgres 14+ の **pgoutput** プラグインでのみサポートされています ([Postgres ドキュメント](#))。

メッセージイベントは、一般的に **pg_logical_emit_message** 関数を使用して、汎用の論理デコードメッセージが WAL に直接挿入されたことを通知します。メッセージキーは、ここでは **prefix** という名前の1つのフィールドを持つ **Struct** で、メッセージを挿入する際に指定された接頭辞を持ちます。トランザクションメッセージの場合、メッセージの値は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "source": { ❶
      "version": "2.1.4.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": false,
      "db": "postgres",
      "schema": "",
      "table": "",
      "txId": 556,
      "lsn": 46523128,
      "xmin": null
    },
    "op": "m", ❷
    "ts_ms": 1559033904961, ❸
    "message": { ❹
      "prefix": "foo",
      "content": "Ymfy"
    }
  }
}
```

他のイベントタイプとは異なり、非トランザクションメッセージは、関連する **BEGIN** や **END** のトランザクションイベントを持ちません。メッセージの値は、非取引メッセージの場合は以下のようになります。

```
{
  "schema": { ... },
  "payload": {
    "source": { ❶
      "version": "2.1.4.Final",
      "connector": "postgresql",
      "name": "PostgreSQL_server",
      "ts_ms": 1559033904863,
      "snapshot": false,
      "db": "postgres",
```

```

    "schema": "",
    "table": "",
    "lsn": 46523128,
    "xmin": null
  },
  "op": "m", ②
  "ts_ms": 1559033904961 ③
  "message": { ④
    "prefix": "foo",
    "content": "Ymfy"
  }
}

```

表7.11 message イベント値フィールドの説明

項目	フィールド名	説明
1	source	<p>イベントのソースメタデータを記述する必須のフィールド。メッセージ イベント値では、source フィールド構造は、いかなるメッセージ イベントの table または scheme 情報も持たず、メッセージ イベントがトランザクションである場合にのみ、tx id を持つことになります。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベース名 ● スキーマ名 (message イベントの場合は常に "") ● テーブル名 (message イベントの場合は常に "") ● イベントがスナップショットの一部である場合 (メッセージ イベントの場合は常に false) ● 操作が行われたトランザクションの ID (非トランザクションの メッセージ イベントの場合は null) ● データベースログの操作のオフセット ● トランザクションメッセージ。メッセージが WAL に挿入された時のタイムスタンプ ● トランザクション以外のメッセージ。コネクターがメッセージに遭遇したときのタイムスタンプ
2	op	<p>操作の型を記述する必須の文字列。op フィールドの値は m で、これが メッセージ イベントであることを示しています。</p>

項目	フィールド名	説明
3	ts_ms	<p>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。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。</p> <p>トランザクションメッセージ イベントの場合、source オブジェクトの ts_ms 属性は、トランザクションメッセージ イベントにおいて、データベースで変更が行われた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p> <p>非トランザクションメッセージ イベントの場合、source オブジェクトの ts_ms は、コネクターがメッセージ イベントに遭遇した時間を示し、payload.ts_ms は、コネクターがイベントを処理した時間を示します。この違いは、Postgres の一般的な論理メッセージ形式にはコミットのタイムスタンプが存在せず、非トランザクションの論理メッセージには (タイムスタンプ情報を持つ) BEGIN イベントが先行していないことに起因します。</p>
4	message	<p>メッセージのメタデータを格納するフィールド</p> <ul style="list-style-type: none"> ● 接頭辞 (テキスト) ● コンテンツ (バイナリープロセスモード の設定に基づいてエンコードされたバイト配列)

7.4. DEBEZIUM POSTGRESQL コネクターによるデータ型のマッピング方法

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。その値がどのようにイベントで示されるかは、列の PostgreSQL のデータ型によって異なります。以下のセクションでは、PostgreSQL データ型をイベントフィールドの **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説明します。

- **literal type** 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、値がどのように表現されるかを記述します。
- **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。

デフォルトのデータ型変換が要件に合わない場合は、コネクター用の [カスタムコンバータの作成](#) が可能です。

詳細は以下を参照してください。

- [基本型](#)
- [時間型](#)
- [TIMESTAMP 型](#)

- 10 進数型
- HSTORE 型
- ドメイン型
- ネットワークアドレス型
- PostGIS タイプ
- TOAST 化された値

基本型

以下の表は、コネクタによる基本型へのマッピング方法を説明しています。

表7.12 PostgreSQL の基本データ型のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
BOOLEAN	BOOLEAN	該当なし
BIT(1)	BOOLEAN	該当なし
BIT(> 1)	BYTES	io.debezium.data.Bits length パラメーターには、ビット数を表す整数が含まれます。結果となる 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。例えば、 numBytes = n/8 + (n % 8 == 0 ? 0 : 1) n はビット数。
BIT VARYING[(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 (列に長さが指定されていない場合は $2^{31}-1$)。結果となる byte[] にはビットがリトルエンディアン形式で含まれ、コンテンツに基づいてサイズが指定されます。 io.debezium.data.Bits 型の length パラメーターには、指定したサイズ (M) が格納されます。
SMALLINT, SMALLSERIAL	INT16	該当なし
INTEGER, SERIAL	INT32	該当なし
BIGINT, BIGSERIAL, OID	INT64	該当なし
REAL	FLOAT32	該当なし
DOUBLE PRECISION	FLOAT64	該当なし

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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
TIMESTAMPZ, 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。
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 (デフォルト) 日数の月平均に 365.25 / 12.0 式を使用した時間間隔の概数 (ミリ秒単位)。
INTERVAL [P]	STRING	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 に従ったインターバル値の文字列表現。たとえば P1Y2M3DT4H5M6.78S
BYTEA	BYTES または STRING	該当なし binary handling mode コネクター設定プロパティの設定に基づいて、raw バイト (デフォルト)、base64 でエンコードされた文字列、base64-url-safe-encoded 文字列、または 16 進数でエンコードされた文字列のいずれか。 Debezium は Postgres の bytea_output の設定値 hex のみをサポートしています。PostgreSQL バイナリーデータ型の詳細は、 PostgreSQL のドキュメント を参照してください。

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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、 (x,y) を持つ構造体を含みます。各フィールドは、描画ポイントの座標を表します。
LTREE	STRING	io.debezium.data.Ltree PostgreSQL の 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	n/a bigint の範囲。
NUMRANGE	STRING	n/a numeric の範囲
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。

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
TSTZRANGE	STRING	該当なし ローカルシステムのタイムゾーンが含まれる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 日付の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 の ENUM 値の文字列表現を含みます。許可される値のセットは、 allowed スキーマパラメーターで維持されます。

時間型

タイムゾーン情報が含まれる PostgreSQL の **TIMESTAMP TZ** and **TIMETZ** データ型以外に、時間型がマッピングされる仕組みは **time.precision.mode** コネクター設定プロパティの値によって異なります。ここでは、以下のマッピングについて説明します。

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

time.precision.mode=adaptive

time.precision.mode プロパティがデフォルトの **adaptive** に設定された場合、コネクターは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。これにより、イベントがデータベースの値を **正確** に表すようになります。

表7.13 **time.precision.mode** が **adaptive** の場合のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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。

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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。
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=adaptive_time_microseconds

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には、コネクタは列のデータ型定義に基づいて一時的な型のリテラル型とセマンティック型を決定します。これにより、マイクロ秒としてキャプチャーされた **TIME** フィールド以外は、イベントがデータベースの値を **正確** に表すようになります。

表7.14 **time.precision.mode** が **adaptive_time_microseconds** の場合のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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=connect

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 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 の論理型のみを処理でき、可変精度の時間値を処理できない場合に便利です。ただし、PostgreSQL はマイクロ秒の精度をサポートするため、**connect** 時間精度を指定して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**。

表7.15 **time.precision.mode** が **connect** の場合のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
DATE	INT32	org.apache.kafka.connect.data.Date エポックからの日数を表します。
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 の精度 P が許可され、マイクロ秒の精度まで保存されますが、 P が 3 よりも大きい場合は、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 の精度 P が許可され、マイクロ秒の精度まで保存されますが、 P が 3 よりも大きい場合は、このモードでは精度が失われます。

TIMESTAMP 型

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。例えば、**time.precision.mode** が **connect** に設定されていない場合、**TIMESTAMP** 値 2018-06-20 15:13:16.945104 は、**io.debezium.time.Micro Timestamp** の値 1529507596945104 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ません。

PostgreSQL は **TIMESTAMP** 列に **+/-infinite** の値を使用することをサポートしています。これらの特殊な値は、正の無限大の場合は **9223372036825200000**、負の無限大の場合は **-9223372036832400000** の値を持つタイムスタンプに変換されます。この動作は、PostgreSQL JDBC ドライバーの標準的な動作に似ています。詳細は、[org.postgresql.PGStatement](#) インターフェイスを参照してください。

10 進数型

PostgreSQL コネクター設定プロパティーの設定 **decimal.handling.mode** は、コネクターが 10 進数型をマッピングする方法を決定します。

decimal.handling.mode プロパティが **precise** に設定されている場合にはコネクタは **DECIMAL** と **NUMERIC**、**MONEY** 列すべてに 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。

表7.16 **decimal.handling.mode** が **precise** 場合のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
NUMERIC [(M[,D])]	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。
DECIMAL [(M[,D])]	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。
MONEY [(M[,D])]	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。 scale スキーマのパラメーターは、コネクタの設定プロパティである money.fraction.digits コネクタの設定プロパティです。

このルールには例外があります。スケーリング制約なしで **NUMERIC** または **DECIMAL** 型が使用されると、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ります (可変)。この場合、コネクタは **io.debezium.data.Variable Scale Decimal** を使用し、これには転送された値とスケールの両方が含まれます。

表7.17 スケーリング制約がない場合の **DECIMAL** および **NUMERIC** 型のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
NUMERIC	STRUCT	io.debezium.data.VariableScaleDecimal 転送された値のスケールが含まれる INT32 型の scale と、元の値がスケーリングされていない形式で含まれる BYTES 型の value の2つのフィールドがある構造が含まれます。
10 進数	STRUCT	io.debezium.data.VariableScaleDecimal 転送された値のスケールが含まれる INT32 型の scale と、元の値がスケーリングされていない形式で含まれる BYTES 型の value の2つのフィールドがある構造が含まれます。

decimal.handling.mode プロパティが **double** に設定されている場合、コネクターはすべての **DECIMAL**、**NUMERIC**、**MONEY** 値を Java の **double** 値として表し、次の表のようにエンコードします。

表7.18 **decimal.handling.mode** が **double** の場合のマッピング

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

decimal.handling.mode 設定プロパティの最後の設定は **string** です。この場合、コネクターは **DECIMAL**、**NUMERIC** および **MONEY** 値をフォーマットされた文字列表現として表し、それらを以下の表に示すとおりエンコードします。

表7.19 **decimal.handling.mode** が **string** の場合のマッピング

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

PostgreSQL は、**decimal.handling.mode** の設定が **string** または **double** の場合、**DECIMAL** / **NUMERIC** 値に格納される特別な値として **NaN**(not a number) をサポートしています。この場合、コネクターは **NaN** を **Double.NaN** または文字列定数 **NAN** のいずれかとしてエンコードします。

HSTORE 型

PostgreSQL コネクター設定プロパティの設定 **hstore.handling.mode** は、コネクターが **HSTORE** の値をマッピングする方法を決定します。

dhstore.handling.mode コネクター設定プロパティが **json** (デフォルト) に設定されている場合、コネクターは **HSTORE** 値を JSON 値の文字列表現として表し、以下の表で示すようにエンコードします。**hstore.handling.mode** プロパティが **map** に設定されている場合、コネクターは **HSTORE** 値に **MAP** スキーマタイプを使用します。

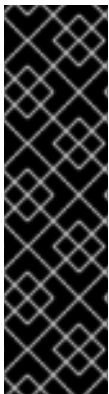
表7.20 **HSTORE** データタイプのマッピング

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

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
HSTORE	MAP	該当なし 例: JSON コンバーターを使用した出力表現: {"key" : "val"}

ドメイン型

PostgreSQL は、他の基礎となるタイプに基づいたユーザー定義の型をサポートします。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

PostgreSQL ドメイン型を使用する列で変更をキャプチャーするには、特別に考慮する必要があります。デフォルトデータベース型の1つを拡張するドメインタイプと、カスタムの長さまたはスケールを定義するドメインタイプが含まれるように列が定義されると、生成されたスキーマは定義されたその長さやスケールを継承します。

カスタムの長さまたはスケールを定義するドメインタイプを拡張する別のドメインタイプが含まれるように列が定義されていると、その情報は PostgreSQL ドライバーの列メタデータにはないため、生成されたスキーマは定義された長さやスケールを継承しません。

ネットワークアドレス型

PostgreSQL には、IPv4、IPv6、および MAC アドレスを保存できるデータ型があります。ネットワークアドレスの格納には、プレーンテキスト型ではなくこの型を使用することが推奨されます。ネットワークアドレス型は、入力エラーチェックと特化した演算子および関数を提供します。

表7.21 ネットワークアドレス型のマッピング

PostgreSQL のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
INET	STRING	該当なし IPv4 ネットワークおよび IPv6 ネットワーク
CIDR	STRING	該当なし IPv4 と IPv6 のホストおよびネットワーク
MACADDR	STRING	該当なし MAC アドレス
MACADDR8	STRING	該当なし EUI-64 形式の MAC アドレス

PostGIS タイプ

PostgreSQL コネクターは、すべての [PostGIS データ型](#) をサポートします。

表7.22 PostGIS データ型のマッピング

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) および注記
GEOMETRY (planar)	STRUCT	io.debezium.data.geometry.Geometry :フィールドが2つの構造が含まれます。 <ul style="list-style-type: none"> ● srid (INT32) - 構造に保存されるジオメトリオブジェクトの型を定義する、空間参照システム識別子。 ● wkb (BYTES) - Well-Known-Binary 形式でエンコードされたジオメトリオブジェクトのバイナリ表現。 詳細は、 Open Geospatial Consortium Simple Features Access を参照してください。
GEOGRAPHY (spherical)	STRUCT	io.debezium.data.geometry.Geography :フィールドが2つの構造が含まれます。 <ul style="list-style-type: none"> ● srid (INT32) - 構造に保存されるジオグラフィオブジェクトの型を定義する、空間参照システム識別子。 ● wkb (BYTES) - Well-Known-Binary 形式でエンコードされたジオメトリオブジェクトのバイナリ表現。 詳細は、 Open Geospatial Consortium Simple Features Access を参照してください。

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、約 8KB 以上の値は、[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、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。TOAST メカニズムを使用して保存され、変更されていない値は、テーブルのレプリカ ID の一部でない限り、メッセージに含まれません。競合が発生する可能性があるため、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。そのため、Debezium は以下のルールに従って、TOAST 化された値を処理します。

- **REPLICA IDENTITY FULL** - TOAST 列の値を持つテーブルは、他の列と同様に変更イベントの **before** および **after** フィールドの一部となります。
- **REPLICA IDENTITY DEFAULT** のあるテーブル - 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 列値はイベントに含まれません。同様に、**DELETE** イベントを受信するときに TOAST 列 (ある場合) は **before** フィールドにありません。この場合、Debezium は列値を安全に提供できないため、コネクターはコネクター設定プロパティ **unavailable.value.placeholder** によって定義されたとおりにプレースホルダー値を返します。

デフォルト値

データベーススキーマの列にデフォルト値が指定されている場合、Postgre SQL コネクタは可能な限りこの値を Kafka スキーマに反映させようとします。ほとんどの一般的なデータタイプがサポートされています。

- **BOOLEAN**
- 数値型 ((**INT**、**FLOAT**、**NUMERIC** など)
- テキストタイプ (**CHAR**、**VARCHAR**、**TEXT** など)
- 時間の種類 (**DATE**、**TIME**、**INTERVAL**、**TIMESTAMP**、**TIMESTAMPTZ**)
- **JSON**, **JSONB**, **XML**
- **UUID**

時間型の場合、デフォルト値の解析は Postgre SQL ライブラリーによって提供されることに注意してください。したがって、Postgre SQL で通常サポートされている文字列表現は、コネクタでもサポートされている必要があります。

デフォルト値がインラインで直接指定されるのではなく関数によって生成される場合、コネクタは代わりに、指定されたデータ型の **0** に相当するものをエクスポートします。これらの値は以下の通りです。

- **BOOLEAN** では **FALSE**
- 数値タイプの場合、適切な精度で **0**
- text/XML タイプの場合は空の文字列
- JSON タイプの場合は **{}**
- **1970-01-01DATE**、**TIMESTAMP**、**TIMESTAMPTZ** タイプの場合
- **TIME00:00**
- **INTERVAL** の **EPOCH**
- **00000000-0000-0000-0000-000000000000 (UUID)**

現在、このサポートは、関数の明示的な使用にのみ適用されます。たとえば、**CURRENT_TIMESTAMP(6)** は括弧付きでサポートされていますが、**CURRENT_TIMESTAMP** はサポートされていません。

重要

デフォルト値の伝搬のサポートは、主に、スキーマのバージョン間の互換性を強制するスキーマレジストリーを持つ PostgreSQL コネクタを使用する際に、スキーマを安全に進化させるために存在します。この主な問題と、異なるプラグインのリフレッシュ動作のために、Kafka スキーマに存在するデフォルト値は、データベーススキーマのデフォルト値と常に同期していることは保証されません。

- デフォルト値は、あるプラグインがいつ、どのようにインメモリースキーマの更新をトリガーするかによって、Kafka スキーマに遅れて現れることがあります。リフレッシュの間にデフォルトが何度も変更されると、Kafka スキーマに値が現れないか、スキップされることがある。
- コネクタに処理を待機しているレコードがあるときにスキーマの更新がトリガーされた場合、デフォルト値が Kafka スキーマに早期に表示されることがあります。これは、列のメタデータがレプリケーションメッセージに含まれているのではなく、リフレッシュ時にデータベースから読み取られるためです。これは、コネクタが遅れていてリフレッシュが発生した場合や、更新がソースデータベースに書き込まれ続けている間にコネクタが一時的に停止した場合に、コネクタの起動時に発生する可能性があります。

この動作は予想外かもしれませんが、それでも安全です。影響を受けるのはスキーマ定義のみで、メッセージに含まれる実際の値はソースデータベースに書き込まれたものと一貫性を保ちます。

7.5. DEBEZIUM コネクタを実行するための POSTGRESQL の設定

本リリースの Debezium では、ネイティブの **pgoutput** 論理レプリケーションストリームのみがサポートされます。**pgoutput** プラグインを使用するように 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、レプリケーションの実行に必要な権限を持つユーザーを設定します。

詳細は以下を参照してください。

- [「Debezium pgoutput プラグインのレプリケーションスロットの設定」](#)
- [「Debezium コネクタの PostgreSQL パーミッションの設定」](#)
- [「Debezium が PostgreSQL パブリケーションを作成できるように権限を設定」](#)
- [「Debezium コネクタホストでのレプリケーションを許可するように PostgreSQL を設定」](#)
- [「Debezium WAL ディスク領域の消費を管理するための PostgreSQL の設定」](#)

7.5.1. Debezium pgoutput プラグインの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。レプリケーションスロットを設定するには、**postgresql.conf** ファイルに以下を指定します。

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

これらの設定は、PostgreSQL サーバーを以下のように指示します。

- **wal_level** - 先行書き込みログで論理デコードを使用します。

- **max_wal_senders** - WAL 変更の処理に、1つの個別プロセスの最大を使用します。
- **max_replication_slots** - WAL の変更をストリーミングするために作成される1つのレプリケーションスロットの最大を許可します。

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 エントリーを保持することが保証されます。したがって、以下の点を避けるために、レプリケーションスロットを注意して監視することが重要になります。

- 過剰なディスク消費量。
- レプリケーションスロットが長期間使用されないと発生する可能性がある、あらゆる状態 (カタログの肥大化など)。

詳細は、[レプリケーションスロットに関する PostgreSQL のドキュメント](#) を参照してください。



注記

[PostgreSQL ログ先行書き込みの設定](#) や仕組みを理解していると、Debezium PostgreSQL コネクタを使用する場合に役立ちます。

7.5.2. Debezium コネクタの PostgreSQL パーミッションの設定

PostgreSQL サーバーを設定して Debezium コネクタを実行するには、レプリケーションを実行できるデータベースユーザーが必要です。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のみが実行でき、設定された数のホストに対してのみ実行できます。

[セキュリティ](#) で説明されているように、スーパーユーザーはデフォルトに必要な **REPLICATION** および **LOGIN** ロールを持っていますが、Debezium レプリケーションユーザーの権限を昇格しないことが推奨されます。代わりに、必要最低限の特権を持つ Debezium ユーザーを作成します。

前提条件

- PostgreSQL の管理者権限。

手順

1. ユーザーに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** 権限を持つ PostgreSQL ロールを定義し、そのロールをユーザーに付与します。以下に例を示します。

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

7.5.3. Debezium が PostgreSQL パブリケーションを作成できるように権限を設定

Debezium は、PostgreSQL ソーステーブルの変更イベントを、テーブル用に作成された **パブリケーション** からストリーミングします。パブリケーションには、1つ以上のテーブルから生成される変更イベントのフィルターされたセットが含まれます。各パブリケーションのデータは、パブリケーションの仕様に基づいてフィルターされます。この仕様は、PostgreSQL データベース管理者または Debezium コネクタが作成できます。Debezium PostgreSQL コネクタに、パブリケーションの作成やレプリケートするデータの指定を許可するには、コネクタはデータベースで特定の権限で操作する必要があります。

パブリケーションの作成方法を決定するオプションは複数あります。通常、コネクタを設定する前に、キャプチャーするテーブルのパブリケーションを手動で作成することが推奨されます。しかし、

Debezium がパブリケーションを自動的に作成し、それに追加するデータを指定できるように、ご使用の環境を設定できます。

Debezium は include list および exclude list プロパティを使用して、データがパブリケーションに挿入される方法を指定します。Debezium がパブリケーションを作成できるようにするオプションの詳細は、[publication.autocreate.mode](#) を参照してください。

Debezium が PostgreSQL パブリケーションを作成するには、以下の権限を持つユーザーとして実行する必要があります。

- パブリケーションにテーブルを追加するためのデータベースのレプリケーション権限。
- パブリケーションを追加するためのデータベースの **CREATE** 権限。
- 最初のテーブルデータをコピーするためのテーブルの **SELECT** 権限。テーブルの所有者には、テーブルに対する **SELECT** 権限が自動的に付与されます。

テーブルをパブリケーションに追加する場合は、ユーザーはテーブルの所有者である必要があります。ただし、ソーステーブルはすでに存在するため、元の所有者と所有権を共有する仕組みが必要です。共有所有権を有効にするには、PostgreSQL レプリケーショングループを作成した後、既存のテーブルの所有者とレプリケーションユーザーをそのグループに追加します。

手順

1. レプリケーショングループを作成します。

```
CREATE ROLE <replication_group>;
```

2. テーブルの元の所有者をグループに追加します。

```
GRANT REPLICATION_GROUP TO <original_owner>;
```

3. Debezium レプリケーションユーザーをグループに追加します。

```
GRANT REPLICATION_GROUP TO <replication_user>;
```

4. テーブルの所有権を <replication_group> に移します。

```
ALTER TABLE <table_name> OWNER TO REPLICATION_GROUP;
```

Debezium がキャプチャ設定を指定するためには、の値が [publication.autocreate.mode](#) を **filtered** に設定する必要があります。

7.5.4. Debezium コネクターホストでのレプリケーションを許可するように PostgreSQL を設定

Debezium による PostgreSQL データのレプリケーションを可能にするには、データベースを設定し、PostgreSQL コネクターを実行するホストでのレプリケーションを許可する必要があります。データベースとのレプリケーションが許可されるクライアントを指定するには、エントリーを PostgreSQL ホストベースの認証ファイル [pg_hba.conf](#) に追加します。[pg_hba.conf](#) ファイルの詳細は、[the PostgreSQL のドキュメント](#) を参照してください。

手順

- **pg_hba.conf** ファイルにエントリーを追加して、データベースホストでレプリケートできる Debezium コネクターホストを指定します。以下に例を示します。

pg_hba.conf ファイルの例です。

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

ローカル (つまりサーバーマシン上) 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。

IPv4 を使用してレプリケーションの変更を受信することを、**localhost** の **<youruser>** に許可するようサーバーに指示します。

IPv6 を使用したレプリケーション変更の受信を **localhost** の **<youruser>** に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。

7.5.5. Debezium WAL ディスク領域の消費を管理するための PostgreSQL の設定

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、異常に急上昇したり増加することがあります。このような場合、いくつかの理由が考えられます。

- コネクターがデータを受信した最大の LSN は、サーバーの **pg_replication_slots** ビューの **confirmed_flush_lsn** 列で確認できます。この LSN よりも古いデータは利用できず、データベースがディスク領域を解放します。
また、**pg_replication_slots** ビューの **restart_lsn** 列には、コネクターが必要とする可能性のある最も古い WAL の LSN が含まれています。**confirmed_flush_lsn** の値が定期的に増加し、**restart_lsn** の値に遅延が発生する場合は、データベースは領域を解放する必要があります。
データベースは、通常バッチブロックでディスク領域を解放します。これは想定内の動作であり、ユーザーによるアクションは必要ありません。
- 追跡されるデータベースには多くの更新がありますが、一部の更新のみがコネクターの変更をキャプチャーするテーブルおよびスキーマに関連します。この状況は、定期的なハートビートイベントで簡単に解決できます。コネクターの **heartbeat.interval.ms** コネクター設定プロパティを設定します。
- PostgreSQL インスタンスには複数のデータベースが含まれ、その1つがトラフィックが多いデータベースです。Debezium は、他のデータベースと比較して、トラフィックが少ない別のデータベースで変更をキャプチャーします。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

すべてのデータベースで共有されているため、Debezium が変更をキャプチャーするデータベースによってイベントが出力されるまで、使用量が増加する傾向にあります。これに対応するには、以下を行う必要があります。

- **heartbeat.interval.ms** コネクタ設定プロパティを使用して、定期的なハートビートレコードの生成を有効にします。
- Debezium が変更をキャプチャーするデータベースから変更イベントを定期的送信します。

新しい行を挿入したり、同じ行を定期的に更新することで、別のプロセスがテーブルを定期的に更新します。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 領域を解放できるようにします。このタスクは、**heartbeat.action.query** コネクタ設定プロパティを使用して自動化できます。

7.6. DEBEZIUM POSTGRESQL コネクタのデプロイメント

以下の方法のいずれかを使用して Debezium PostgreSQL コネクタをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタプラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。

関連情報

- [「Debezium PostgreSQL コネクタ設定プロパティの説明」](#)

7.6.1. AMQ Streams を使用した PostgreSQL コネクタデプロイメント

Debezium 1.7 以降、Debezium コネクタのデプロイに推奨される方法は、AMQ Streams を使用してコネクタプラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクタアーティファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクタがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクタを開始します。

Kafka Connect イメージのビルド仕様では、デプロイ可能なコネクタを指定できます。各コネクタプラグインに対して、デプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーティファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をデプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクタを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。
- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

7.6.2. AMQ Streams を使用した Debezium PostgreSQL コネクタのデプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクタをデプロイするには、最初にコネクタ用の Kafka Connect イメージをビルドする必要がありました。コネクタを OpenShift にデプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクタプラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクタ定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリまたは別の設定済みの HTTP サーバーから必要なアーティファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのデプロイに使用されます。AMQ Streams が Kafka Connect イメージをビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクタを起動します。

前提条件

- クラスター Operator がインストールされている OpenShift クラスターにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスターが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合

- [ImageStream](#) リソースを新規コンテナイメージを保存するためにクラスターにデプロイします。クラスターの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスターにログインします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例7.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium PostgreSQL コネクタアーカイブです。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクタで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプティング SMT アーカイブと、Debezium コネクタで使用する関連スクリプティングエンジン。SMT アーカイブとスクリプト言語の依存関係はオプションのコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場合にのみ、これらのコンポーネントを追加します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
    type: imagestream ❹
    image: debezium-streams-connect:latest
  plugins: ❺
    - name: debezium-connector-postgres
      artifacts:
        - type: zip ❻
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-postgres/2.1.4.Final-redhat-00001/debezium-connector-postgres-2.1.4.Final-redhat-00001-plugin.zip ❼
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3.0.Final-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3.0.Final-redhat-<build-number>.zip ❽

```

```

- type: zip
  url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip 9
- type: jar
  url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-
3.0.11.jar 10
- type: jar
  url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
jsr223/3.0.11/groovy-jsr223-3.0.11.jar
- type: jar
  url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-
json3.0.11/groovy-json-3.0.11.jar

bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

...

```

表7.23 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクタを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクタのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタプラグインに対して、コネクタと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクタアーカイブは、 .zip ファイル形式で提供されます。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。

項目	説明
7	artifacts.url の値は、コネクターアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。Debezium コネクターアーティファクトは Red Hat リポジトリで入手できます。OpenShift クラスタは指定されたサーバーにアクセスできる必要があります。
8	(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクターが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。
9	(オプション) Debezium コネクターで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベースルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 30px; height: 30px; background-color: black; margin-right: 10px;"></div> <div> <p>重要</p> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタープラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクターが失敗します。</p> </div> </div> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>別の方法として、Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>

3. 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream に

プッシュし、Kafka Connect クラスターを起動します。設定に一覧表示されているコネクタアーティファクトはクラスターで利用できます。

4. **KafkaConnector** リソースを作成し、デプロイする各コネクタのインスタンスを定義します。
たとえば、以下の **KafkaConnector** CR を作成し、**postgresql-inventory-connector.yaml** として保存します。

例7.2 Debezium コネクタの KafkaConnector カスタムリソースを定義する postgresql-inventory-connector.yaml ファイル

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-postgresql 1
spec:
  class: io.debezium.connector.postgresql.PostgresConnector 2
  tasksMax: 1 3
  config: 4
    database.hostname: postgresql.debezium-postgresql.svc.cluster.local 5
    database.port: 5432 6
    database.user: debezium 7
    database.password: dbz 8
    database.dbname: mydatabase 9
    topic.prefix: inventory-connector-postgresql 10
    table.include.list: public.inventory 11
  ...

```

表7.24 コネクタ設定の説明

項目	説明
1	Kafka Connect クラスターに登録するコネクタの名前。
2	コネクタクラスの名前。
3	同時に動作できるタスクの数。
4	コネクタの設定。
5	ホストデータベースインスタンスのアドレス。
6	データベースインスタンスのポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名前。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。

項目	説明
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターのトピック接頭辞。 指定する名前は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクタから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名前はクラスターのコネクタ間で一意である必要があります。 コネクタを Avro コネクタ と統合する場合、この名前空間は関連する Kafka Connect スキーマの名前や、対応する Avro スキーマの名前空間でも使用されます。
11	コネクタが変更イベントをキャプチャーするテーブルの一覧。

5. 以下のコマンドを実行してコネクタリソースを作成します。

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

```
oc create -n debezium -f {context}-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium PostgreSQL デプロイメントを検証する](#) 準備が整いました。

7.6.3. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium PostgreSQL コネクタのデプロイ

Debezium PostgreSQL コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、2つのカスタムリソース (CR) を作成する必要があります。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium Db2 コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用したのと同じ OpenShift インスタンスに適用します。

前提条件

- PostgreSQL が実行され、[PostgreSQL を設定して Debezium コネクタを実行する](#) 手順が実行済みである。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。

- Podman または Docker がインストールされている。
- Debezium コネクタを実行するコンテナを追加する予定のコンテナレジストリー (**quay.io** や **docker.io** など) でコンテナを作成および管理するアカウントとパーミッションを持っている。

手順

1. Kafka Connect の Debezium PostgreSQL コンテナを作成します。
 - a. **registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12** をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-postgresql.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium 2
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
postgres/2.1.4.Final-redhat-00001/debezium-connector-postgres-2.1.4.Final-redhat-
00001-plugin.zip \
&& unzip debezium-connector-postgres-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-postgres-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-postgresql.yaml** という名前の Dockerfile を作成します。

- b. 前のステップで作成した **debezium-container-for-postgresql.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

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

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

build コマンドは、**debezium-container-for-postgresql** という名前のコンテナイメージを構築します。

- c. カスタムイメージを **quay.io** などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする

OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

```
podman push <myregistry.io>/debezium-container-for-postgresql:latest
```

```
docker push <myregistry.io>/debezium-container-for-postgresql:latest
```

- d. 新しい Debezium PostgreSQL **KafkaConnect** カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  image: debezium-container-for-postgresql ❷
...
```

項目	説明
1	KafkaConnector リソースはこの Kafka Connect クラスタでコネクタを設定するために使用されることを、 metadata.annotations は Cluster Operator に示します。
2	spec.image は Debezium コネクタを実行するために作成したイメージの名前を指定します。設定された場合、このプロパティによって Cluster Operator の STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE 変数がオーバーライドされます。

- e. 以下のコマンドを実行して、**KafkaConnect** CR を OpenShift Kafka インスタンスに適用します。

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

これにより、OpenShift の Kafka Connect 環境が更新され、Debezium コネクタを実行するために作成したイメージの名前を指定する Kafka Connector インスタンスが追加されます。

2. Debezium PostgreSQL コネクタインスタンスを設定する **KafkaConnector** カスタムリソースを作成します。
通常、コネクタ設定プロパティを設定する **.yaml** ファイルに Debezium PostgreSQL コネクタを設定します。コネクタ設定は、Debezium に対して、スキーマおよびテーブルのサブセットにイベントを生成するよう指示する可能性があり、または機密性の高い、大きすぎる、または不必要な指定の列で Debezium が値を無視、マスク、または切り捨てるようにプロパティを設定する可能性もあります。Debezium PostgreSQL コネクタに設定できる設定プロパティの完全リストは [PostgreSQL コネクタプロパティ](#) を参照してください。

次の例は、ポート **5432** で PostgreSQL サーバーホスト **192.168.99.100** に接続する Debezium コネクタを設定するカスタムリソースからの抜粋です。このホストには、**sampledb** という名前のデータベース、**public** という名前のスキーマ、**inventory-connector-postgresql** という論理名のサーバーがあります。

inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector-postgresql ❶
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.postgresql.PostgresConnector
  tasksMax: 1 ❷
  config: ❸
    database.hostname: 192.168.99.100 ❹
    database.port: 5432
    database.user: debezium
    database.password: dbz
    database.dbname: sampledb
    topic.prefix: inventory-connector-postgresql ❺
    schema.include.list: public ❻
    plugin.name: pgoutput ❼
  ...
```

❶❶❶❶❶ コネクタの名前。

❷❷❷❷❷ 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。

❸❸❸ コネクタの設定。

❹❹❹ PostgreSQL サーバーを実行しているデータベースホストの名前。この例では、データベースのホスト名は **192.168.99.100** です。

❺❺❺ 一意のトピック接頭辞。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、変更イベントレコードを受信するすべての Kafka トピックの接頭辞として使用されます。

❻❻❻ コネクタは **public** スキーマでのみ変更をキャプチャーします。選択したテーブルでのみ変更をキャプチャーするようにコネクタを設定できます。詳細は [table.include.list](#) を参照してください。

❼❼❼ PostgreSQL サーバーにインストールされている PostgreSQL [論理デコードプラグイン](#) の名前。PostgreSQL 10 以降でサポートされている値は **pgoutput** のみですが、明示的に **plugin.name** を **pgoutput** に設定する必要があります。

3. Kafka Connect でコネクタインスタンスを作成します。たとえば、**KafkaConnector** リソースを **inventory-connector.yaml** ファイルに保存した場合は、以下のコマンドを実行します。

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

このコマンドは **inventory-connector** を登録して、コネクタが **KafkaConnector** CR に定義されている **sampledb** データベースに対して実行を開始します。

結果

コネクタが起動すると、コネクタが設定された PostgreSQL サーバーデータベースの [整合性スナップショットが実行](#) されます。その後、コネクタは行レベルの操作のデータ変更イベントの生成を開始し、変更イベントレコードを Kafka トピックにストリーミングします。

7.6.4. Debezium PostgreSQL コネクタが実行していることの確認

コネクタがエラーなしで正常に起動すると、コネクタがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウンストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクタが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクタのステータスを確認します。
- コネクタがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクタが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクタが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。
 - OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから、チェックするコネクタの名前をクリックします (例: **inventory-connector-postgresql**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
 - ターミナルウィンドウから以下を実行します。

- a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-postgresql -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例7.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-postgresql
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector

...

Status:
Conditions:
  Last Transition Time: 2021-12-08T17:41:34.897153Z
  Status:              True
  Type:                Ready
Connector Status:
Connector:
  State:  RUNNING
  worker_id: 10.131.1.124:8083
Name:    inventory-connector-postgresql
Tasks:
  Id:    0
  State: RUNNING
  worker_id: 10.131.1.124:8083
  Type:  source
Observed Generation: 1
Tasks Max: 1
Topics:
inventory-connector-postgresql.inventory
inventory-connector-postgresql.inventory.addresses
inventory-connector-postgresql.inventory.customers
inventory-connector-postgresql.inventory.geom
inventory-connector-postgresql.inventory.orders
inventory-connector-postgresql.inventory.products
inventory-connector-postgresql.inventory.products_on_hand
Events: <none>
```

2. コネクタによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。

- b. Search ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. KafkaTopics リストから確認するトピックの名前をクリックします (例: **inventory-connector-postgresql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例7.4 KafkaTopic リソースのステータス

NAME	CLUSTER
connect-cluster-configs	debezium-kafka-cluster 1
1 True	
connect-cluster-offsets	debezium-kafka-cluster 25
1 True	
connect-cluster-status	debezium-kafka-cluster 5
1 True	
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fad1c6a	
debezium-kafka-cluster 50 1 True	
inventory-connector-postgresql--a96f69b23d6118ff415f772679da623fbbb99421	
debezium-kafka-cluster 1 1 True	
inventory-connector-postgresql.inventory.addresses---1b6beaf7b2eb57d177d92be90ca2b210c9a56480	debezium-kafka-cluster
1 1 True	
inventory-connector-postgresql.inventory.customers---9931e04ec92ecc0924f4406af3fdace7545c483b	debezium-kafka-cluster 1
1 True	
inventory-connector-postgresql.inventory.geom---9f7e136091f071bf49ca59bf99e86c713ee58dd5	debezium-kafka-cluster
1 1 True	
inventory-connector-postgresql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d	debezium-kafka-cluster
1 1 True	
inventory-connector-postgresql.inventory.products---df0746db116844cee2297fab611c21b56f82dcef	debezium-kafka-cluster 1
1 True	
inventory-connector-postgresql.inventory.products_on_hand---8649e0f17fcc9212e266e31a7aeaa4585e5c6b5	debezium-kafka-cluster 1
1 True	
schema-changes.inventory	debezium-kafka-cluster
1 1 True	
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55	debezium-
kafka-cluster 1 1 True	kafka-cluster

```
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b debezium-kafka-cluster 1 1
True
```

3. トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。

```
oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>
```

以下に例を示します。

```
oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-postgresql.inventory.products_on_hand
```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory-connector-postgresql.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例7.5 Debezium 変更イベントの内容

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},"optional":false,"name":"inventory-
connector-postgresql.inventory.products_on_hand.Key"},"payload":{"product_id":101}}
{"schema":{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-postgresql.inventory.products_on_hand.Value","field":"before"},
{"type":"struct","fields":[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory-
connector-postgresql.inventory.products_on_hand.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":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
{"allowed":"true,last,false"},"default":"false","field":"snapshot"},
{"type":"string","optional":false,"field":"db"},
{"type":"string","optional":true,"field":"sequence"},
{"type":"string","optional":true,"field":"table"},
{"type":"int64","optional":false,"field":"server_id"},
{"type":"string","optional":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}]},"optional":false,"name":"io.debezium.connecto
```

```
r.postgresql.Source", "field": "source"}, {"type": "string", "optional": false, "field": "op"},
{"type": "int64", "optional": true, "field": "ts_ms"}, {"type": "struct", "fields":
[{"type": "string", "optional": false, "field": "id"},
{"type": "int64", "optional": false, "field": "total_order"},
{"type": "int64", "optional": false, "field": "data_collection_order"}], "optional": true, "field": "transacti
on"}], "optional": false, "name": "inventory-connector-
postgresql.inventory.products_on_hand.Envelope"}, "payload": {"before": null, "after":
{"product_id": 101, "quantity": 3}, "source": {"version": "2.1.4.Final-redhat-
00001", "connector": "postgresql", "name": "inventory-connector-
postgresql", "ts_ms": 1638985247805, "snapshot": "true", "db": "inventory", "sequence": null, "tabl
e": "products_on_hand", "server_id": 0, "gtid": null, "file": "postgresql-
bin.000003", "pos": 156, "row": 0, "thread": null, "query": null, "op": "r", "ts_ms": 1638985247805, "t
ransaction": null}}
```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

7.6.5. Debezium PostgreSQL コネクター設定プロパティの説明

Debezium PostgreSQL コネクターには、アプリケーションに適したコネクター動作を実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。プロパティに関する情報は、以下のように設定されています。

- [必要な設定プロパティ](#)
- [高度な設定プロパティ](#)
- [パススルー設定プロパティ](#)

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

表7.25 必要なコネクター設定プロパティ

プロパティ	デフォルト	説明
name	デフォルトなし	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。このプロパティはすべての Kafka Connect コネクターに必要です。
connector.class	デフォルトなし	コネクターの Java クラスの名前。PostgreSQL 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。
tasks.max	1	このコネクターのために作成する必要が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。

プロパティ	デフォルト	説明
<code>plugin.name</code>	<code>decoderbufs</code>	<p>PostgreSQL サーバーにインストールされている PostgreSQL 論理デコードプラグイン の名前。</p> <p>サポートされている値は <code>pgoutput</code> のみです。<code>pgoutput</code> には <code>plugin.name</code> を明示的に設定する必要があります。</p>
<code>slot.name</code>	<code>debezium</code>	<p>特定のデータベース/スキーマの特定のプラグインから変更をストリーミングするために作成された PostgreSQL 論理デコードスロット の名前。サーバーはこのスロットを使用して、設定する Debezium コネクタにイベントをストリーミングします。</p> <p>スロット名は PostgreSQL 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、命名ルールには各レプリケーションスロットには名前が付けられ、名前にはアルファベットの小文字、数字、およびアンダースコアを使用できます。と記載されています。</p>
<code>slot.drop.on.stop</code>	<code>false</code>	<p>コネクタが正常に想定されるように停止した場合に論理レプリケーションスロットを削除するかどうか。デフォルトの動作では、コネクタが停止したときにレプリケーションスロットはコネクタに設定された状態を保持します。コネクタが再起動すると、同じレプリケーションスロットがあるため、コネクタは停止した場所から処理を開始できます。</p> <p>テストまたは開発環境でのみ <code>true</code> に設定します。スロットを削除すると、データベースは WAL セグメントを破棄できます。コネクタが再起動すると、新しいスナップショットが実行されるか、Kafka Connect オフセットトピックの永続オフセットから続行できます。</p>

プロパティ	デフォルト	説明
publication.name	dbz_publication	<p>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。</p> <p>このパブリケーションが存在しない場合は起動時に作成され、すべてのテーブルが含まれます。Debezium は、設定されている場合は、独自の include/exclude リストフィルターを適用し、対象となる特定のテーブルのイベントのみをパブリケーションが変更するように制限します。コネクターユーザーがこのパブリケーションを作成するには、スーパーユーザーの権限が必要であるため、通常はコネクターを初めて開始する前にパブリケーションを作成することをお勧めします。</p> <p>パブリケーションがすでに存在し、すべてのテーブルが含まれているか、テーブルのサブセットで設定されている場合、Debezium は定義されているようにパブリケーションを使用します。</p>
database.hostname	デフォルトなし	PostgreSQL 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	5432	PostgreSQL データベースサーバーのポート番号 (整数)。
database.user	デフォルトなし	PostgreSQL データベースサーバーに接続するための PostgreSQL データベースユーザーの名前。
database.password	デフォルトなし	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	デフォルトなし	変更をストリーミングする PostgreSQL データベースの名前。

プロパティ	デフォルト	説明
topic.prefix	デフォルトなし	<p>Debezium が変更をキャプチャする特定の PostgreSQL データベースサーバーまたはクラスタの名前空間を提供するトピック接頭辞。接頭辞は、他のコネクタ全体で一意となる必要があります。これは、このコネクタからレコードを受信するすべての Kafka トピックのトピック名接頭辞として使用されるためです。データベースサーバーの論理名には英数字とハイフン、ドット、アンダースコアのみを使用する必要があります。</p> <div data-bbox="884 658 1428 1160" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <p style="text-align: center;"> 警告</p> <p>このプロパティの値を変更しないでください。名前の値を変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。</p> </div>
schema.include.list	デフォルトなし	<p>変更をキャプチャーする対象とするスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。schema.include.list に含まれていないスキーマ名は、変更をキャプチャーする対象から除外されます。デフォルトでは、システム以外のスキーマはすべて変更がキャプチャーされます。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの ID 全体と照合されます。</p> <p>このプロパティを設定に含める場合は、schema.exclude.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
schema.exclude.list	デフォルトなし	<p>変更をキャプチャーする対象としないスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。システムスキーマ以外で、schema.exclude.list に名前が含まれていないスキーマの変更がキャプチャーされます。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの ID 全体と照合されます。</p> <p>このプロパティを設定に含める場合は、schema.include.list プロパティを設定しないでください。</p>
table.include.list	デフォルトなし	<p>変更をキャプチャー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)。このプロパティが設定されている場合、コネクターは指定されたテーブルからのみ変更をキャプチャします。各識別子の形式は schemaName.tableName です。デフォルトでは、コネクターは変更がキャプチャーされる各スキーマのシステムでないすべてのテーブルの変更をキャプチャーします。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの ID 全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.exclude.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
table.exclude.list	デフォルトなし	<p>変更をキャプチャーしない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)。各識別子の形式は schemaName.tableName です。このプロパティが設定されている場合、コネクタは、指定のないすべてのテーブルから変更をキャプチャーします。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの ID 全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.include.list プロパティも設定しないでください。</p>
column.include.list	デフォルトなし	<p>変更イベントレコード値に含ま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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、列の名前文字列全体を一致させるために式が使用され、列名に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、column.exclude.list プロパティを設定しないでください。</p>
column.exclude.list	デフォルトなし	<p>変更イベントレコード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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、列の名前文字列全体を一致させるために式が使用され、列名に存在する可能性のある部分文字列とは一致しません。</p> <p>このプロパティを設定に含める場合は、column.include.list プロパティを設定しないでください。</p>

プロパティ	デフォルト	説明
time.precision.mode	adaptive	<p>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。</p> <p>adaptive 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およびタイムスタンプ値をキャプチャーします。</p> <p>adaptive_time_microseconds 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。例外は TIME 型フィールドで、これは常にマイクロ秒としてキャプチャーされます。</p> <p>connect、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。この組み込み表現は、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詳細は、一時的な値 を参照してください。</p>
decimal.handling.mode	precise	<p>コネクターによる DECIMAL および NUMERIC 列の値の処理方法を指定します。</p> <p>precise はバイナリー形式で変更イベントに表される java.math.BigDecimal 値を使用して正確に表します。</p> <p>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、簡単に使用できます。</p> <p>string 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。簡単に使用できますが、本来の型に関するセマンティック情報は失われます。詳細は、10 進数の型 を参照してください。</p>
hstore.handling.mode	map	<p>コネクターによる hstore 列の値の処理方法を指定します。</p> <p>map は MAP を使用して値を表します。</p> <p>json は json string を使用して値を表します。この設定では、値は {"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されます。詳細は、PostgreSQL HSTORE タイプ を参照してください。</p>

プロパティ	デフォルト	説明
interval.handling.mode	numeric	<p>numericは、マイクロ秒単位の概算値で間隔を表します。</p> <p>stringは、P<years>Y<months>M<days>DT<hours>H<minutes>M<seconds>S の文字列パターン表現を使用して間隔を正確に表します。例: P1Y2M3DT4H5M6.78S。詳細は、PostgreSQL の基本タイプ を参照してください。</p>
database.sslmode	disable	<p>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。</p> <p>disable は暗号化されていない接続を使用します。</p> <p>require はセキュアな (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。</p> <p>verify-ca は、require のように動作しますが、設定済みの認証局 (CA) 証明書に対してサーバー TLS 証明書を検証します。一致する有効な CA 証明書が見つからない場合は失敗します。</p> <p>verify-full は、verify-ca のように動作しますが、サーバー証明書がコネクターが接続しようとしているホストと一致することを検証します。詳細は the PostgreSQL のドキュメント を参照してください。</p>
database.sslcert	デフォルトなし	<p>クライアントの SSL 証明書が含まれるファイルへのパス。詳細は the PostgreSQL のドキュメント を参照してください。</p>
database.sslkey	デフォルトなし	<p>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 the PostgreSQL のドキュメント を参照してください。</p>
database.sslpassword	デフォルトなし	<p>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 the PostgreSQL のドキュメント を参照してください。</p>

プロパティ	デフォルト	説明
<code>database.sslrootcert</code>	デフォルトなし	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 the PostgreSQL のドキュメント を参照してください。
<code>database.tcpKeepAlive</code>	<code>true</code>	TCP keep-alive プロブを有効にして、データベース接続がまだ有効であることを確認します。詳細は the PostgreSQL のドキュメント を参照してください。
<code>tombstones.on.delete</code>	<code>true</code>	<p>削除 イベントの後に廃棄 (tombstone) イベントが続くかどうかを制御します。</p> <p>true: 削除操作は、削除 イベントと後続の破棄 (tombstone) イベントで表されます。</p> <p>false - delete イベントのみが出力されます。</p> <p><code>log compaction</code> がトピックで有効になっている場合には、ソースレコードの削除後に廃棄 (tombstone) イベントを出力すると (デフォルト動作)、Kafka は削除された行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。</p>
<code>column.truncate.to.length.chars</code>	該当なし	<p>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)。プロパティ名の長さ で指定された文字数を超えた場合に、一連の列のデータを切り捨てる場合は、このプロパティを設定します。length を正の整数値に設定します (例: <code>column.truncate.to.20.chars</code>)。</p> <p>列の完全修飾名は、<code><schemaName>.<tableName>.<columnName></code> の形式に従います。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>

プロパティ	デフォルト	説明
<code>column.mask.with.length.chars</code>	該当なし	<p>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)。一連の列の値をコネクタでマスクする場合 (たとえば、列に機密データが含まれている場合) は、このプロパティを設定します。 length を正の整数に設定して、指定された列のデータをプロパティ名の長さで指定されたアスタリスク (*) 文字数で置き換えます。指定した列のデータを空の文字列に置き換えるには、長さを 0 (ゼロ) に設定します。</p> <p>列の完全修飾名は、次の形式に従います: schemaName.tableName.columnName列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>

プロパティ	デフォルト	説明
<p><code>column.mask.hash.hashAlgorithm.with.salt.salt;</code> <code>column.mask.hash.v2.hashAlgorithm.with.salt.salt</code></p>	<p>該当なし</p>	<p>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)。列の完全修飾名の形式は <code><schemaName>.<tableName>.<columnName></code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。作成された変更イベントレコードでは、指定された列の値は仮名に置き換えられます。</p> <p>仮名は、指定された <code>hashAlgorithm</code> と <code>salt</code> を適用すると得られるハッシュ化された値で設定されます。使用されるハッシュ関数に基づいて、参照整合性は保持され、列値は仮名に置き換えられます。サポートされるハッシュ関数は、Java Cryptography Architecture Standard Algorithm Name Documentation の MessageDigest セクション に説明されています。</p> <p>以下の例では、CzQMA0cB5K が無作為に選択された salt になります。</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>必要な場合は、仮名は自動的に列の長さに短縮されます。コネクター設定には、異なるハッシュアルゴリズムと salt を指定する複数のプロパティを含めることができます。</p> <p>使用される <code>hashAlgorithm</code>、選択された <code>salt</code>、および実際のデータセットによっては、結果のデータセットが完全にマスクされない場合があります。</p> <p>値が異なる場所やシステムでハッシュ化されている場合は、ハッシュ化ストラテジーバージョン 2 を使用する必要があります。</p>

プロパティ	デフォルト	説明
<p><code>column.propagate.source.type</code></p>	<p>該当なし</p>	<p>列のメタデータを表す追加パラメーターをコネクタに発行させたい列の完全修飾名に一致する、オプションのコンマ区切りの正規表現のリスト。このプロパティが設定されている場合、コネクタは次のフィールドをイベントレコードのスキーマに追加します。</p> <ul style="list-style-type: none"> ● <code>__debezium.source.column.type</code> ● <code>__debezium.source.column.length</code> ● <code>__debezium.source.column.scale</code> <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクタがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、次のいずれかの形式に従います:</p> <p><code>databaseName.tableName.columnName</code>、または</p> <p><code>databaseName.schemaName.tableName.columnName</code>.</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p>

プロパティ	デフォルト	説明
<p>datatype.propagate.source.type</p>	<p>該当なし</p>	<p>データベース内の列に対して定義されているデータ型の完全修飾名を指定する正規表現のオプションのコンマ区切りリスト。このプロパティが設定されている場合、データ型が一致する列に対して、コネクターはスキーマに次の追加フィールドを含むイベントレコードを発行します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクターがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名の形式は、databaseName.tableName.typeName、または databaseName.schemaName.tableName.typeName のいずれかになります。</p> <p>データ型の名前を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データ型の名前文字列全体に対して照合されます。式は、型名に存在する可能性のある部分文字列と一致しません。</p> <p>PostgreSQL 固有のデータ型名の一覧は、PostgreSQL データ型マッピング を参照してください。</p>

プロパティ	デフォルト	説明
<p><code>message.key.columns</code></p>	<p>空の文字列</p>	<p>指定のテーブルの Kafka トピックに公開する変更イベントレコードのカスタムメッセージキーを形成するためにコネクタが使用する列を指定する式のリスト。</p> <p>デフォルトでは、Debezium はテーブルのプライマリーキー列を、出力するレコードのメッセージキーとして使用します。デフォルトの代わりに、またはプライマリーキーのないテーブルのキーを指定するには、1つ以上の列をもとにカスタムメッセージキーを設定できます。</p> <p>テーブルにカスタムメッセージキーを設定するには、テーブルを列挙した後、メッセージキーとして使用する列を列挙します。各リストエントリは、</p> <p><fully-qualified_tableName>:<keyColumn>,<keyColumn></p> <p>の形式を取ります。複数の列名をベースにテーブルキーを作成するには、列名の間にはコンマを挿入します。</p> <p>各完全修飾テーブル名は、</p> <p><schemaName>.<tableName> の形式の正規表現です。</p> <p>プロパティには複数のテーブルのエントリを含めることができます。セミコロンを使用して、リスト内のテーブルエントリを区切ります。</p> <p>以下の例は、テーブル inventory.customers および purchase.orders:</p> <p>inventory.customers:pk1,pk2; (.*)purchaseorders:pk3,pk4</p> <p>のメッセージキーを設定します。テーブル inventory.customer の場合、列 pk1 と pk2 がメッセージキーとして指定されます。どのスキーマの purchaseorders テーブルでも、pk3 と pk4 の列がメッセージキーとして使用されます。</p> <p>カスタムメッセージキーの作成に使用する列の数に制限はありません。ただし、一意の鍵を指定するために必要な最小数を使用することが推奨されます。</p>

プロパティ	デフォルト	説明
<p><code>publication.autocreate.mode</code></p>	<p><code>all_tables</code></p>	<p>pgoutput プラグインを使用して変更をストリーミングする場合にのみ適用されます。この設定は、パブリケーション の作成がどのように機能するかを決定します。次の値のいずれかを指定します。</p> <p>all_tables - コネクタは、パブリケーションが存在する場合に使用します。パブリケーションが存在しない場合は、コネクタが変更をキャプチャーするデータベースのすべてのテーブルに対してパブリケーションを作成します。コネクタがパブリケーションを作成するには、パブリケーションの作成およびレプリケーションの実行権限を持つデータベースユーザーアカウントを使用してデータベースにアクセスする必要があります。次の SQL コマンド</p> <p>CREATE PUBLICATION <code><publication_name> FOR ALL TABLES</code> を使用して、必要な権限を付与します。</p> <p>disabled - コネクタはパブリケーションの作成を試みません。レプリケーションを実行するよう設定されたデータベース管理者またはユーザーは、コネクタを実行する前にパブリケーションを作成する必要があります。コネクタがパブリケーションを見つけられない場合は、コネクタは例外を出力し、停止します。</p> <p>filtered: パブリケーションが存在する場合、コネクタはそれを使用します。パブリケーションが存在しない場合は、schema.include.list、schema.exclude.list、table.include.list、table.exclude.list の各コネクタ設定プロパティで指定された現在のフィルター設定に一致するテーブルの新しいパブリケーションが作成されます。</p> <p>例: CREATE PUBLICATION <code><publication_name> FOR TABLE <tbl1, tbl2, tbl3></code>。パブリケーションが存在する場合、コネクタは現在のフィルター設定に一致するテーブルのパブリケーションを更新します。例: ALTER PUBLICATION <code><publication_name> SET TABLE <tbl1, tbl2, tbl3></code>。</p>

プロパティ	デフォルト	説明
binary.handling.mode	bytes	<p>変更イベントでのバイナリー (bytea) 列の表現方法を指定します。</p> <p>bytes はバイナリーデータをバイト配列として表します。</p> <p>base64 は、base64 でエンコードされた文字列としてバイナリーデータを表します。</p> <p>base64-url-safe は、base64-url-safe でエンコードした文字列としてバイナリーデータを表します。</p> <p>hex は、バイナリーデータを 16 進数でエンコードした (base16) 文字列として表します。</p>
schema.name.adjustment.mode	none	<p>コネクタで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能:</p> <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。
money.fraction.digits	2	<p>Postgres の money タイプを、変更イベントの値を表す java.math.BigDecimal に変換する際に、何桁の 10 進数を使用するかを指定します。decimal.handling.mode が precise に設定されている場合のみ適用されます。</p>

プロパティ	デフォルト	説明
message.prefix.include.list	デフォルトなし	<p>コネクターでキャプチャーする論理デコードメッセージの接頭辞と一致するコンマ区切りの正規表現 (任意)。デフォルトでは、コネクターはすべての論理デコードメッセージをキャプチャーします。このプロパティが設定されている場合、コネクターはプロパティで指定された接頭辞が含まれる論理デコードメッセージのみをキャプチャーします。その他の論理デコードメッセージはすべて除外されます。</p> <p>メッセージの接頭辞名を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、メッセージ接頭辞文字列全体と照合されず。式は、接頭辞に存在する可能性のある部分文字列と一致しません。</p> <p>このプロパティを設定に含める場合は、message.prefix.exclude.list プロパティも設定しないでください。</p> <p>メッセージ イベントの構造とその順序付けのセマンティクスについては、メッセージイベント を参照してください。</p>
message.prefix.exclude.list	デフォルトなし	<p>コネクターでキャプチャーしない論理デコードメッセージの接頭辞名と一致するコンマ区切りの正規表現 (任意)。このプロパティが設定されている場合、コネクターは、指定された接頭部を使用する論理デコードメッセージをキャプチャーしません。その他のメッセージはすべてキャプチャーされます。</p> <p>論理デコードメッセージをすべて除外するには、このプロパティの値を <code>*</code> に設定します。</p> <p>メッセージの接頭辞名を照合するために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、メッセージ接頭辞文字列全体と照合されず。式は、接頭辞に存在する可能性のある部分文字列と一致しません。</p> <p>このプロパティを設定に含める場合は、message.prefix.include.list プロパティも設定しないでください。</p> <p>メッセージ イベントの構造とその順序付けのセマンティクスについては、メッセージイベント を参照してください。</p>

以下の **高度な** 設定プロパティには、ほとんどの状況で機能する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。

表7.26 高度なコネクタ設定プロパティ

プロパティ	デフォルト	説明
<p>converters</p>	<p>デフォルトなし</p>	<p>コネクタが使用できる カスタムコンバーター インスタンスのシンボリック名のコマ区切りリストを列挙します。以下に例を示します。</p> <p>isbn</p> <p>コネクタがカスタムコンバータを使用できるようにするには、converters タプロパティを設定する必要があります。</p> <p>コネクタに設定するコンバーターごとに、コンバーターインターフェイスを実装するクラスの完全修飾名を指定する .type プロパティも追加する必要があります。.type プロパティでは、以下の形式を使用します。</p> <p><converterSymbolicName>.type</p> <p>以下に例を示します。</p> <pre> isbn.type: io.debezium.test.IsbnConverter </pre> <p>設定されたコンバータの動作をさらに制御したい場合は、1つ以上の設定パラメータを追加して、コンバータに値を渡すことができます。追加の設定パラメータとコンバータを関連付けるには、パラメータ名の前にコンバータのシンボリック名を付けます。</p> <p>以下に例を示します。</p> <pre> isbn.schema.name: io.debezium.postgresql.type.Isbn </pre>

プロパティ	デフォルト	説明
<code>snapshot.mode</code>	<code>Initial</code>	<p>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。</p> <p>initial - コネクターは、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のみスナップショットを実行します。</p> <p>always - コネクターはコネクターが開始するたびにスナップショットを実行します。</p> <p>never - コネクターはスナップショットを実行しません。このようにコネクターを設定したすると、起動時の動作は次のようになります。Kafka オフセットトピックに以前保存された LSN がある場合、コネクターはその位置から変更をストリーミングを続行します。保存された LSN がない場合、コネクターはサーバーで PostgreSQL の論理レプリケーションスロットが作成された時点で変更のストリーミングを開始します。never スナップショットモードは、対象のデータがすべて WAL に反映されたままであることが分かっている場合にのみ有効です。</p> <p>initial_only: コネクターは最初のスナップショットを実行し、その後の変更を処理せずに停止します。</p> <p>exported - deprecated</p> <p>詳細は、snapshot.mode オプションのテーブル を参照してください。</p>

プロパティ	デフォルト	説明
snapshot.include.collecton.list	<code>table.include.list</code> に指定したすべてのテーブル	<p>スナップショットに含めるテーブルの完全修飾名 (<code><schemaName>.<tableName></code>) と一致する正規表現のコンマ区切りリスト (オプション) です。指定する項目は、コネクタの table.include.list プロパティで名前を付ける必要があります。このプロパティは、コネクタの snapshot.mode プロパティが never 以外の値に設定されている場合にのみ有効です。</p> <p>このプロパティは増分スナップショットの動作には影響しません。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p>
snapshot.lock.timeout.ms	10000	<p>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。コネクタがこの期間にテーブルロックを取得できないと、スナップショットは失敗します。詳細は、コネクタによるスナップショットの実行方法 を参照してください。</p>

プロパティ	デフォルト	説明
<p>snapshot.select.statement.overrides</p>	<p>デフォルトなし</p>	<p>スナップショットに追加するテーブル行を指定します。スナップショットにテーブルの行のサブセットのみを含める場合は、プロパティを使用します。このプロパティはスナップショットにのみ影響します。コネクターがログから読み取るイベントには影響しません。</p> <p>プロパティには、<schemaName>.<tableName> の形式で完全修飾テーブル名のコンマ区切りリストが含まれます。たとえば、</p> <pre>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</pre> <p>をリスト内の各テーブルに対して、スナップショットを作成する場合には、その他の設定プロパティを追加して、コネクターがテーブルで実行するように SELECT ステートメントを指定します。指定した SELECT ステートメントは、スナップショットに追加するテーブル行のサブセットを決定します。以下の形式を使用して、この SELECT ステートメントプロパティの名前 (</p> <p>snapshot.select.statement.overrides.<schemaName>.<tableName>) を指定します。例:</p> <pre>snapshot.select.statement.overrides. customers.orders.</pre> <p>例:</p> <p>スナップショットにソフト削除以外のレコードのみを含める場合は、soft-delete 列 (delete_flag) を含む customers.orders テーブルから、以下のプロパティを追加します。</p> <pre>"snapshot.select.statement.overrides" : "customer.orders", "snapshot.select.statement.overrides. customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre>

プロパティ	デフォルト	説明
<code>event_processing.failure_handling.mode</code>	<code>fail</code>	<p>作成されるスナップショットでは、コネクタには <code>delete flag = 0</code> のレコードのみイベントの処理中にコネクタが例外に反応する方法を指定します。</p> <p>fail は例外を伝播し、問題のあるイベントのオフセットを示し、コネクタを停止させます。</p> <p>warn は問題のあるイベントのオフセットをログに記録し、そのイベントを省略し、処理を継続します。</p> <p>skip は問題のあるイベントを省略し、処理を継続します。</p>
<code>max.batch.size</code>	2048	コネクタが処理す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。
<code>max.queue.size</code>	8192	ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクタが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクタがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。 max.queue.size の値を、 max.batch.size の値よりも大きくするように設定します。
<code>max.queue.size.in.bytes</code>	0	<p>ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。</p> <p>max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、max.queue.size=1000、max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。</p>

プロパティ	デフォルト	説明
<code>poll.interval.ms</code>	500	コネクターがイベントのバッチの処理を開始する前に、新しい変更イベントの発生を待つ期間をミリ秒単位で指定する正の整数値。デフォルトは 500 ミリ秒です。
<code>include.unknown.datatypes</code>	false	<p>コネクターがデータタイプが不明なフィールドを見つけたときのコネクターの動作を指定します。コネクターが変更イベントからフィールドを省略し、警告をログに記録するのがデフォルトの動作です。</p> <p>変更イベントにフィールドの不透明なバイナリー表現を含める場合は、このプロパティを true に設定します。これにより、コンシューマーはフィールドをデコードできます。binary handling mode プロパティを設定すると、正確な表現を制御できます。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 20px; height: 100px; background: repeating-linear-gradient(45deg, transparent, transparent 2px, black 2px, black 4px); margin-right: 10px;"></div> <div> <p>注記</p> <p>include.unknown.datatypes が true に設定されていると、コンシューマーは後方互換性の問題を抱えることとなります。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、最終的にデータ型が Debezium によってサポートされる場合、データ型は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となります。通常、サポートされていないデータ型が検出された場合は、機能リクエストを作成して、サポートを追加できるようにします。</p> </div> </div>

プロパティ	デフォルト	説明
database.initial.statement s	デフォルトなし	<p>データベースへの JDBC 接続を確立するときにコネクタが実行する SQL ステートメントのセミコロン区切りリスト。セミコロンを区切り文字としてではなく、文字として使用する場合は、2つの連続したセミコロン ;; を指定します。</p> <p>コネクタは JDBC 接続を独自の判断で確立する可能性があります。そのため、このプロパティはセッションパラメーターのみの設定に便利です。また、DML ステートメントの実行には適していません。</p> <p>トランザクションログを読み取るコネクションを作成する場合、コネクタはこれらのステートメントを実行しません。</p>
status.update.interval.ms	10000	<p>レプリケーションの接続状態をサーバーに送信する頻度をミリ秒単位で指定します。また、このプロパティは、データベースがシャットダウンされた場合にデッドコネクションを検出するために、データベースの状態をチェックする頻度を制御します。</p>

プロパティ	デフォルト	説明
heartbeat.interval.ms	0	<p>コネクターがハートビートメッセージを Kafka トピックに送信する頻度を制御します。デフォルトの動作では、コネクターはハートビートメッセージを送信しません。</p> <p>ハートビートメッセージは、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のに便利です。ハートビートメッセージは、コネクターの再起動時に再送信する必要がある変更イベントの数を減らすのに役立つ可能性があります。ハートビートメッセージを送信するには、このプロパティを、ハートビートメッセージの間隔をミリ秒単位で示す正の整数に設定します。</p> <p>追跡されるデータベースに多くの更新がある場合にハートビートメッセージが必要になりますが、一部の更新のみがコネクターの変更をキャプチャーするテーブルおよびスキーマに関連します。この場合、コネクターは通常どおりにデータバーストランザクションログから読み取りしますが、変更レコードを Kafka に出力することはほとんどありません。つまり、オフセットの更新は Kafka にコミットされず、コネクターには最新の LSN をデータベースに送信する機会はありません。データベースは、コネクターによってすでに処理されたイベントが含まれる WAL ファイルを保持します。ハートビートメッセージを送信すると、コネクターは最新の取得された LSN をデータベースに送信できます。これにより、データベースは不必要になった WAL ファイルによって使用されるディスク領域を解放できます。</p>

プロパティ	デフォルト	説明
heartbeat.action.query	デフォルトなし	<p>コネクタがハートビートメッセージを送信するときにコネクタがソースデータベースで実行するクエリを指定します。</p> <p>これは、Debezium WAL ディスク領域の消費を管理するための PostgreSQL の設定で説明されている状況を解決するのに役立ちます。この場合、トラフィックの多いデータベースと同じホストにあるトラフィックが少ないデータベースから変更をキャプチャすることで、Debezium が WAL レコードを処理しないようにし、よってデータベースで WAL の位置を受け入れます。この状況に対処するには、トラフィックの少ないデータベースでハートビートテーブルを作成し、このプロパティをそのテーブルにレコードを挿入するステートメントに設定します (例:</p> <pre>INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')</pre> <p>)。これにより、コネクタはトラフィックの少ないデータベースから変更を受信し、LSN を受け入れでき、データベースホストでバインドされていない WAL が増加しないようにします。</p>
schema.refresh.mode	columns_diff	<p>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。</p> <p>columns_diff は最も安全なモードです。インメモリースキーマがデータベーステーブルの水一まと常に同期されるようになります。</p> <p>columns_diff_exclude_unchanged_toast は、未変更の TOASTable データのみが不一致の原因である場合を除き、受信メッセージから派生するスキーマに不一致があれば、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に指示します。</p> <p>この設定は、ほとんど更新の対象とならない TOASTed データが頻繁に更新されるテーブルがある場合に、コネクタのパフォーマンスを大幅に向上できます。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。</p>

プロパティ	デフォルト	説明
snapshot.delay.ms	デフォルトなし	コネクターの起動時にスナップショットを実行するまでコネクターが待つ必要がある間隔 (ミリ秒単位)。クラスターで複数のコネクターを起動する場合、このプロパティは、コネクターのリバランスが行われる原因となるスナップショットの中断を防ぐのに役立ちます。
snapshot.fetch.size	10240	スナップショットの実行中、コネクターは行のバッチでテーブルの内容を読み取ります。このプロパティは、バッチの行の最大数を指定します。
slot.stream.params	デフォルトなし	設定された論理デコードプラグインに渡すパラメーターのセミコロン区切りリスト。例えば、 add-tables=public.table,public.table2;include-lsn=true のようにします。
sanitize.field.names	コネクターが key.converter または value.converter プロパティを Avro コンバーターに設定する場合は true に設定します。 そうでない場合は false に設定します。	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を示します。
slot.max.retries	6	レプリケーションスロットへの接続に失敗した場合に、連続して接続を試行する最大回数です。
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を行う間隔 (ミリ秒単位)。
unavailable.value.placeholder	<code>__debezium_unavailable_value</code>	コネクターが提供する定数を指定して、元の値がデータベースによって提供されていない Toast 化された値であることを示します。 unavailable.value.placeholder の設定が 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を表すことが想定されます。詳細は、 toast 化された値 を参照してください。

プロパティ	デフォルト	説明
provide.transaction.meta.data	<code>false</code>	コネクタがトランザクション境界でイベントを生成し、トランザクションメタデータで変更イベントエンベロープを強化するかどうかを決定します。コネクタにこれを実行させる場合は <code>true</code> を指定します。詳しくは、 トランザクションメタデータ を参照してください。
flush.lsn.source	<code>true</code>	WAL ログを削除できるように、コネクタがソース postgres データベースに、処理されたレコードの LSN をコミットする必要があるかどうかを決定します。コネクタでこれを行わない場合は <code>false</code> を指定します。 <code>false</code> に設定すると、LSN が Debezium によって認識されず、結果として WAL ログが消去されないため、ディスク容量の問題が発生する可能性があることに注意してください。ユーザーは、Debezium 外で LSN を確認する必要があります。
retriable.restart.connector.wait.ms	10000 (10 秒)	再試行可能なエラーが発生した後にコネクタを再起動するまで待機する時間 (ミリ秒単位)。
skipped.operations	<code>t</code>	ストリーミング中にスキップされる oplog 操作のコンマ区切りリスト。挿入/作成は <code>c</code> 、更新は <code>u</code> 、削除は <code>d</code> 、切り捨ては <code>t</code> 、操作をスキップしない場合は <code>none</code> となります。デフォルトでは、切り捨て操作が省略されます。
signal.data.collection	デフォルト値なし	シグナルをコネクタへの送信に使用されるデータコレクションの完全修飾名。コレクション名の指定には、 <code><schemaName>.<tableName></code> の形式を使用します。
incremental.snapshot.chunk.size	1024	増分スナップショットのチャンクの実行中にコネクタがメモリーを取得して読み取る行の最大数。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。

プロパティ	デフォルト	説明
xmin.fetch.interval.ms	0	レプリケーションスロットから XMIN が読み込まれる頻度 (ミリ秒単位)。XMIN 値は、新しいレプリケーションスロットの開始位置の下限を示す。デフォルト値の 0 は、XMIN の追跡を無効にします。
topic.naming.strategy	<code>io.debezium.schema.SchemaTopicNamingStrategy</code>	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する TopicNamingStrategy クラスの名前。デフォルトは SchemaTopicNamingStrategy 。
topic.delimiter	.	トピック名の区切り文字を指定します。デフォルトは . です。
topic.cache.size	10000	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。
topic.heartbeat.prefix	<code>__debezium-heartbeat</code>	コネクターがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は __debezium-heartbeat.fulfillment になります。
topic.transaction	<code>transaction</code>	コネクターがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、 <code>topic.prefix.topic.transaction</code> です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は fulfillment.transaction になります。

パススルーコネクター設定プロパティ

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 コネクタは [新しいコンシューマー設定プロパティ](#) を使用します。

7.7. DEBEZIUM POSTGRES SQL コネクタのパフォーマンスの監視

Debezium PostgreSQL コネクタは、Zookeeper、Kafka、および Kafka Connect によって提供される JMX メトリクスの組み込みサポートに加えて、2 種類のメトリクスを提供します。

- [スナップショットメトリクス](#) は、スナップショットの実行中にコネクタ操作に関する情報を提供します。
- [メトリクスのストリーミング](#) は、コネクタが変更をキャプチャーし、変更イベントレコードをストリーミングする際のコネクタ操作に関する情報を提供します。

[Debezium モニタリングのドキュメント](#) では、JMX を使用してこれらのメトリクスを公開する方法の詳細を説明しています。

7.7.1. PostgreSQL データベースのスナップショット作成時の Debezium の監視

MBean は `debezium.postgres:type=connector-metrics,context=snapshot,server=<postgresql.server.name>` です。

スナップショット操作がアクティブでない場合や、最後のコネクタの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	タイプ	説明
LastEvent	string	コネクタが読み取りした最後のスナップショットイベント。
MillisecondsSinceLastEvent	long	コネクタが最新のイベントを読み取りおよび処理してから経過時間 (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で確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクタに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクタによって取得されるテーブルの一覧。

属性	タイプ	説明
QueueTotalCapacity	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。
QueueRemainingCapacity	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。
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。

属性	タイプ	説明
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは、 max.queue.size.in.bytes が正の長さの値に設定されている場合に利用できます。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

コネクタは、増分スナップショットの実行時に、以下の追加のスナップショットメトリクスも提供します。

属性	タイプ	説明
ChunkId	string	現在のスナップショットチャンクの識別子。
ChunkFrom	string	現在のチャンクを定義するプライマリーキーセットの下限。
ChunkTo	string	現在のチャンクを定義するプライマリーキーセットの上限。
TableFrom	string	現在スナップショットされているテーブルのプライマリーキーセットの下限。
TableTo	string	現在スナップショットされているテーブルのプライマリーキーセットの上限。

7.7.2. Debezium PostgreSQL コネクタレコードストリーミングの監視

MBean は `debezium.postgres:type=connector-metrics,context=streaming,server=<postgresql.server.name>` です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。
TotalNumberOfCreateEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクターが見た更新イベントの合計数。
TotalNumberOfDeleteEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。
Connected	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。

属性	タイプ	説明
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との間の期間 (ミリ秒単位)。この値は、データベースサーバーとコネクタが稼働しているマシンのクロック間の差異に対応します。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは、 max.queue.size.in.bytes が正の長さの値に設定されている場合に利用できます。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

7.8. DEBEZIUM POSTGRESQL コネクタによる障害および問題の処理方法

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 は変更イベントレコードごとに **1度だけ** 配信します。

障害が発生しても、システムはイベントを失いません。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があります。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。

詳細は以下を参照してください。

- [設定および起動エラー](#)
- [PostgreSQL が使用不可能になる](#)
- [クラスタの障害](#)
- [Kafka Connect のプロセスは正常に停止する](#)
- [Kafka Connect プロセスのクラッシュ](#)

- Kafka が使用不可能になる
- コネクターの一定期間の停止

設定および起動エラー

以下の状況では、起動時にコネクターが失敗し、エラーまたは例外がログに記録され、実行が停止されます。

- 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 PostgreSQL に接続できない。
- コネクターは (LSN を使用して) PostgreSQL WAL の以前に記録された位置から再起動され、PostgreSQL ではその履歴が利用できなくなります。

このような場合、エラーメッセージには問題の詳細が含まれ、推奨される回避策も含まれることがあります。設定の修正したり、PostgreSQL の問題に対処した後、コネクターを再起動します。

PostgreSQL が使用不可能になる

コネクターの実行中、接続先の PostgreSQL サーバーが、さまざまな理由で使用できなくなる可能性があります。この場合、コネクターはエラーで失敗し、停止します。サーバーが再び使用できるようになったら、コネクターを再起動します。

PostgreSQL コネクターは、最後に処理されたオフセットを PostgreSQL LSN の形式で外部に保存します。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、コネクターはサーバーと通信し、その特定のオフセットからストリーミングを続行します。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の状態である限り利用できます。プライマリーサーバーでレプリケーションスロットを削除しないでください。削除する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例は、次のセクションを参照してください。

クラスターの障害

PostgreSQL はリリース 12 より、**プライマリーサーバー上でのみ**論理レプリケーションスロットを許可するようになりました。つまり、Debezium PostgreSQL コネクターをデータベースクラスターのアクティブなプライマリーサーバーのみにポイントできます。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。プライマリーサーバーがダウンした場合は、新しいプライマリーを昇格する必要があります。



注記

一部の管理対象 PostgreSQL サービス (AWS RDS や GCP CloudSQL など) は、ディスクを複製して、スタンバイへのレプリケーションを実装します。つまり、レプリケーションスロットがレプリケートされ、フェイルオーバー後も引き続き使用できます。

新しいプライマリーには、**pgoutput** プラグインが使用するよう設定されたレプリケーションスロットと、変更をキャプチャーするデータベースが必要です。その後でのみ、コネクターが新しいサーバーを示すようにし、コネクターを再起動することができます。

フェイルオーバーが発生した場合は重要な注意点があります。レプリケーションスロットがそのままの状態、データを損失してい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に以下を行います。

- アプリケーションが**新しい**プライマリーに書き込みする前に、Debezium のレプリケーションスロットを再作成するプロセスが必要です。これは重要です。このプロセスがないと、アプリケーションが変更イベントを見逃す可能性があります。

- **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りできることを確認する必要があることがあります。

失われた変更があるかどうかを確認し、取り戻すための信頼できる方法の1つは、障害が発生したプライマリバックアップを、障害が発生する直前まで復旧することです。これは管理が難しい場合がありますが、レプリケーションスロットで未使用の変更があるかどうかを確認することができます。

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した場合を想定します。Kafka Connect はそのプロセスをシャットダウンする前に、プロセスのコネクタタスクをそのグループの別の Kafka Connect プロセスに移行します。新しいコネクタタスクは、以前のタスクが停止した場所でプロセスを開始します。コネクタ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。

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、Kafka Connect は他のプロセスでこれらのコネクタタスクを再起動します。ただし、PostgreSQL コネクタは、以前のプロセスで最後に記録された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。

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重複されたイベントがある可能性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。

各変更イベントレコードでは Debezium コネクタは、イベント発生時の PostgreSQL サーバー時間、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、イベント発生元に関するソース固有の情報を挿入します。コンシューマーは、LSN を重点としてこの情報を追跡し、イベントが重複しているかどうかを判断します。

Kafka が使用不可能になる

変更イベントはコネクタ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。Kafka Connect は、Kafka Connect 設定で指定した頻度で、これらの変更イベントにある最新のオフセットを記録します。Kafka ブローカーが利用できなくなった場合、コネクタを実行している Kafka Connect プロセスは Kafka ブローカーへの再接続を繰り返し試みます。つまり、コネクタ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、コネクタは停止した場所から再開します。

コネクタの一定期間の停止

コネクタが正常に停止された場合、データベースを引き続き使用できます。変更はすべて PostgreSQL WAL に記録されます。コネクタが再起動すると、停止した場所で変更のストリーミングが再開されます。つまり、コネクタが停止した間に発生したデータベースのすべての変更に対して変更イベントレコードが生成されます。

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に従って作成され、十分なリソースがあれば Kafka Connect コネクタも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、Debezium コネクタがしばらく停止した後、再起動すると、停止中に発生したデータベースの変更に対して処理の遅れを取り戻す可能性が非常に高くなります。遅れを取り戻すのに掛かる時間は、Kafka の機能やパフォーマンス、および PostgreSQL のデータに加えられた変更の量によって異なります。

第8章 SQL SERVER の DEBEZIUM コネクタ

Debezium の SQL Server コネクタは、SQL Server データベースのスキーマで発生する行レベルの変更をキャプチャーします。

このコネクタと互換性のある SQL Server のバージョンについては、[Debezium](#) でサポートされる設定ページを参照してください。

Debezium SQL Server コネクタとその使用に関する詳細は、以下を参照してください。

- [「Debezium SQL Server コネクタの概要」](#)
- [「Debezium SQL Server コネクタの仕組み」](#)
- [「Debezium SQL Server コネクタのデータ変更イベントの説明」](#)
- [「Debezium SQL Server コネクタによるデータ型のマッピング方法」](#)
- [「Debezium コネクタを実行するための SQL Server のセットアップ」](#)
- [「Debezium SQL Server コネクタのデプロイ」](#)
- [「スキーマ変更後のキャプチャーテーブルの更新」](#)
- [「Debezium SQL Server コネクタのパフォーマンスの監視」](#)

Debezium SQL Server コネクタが SQL Server データベースまたはクラスターに初めて接続すると、データベースのスキーマの整合性スナップショットが作成されます。コネクタは、最初のスナップショットが完了すると、CDC に対して有効になっている SQL Server データベースにコミットされた **INSERT**、**UPDATE** または **DELETE** 操作の行レベルの変更を継続的にキャプチャーします。コネクタは、各データ変更操作のイベントを生成し、それらのイベントを Kafka トピックにストリーミングします。コネクタは、テーブルのすべてのイベントを専用の Kafka トピックにストリーミングします。その後、アプリケーションとサービスは、そのトピックからのデータ変更イベントレコードを使用できます。

8.1. DEBEZIUM SQL SERVER コネクタの概要

Debezium SQL Server コネクタは、[SQL Server 2016 Service Pack 1 \(SP1\) およびそれ以降](#) の Standard エディションまたは Enterprise エディションで利用可能な **変更データキャプチャー (CDC)** 機能に基づいています。SQL Server のキャプチャープロセスでは、指定のデータベースおよびテーブルを監視し、その変更をストアードプロシージャファサードのある特別に作成された **変更テーブル** に格納します。

Debezium SQL Server コネクタがデータベース操作の変更イベントレコードをキャプチャーできるようにするには、最初に SQL Server データベースで変更データキャプチャー (CDC) を有効にする必要があります。データベースと、キャプチャーする各テーブルの両方で、CDC を有効にする必要があります。ソースデータベースで CDC を設定した後、コネクタはデータベースで発生する行レベルの **INSERT**、**UPDATE** および **DELETE** 操作をキャプチャーできます。コネクタは、各ソーステーブルの各レコードを、そのテーブル専用の Kafka トピックに書き込みます。キャプチャーされたテーブルごとに1つのトピックが存在します。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の Kafka トピックを読み取り、これらのトピックから使用する行レベルのイベントに対応できます。

コネクタが SQL Server データベースまたはクラスターに初めて接続すると、変更をキャプチャーするように設定されたすべてのテーブルのスキーマの整合性スナップショットを作成し、この状態を Kafka にストリーミングします。スナップショットの完了後、コネクタは発生する後続の行レベルの

変更を継続的にキャプチャーします。最初にすべてのデータの整合性のあるビューを確立することで、コネクタはスナップショットの実行中に行われた変更を失うことなく読み取りを続行します。

Debezium SQL Server コネクタはフォールトトレラントです。コネクタは変更を読み取り、イベントを生成するため、データベースログにイベントの位置を定期的に記録します (**LSN / Log Sequence Number**)。コネクタ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)、コネクタは再起動後に最後に読み取りした場所から SQL Server **CDC** テーブルの読み取りを再開します。



注記

オフセットは定期的にコミットされます。変更イベントの発生時にはコミットされません。その結果、停止後に重複するイベントが生成される可能性があります。

フォールトトレランスはスナップショットにも適用されます。つまり、スナップショット中にコネクタが停止した場合、コネクタは再起動時に新しいスナップショットを開始します。

8.2. DEBEZIUM SQL SERVER コネクタの仕組み

Debezium SQL Server コネクタを最適に設定および実行するには、コネクタによるスナップショットの実行方法、変更イベントのストリーム方法、Kafka トピック名の決定方法、およびメタデータの使用方法を理解すると便利です。

コネクタの仕組みに関する詳細は、以下のセクションを参照してください。

- [「Debezium SQL Sever コネクタによるデータベーススナップショットの実行方法」](#)
- [「Debezium SQL Server コネクタによる変更データテーブルの読み取り方法」](#)
- [「Debezium SQL Server 変更イベントレコードを受信する Kafka トピックのデフォルト名」](#)
- [「Debezium SQL Server コネクタによるスキーマ変更トピックの使用方法」](#)
- [「Debezium SQL Server コネクタのデータ変更イベントの説明」](#)
- [「トランザクション境界を表す Debezium SQL Server コネクタによって生成されたイベント」](#)

8.2.1. Debezium SQL Sever コネクタによるデータベーススナップショットの実行方法

SQL Server CDC は、データベースの変更履歴を完全に保存するようには設計されていません。Debezium SQL Server コネクタでデータベースの現在の状態のベースラインを確立するためには、**snapshotting** と呼ばれるプロセスを使用します。

コネクタによるスナップショットの作成方法を設定できます。デフォルトでは、コネクタのスナップショットモードは **initial** に設定されます。この **initial** スナップショットモードを基にして、コネクタ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。この初期スナップショットは、コネクタ用に設定された **include** プロパティおよび **exclude** プロパティ (**table.include.list**、**column.include.list**、**table.exclude.list** など) で定義された基準に一致するテーブルの構造とデータをキャプチャします。

コネクタがスナップショットを作成すると、以下のタスクを完了します。

1. キャプチャーするテーブルを決定します。

2. スナップショットの作成時に構造が変更されないように、CDC が有効になっている SQL Server テーブルのロックを取得します。ロックのレベルは、**snapshot.isolation.mode** 設定プロパティ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での最大ログシーケンス番号 (LSN) の位置を読み取ります。
4. 関連するテーブルすべての構造をキャプチャーします。
5. 必要な場合は、ステップ 2 で取得したロックを解放します。ほとんどの場合、ロックは短期間のみ保持されます。
6. ステップ 3 で読み込まれた LSN の位置に基づいてキャプチャーする SQL Server ソーステーブルとスキーマをスキャンし、テーブルの各行の **READ** イベントを生成して、そのテーブルの Kafka トピックにイベントを書き込みます。
7. コネクタ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。

作成された最初のスナップショットは、CDC に対して有効になっているテーブルの各行の現在の状態をキャプチャーします。このベースライン状態から、コネクタは発生した後続の変更をキャプチャーします。

8.2.1.1. アドホックスナップショット

デフォルトでは、コネクタは初回スナップショット操作の開始後にのみ実行されます。通常の場合では、この最初のスナップショットが作成されると、コネクタではスナップショットプロセスは繰り返し処理されません。コネクタがキャプチャーする今後の変更イベントデータはストリーミングプロセス経由でのみ行われます。

ただし、場合によっては、最初のスナップショット中にコネクタを取得したデータが古くなったり、失われたり、または不完全となったり可能性があります。テーブルデータを再キャプチャーするメカニズムを提供するため、Debezium にはアドホックスナップショットを実行するオプションがあります。データベースで以下が変更されたことで、アドホックスナップショットが実行される場合があります。

- コネクタ設定は、異なるテーブルセットをキャプチャーするように変更されます。
- Kafka トピックを削除して、再構築する必要があります。
- 設定エラーや他の問題が原因で、データの破損が発生します。

アドホックと呼ばれるスナップショットを開始することで、以前にスナップショットをキャプチャーしたテーブルのスナップショットを再実行できます。アドホックスナップショットには、**シグナルテーブル**を使用する必要があります。シグナルリクエストを Debezium シグナルテーブルに送信して、アドホックスナップショットを開始します。

既存のテーブルのアドホックスナップショットを開始すると、コネクタはテーブルにすでに存在するトピックにコンテンツを追加します。既存のトピックが削除された場合には、**トピックの自動作成**が有効になっているのであれば、Debezium は自動的にトピックを作成できます。

アドホックのスナップショットシグナルは、スナップショットに追加するテーブルを指定します。スナップショットは、データベースの内容全体をキャプチャーしたり、データベース内のテーブルのサブセットのみをキャプチャーしたりできます。また、スナップショットは、データベース内のテーブルの内容のサブセットをキャプチャできます。

execute-snapshot メッセージをシグナルテーブルに送信してキャプチャーするテーブルを指定します。以下の表で説明されているように、**run-snapshot** シグナルのタイプを **incremental** に設定し、スナップショットに追加するテーブルの名前を指定します。

表8.1 アドホックのexecute-snapshotシグナルレコードの例

フィールド	デフォルト	値
type	incremental	実行するスナップショットのタイプを指定します。タイプの設定は任意です。現在要求できるのは、 incremental スナップショットのみです。
data-collections	該当なし	スナップショットされるテーブルの完全修飾名にマッチする正規表現を含む配列。名前の形式は signal.data.collection 設定オプションと同じです。
additional-condition	該当なし	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。

アドホックスナップショットのトリガー

execute-snapshot シグナルタイプのエントリーをシグナルテーブルに追加して、アドホックスナップショットを開始します。コネクタがメッセージを処理した後に、スナップショット操作を開始します。スナップショットプロセスは、最初と最後のプライマリーキーの値を読み取り、これらの値を各テーブルの開始ポイントおよびエンドポイントとして使用します。テーブルのエントリー数と設定されたチャンクサイズに基づいて、Debezium はテーブルをチャンクに分割し、チャンクごとに1度に1つずつスナップショットを順番に作成していきます。

現在、**execute-snapshot** アクションタイプは [増分スナップショット](#) のみをトリガーします。詳細は、[スナップショットの増分](#)を参照してください。

8.2.1.2. 増分スナップショット

スナップショットを柔軟に管理するため、Debezium には **増分スナップショット** と呼ばれる補助スナップショットメカニズムが含まれています。増分スナップショットは、[Debezium コネクタにシグナルを送信するための Debezium メカニズム](#)に依存します。

増分スナップショットでは、最初のスナップショットのように、データベースの完全な状態を一度にすべてキャプチャーする代わりに、一連の設定可能なチャンクで各テーブルを段階的にキャプチャーします。スナップショットがキャプチャーするテーブルと、[各チャンクのサイズ](#)を指定できます。チャンクのサイズにより、データベース上の各フェッチ操作中にスナップショットで収集される行数が決まります。増分スナップショットのデフォルトのチャンクサイズは1KBです。

増分スナップショットが進むと、Debezium はウォーターマークを使用して進捗を追跡し、キャプチャーする各テーブル行のレコードを管理します。この段階的なアプローチでは、標準の初期スナップショットプロセスと比較して、以下の利点があります。

- スナップショットが完了するまで、ストリーミングストリーミングを延期する代わりに、ストリーミングしたデータキャプチャーと並行して増分スナップショットを実行できます。コネクタはスナップショットプロセス全体で変更ログからのほぼリアルタイムイベントをキャプチャーし続け、他の操作はブロックしません。
- 増分スナップショットの進捗が中断された場合は、データを失うことなく再開できます。プロセスが再開すると、スナップショットは最初からテーブルをキャプチャーするのではなく、停止した時点から開始します。

- いつでも増分スナップショットを実行し、必要に応じてプロセスを繰り返してデータベースの更新に適合できます。たとえば、コネクター設定を変更してテーブルを `table.include.list` プロパティに追加した後にスナップショットを再実行します。

増分スナップショットプロセス

増分スナップショットを実行する場合には、Debezium は各テーブルをプライマリーキー別に分類して、**設定されたチャンクサイズ**に基づいてテーブルをチャンクに分割します。チャンクごとに作業し、テーブルの行ごとにチャンクでキャプチャーします。キャプチャーする行ごとに、スナップショットは **READ** イベントを出力します。そのイベントは、対象となるチャンクのスナップショットを開始する時の行の値を表します。

スナップショットの作成が進むにつれ、他のプロセスがデータベースへのアクセスを継続し、テーブルレコードが変更される可能性があります。このような変更を反映させるように、通常通りに **INSERT**、**UPDATE**、**DELETE** 操作がトランザクションログにコミットされます。同様に、継続中の Debezium ストリーミングプロセスは、これらの変更イベントを検出し、対応する変更イベントレコードを Kafka に出力します。

Debezium を使用してプライマリーキーが同じレコード間での競合を解決する方法

場合によっては、ストリーミングプロセスが出力する **UPDATE** または **DELETE** イベントを順番に受信できません。つまり、ストリーミングプロセスは、スナップショットがその行の **READ** イベントが含まれるチャンクをキャプチャーする前に、テーブルの行を変更するイベントを生成する可能性があります。スナップショットが最終的に対象の行にあった **READ** イベントを出力すると、その値はすでに置き換えられています。Debezium は、シーケンスが到達する増分スナップショットイベントが正しい論理順序で処理されるように、競合を解決するためにバッファースキームを使用します。スナップショットのイベント間で競合が発生し、ストリーミングされたイベントが解決されてからでないと、Debezium はイベントのレコードを Kafka に送信しません。

スナップショットウィンドウ

遅れて入ってきた **READ** イベントと、同じテーブルの行を変更するストリーミングイベント間の競合の解決を容易にするために、Debezium は **スナップショットウィンドウ** と呼ばれるものを使用します。スナップショットウィンドウは、増分スナップショットが指定のテーブルチャンクのデータをキャプチャーしている途中に、間隔を決定します。チャンクのスナップショットウィンドウを開く前に、Debezium は通常の動作に従い、トランザクションログから直接ターゲットの Kafka トピックにイベントをダウンストリームに出力します。ただし、特定のチャンクのスナップショットが開放された瞬間から終了するまで、Debezium は重複除去のステップを実行して、プライマリーキーが同じイベント間での競合を解決します。

データコレクションごとに、Debezium は 2 種類のイベントを出力し、それらの両方のレコードを単一の宛先 Kafka トピックに保存します。テーブルから直接キャプチャーするスナップショットレコードは、**READ** 操作として出力されます。その間、ユーザーはデータコレクションのレコードの更新を続け、各コミットを反映するようにトランザクションログが更新されるので、Debezium は変更ごとに **UPDATE** または **DELETE** 操作を出力します。

スナップショットウィンドウが開放され、Debezium がスナップショットチャンクの処理を開始すると、スナップショットレコードをメモリーバッファに提供します。スナップショットウィンドウ中に、バッファ内の **READ** イベントのプライマリーキーは、受信ストリームイベントのプライマリーキーと比較されます。一致するものが見つからない場合、ストリーミングされたイベントレコードが Kafka に直接送信されます。Debezium が一致を検出すると、バッファされた **READ** イベントを破棄し、ストリーミングされたレコードを宛先トピックに書き込みます。これは、ストリーミングされたイベントが静的スナップショットイベントよりも論理的に優先されるためです。チャンクのスナップショットウィンドウが終了すると、バッファに含まれるのは、関連するトランザクションログイベントが存在しない **READ** イベントのみです。Debezium は、これらの残りの **READ** イベントをテーブルの Kafka トピックに出力します。

コネクターは各スナップショットチャンクにプロセスを繰り返します。



警告

SQL Server の Debezium コネクターでは、増分スナップショットの実行中のスキーマの変更はサポートしません。

8.2.1.3. 増分スナップショットのトリガー

現在、増分スナップショットを開始する唯一の方法は、[アドホックスナップショットシグナル](#) をソースデータベースのシグナルテーブルに送信することです。

シグナルを SQL **INSERT** クエリーとしてシグナルテーブルに送信します。

Debezium がシグナルテーブルの変更を検出すると、シグナルを読み取り、要求されたスナップショット操作を実行します。

送信するクエリーはスナップショットに追加するテーブルを指定し、必要に応じてスナップショット操作の種類を指定します。現在、スナップショット操作で唯一の有効なオプションはデフォルト値の **incremental** だけです。

スナップショットに追加するテーブルを指定するには、テーブルをリストする **data-collections** 配列またはテーブルの照合に使用する正規表現の配列を指定します。以下に例を示します。

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

増分スナップショットシグナルの **data-collections** アレイにはデフォルト値がありません。**data-collections** アレイが空である場合には、アクションが不要であり、スナップショットを実行しないことが、Debezium で検出されます。



注記

スナップショットに含めるテーブルの名前に、データベース、スキーマ、またはテーブルの名前にドット (.) が含まれている場合、そのテーブルを **data-collections** 配列に追加するには、名前の各部分を二重引用符でエスケープする必要があります。

たとえば、以下のようなテーブルを含めるには **public** スキーマに存在し、その名前が **My.Table** のテーブルを含めるには、**"public"."My.Table"** の形式を使用します。

前提条件

- [シグナルが有効になっている](#)。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは [signal.data.collection](#) プロパティーで指定されます。

手順

1. SQL クエリーを送信し、アドホック増分スナップショット要求をシグナルテーブルに追加します。


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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) 1
values ('ad-hoc-1', 2
       'execute-snapshot', 3
       '{"data-collections": ["schema1.table1", "schema2.table2"],' 4
       "type":"incremental", 5
       "additional-condition":"color=blue"}'); 6
```

コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表8.2 シグナルテーブルに増分スナップショットシグナルを送信する SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリへのログメッセージを特定します。Debezium はこの文字列を使用しません。代わりに、スナップショット作成中に、Debezium は独自の ID 文字列をウォーターマークシグナルとして生成します。
3	execute-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドの必須コンポーネントで、スナップショットに含めるテーブル名の配列またはテーブル名と一致する正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクターのシグナリングテーブル名を指定するのと同じ形式を使用します。
5	incremental	実行するスナップショット操作の種類指定するシグナルの data フィールドの任意の type コンポーネント。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 値を指定しない場合には、コネクターは増分スナップショットを実行します。

項目	値	説明
6	additional-condition	テーブルの内容のサブセットを取得するために、テーブルの列に基づいて条件を指定するオプションの文字列。 additional-condition パラメーターの詳細は、 additional-condition 付きのアドホック増分スナップショット を参照してください。

additional-condition 付きのアドホック増分スナップショット

スナップショットに、テーブル内のコンテンツのサブセットのみを含める場合は、スナップショットシグナルシグナルに **additional-condition** パラメーターを追加してシグナル要求を変更できます。

一般的なスナップショットの SQL クエリーは、以下の形式を取ります。

```
SELECT * FROM <tableName> ....
```

additional-condition パラメーターを追加して、以下の例のように **WHERE** 条件を SQL クエリーに追加します。

```
SELECT * FROM <tableName> WHERE <additional-condition> ....
```

以下の例は、シグナルテーブルに追加の条件を含むアドホック増分スナップショット要求を送信する SQL クエリーを示しています。

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

たとえば、以下の列が含まれる **products** テーブルがあるとします。

- **id** (プライマリーキー)
- **color**
- **quantity**

products テーブルの増分スナップショットに **color=blue** のデータ項目のみを含める場合は、次の SQL ステートメントを使用してスナップショットをトリガーできます。

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

additional-condition パラメーターを使用すると、列が2つ以上となる条件を指定することもできます。たとえば、前述の例の **products** テーブルを使用して、**color=blue** および **quantity>10** だけに一致するアイテムのみのデータが含まれる増分スナップショットをトリガーするクエリーを送信できます。

```
INSERT INTO myschema.debezium_signal (id, type, data) VALUES('ad-hoc-1', 'execute-snapshot', '{"data-collections": ["schema1.products"],"type":"incremental", "additional-condition":"color=blue AND quantity>10"}');
```

以下の例は、コネクターによってキャプチャーされる増分スナップショットイベントの JSON を示しています。

例: 増分スナップショットイベントメッセージ

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

項目	フィールド名	説明
1	snapshot	実行するスナップショット操作タイプを指定します。 現在、唯一の有効なオプションはデフォルト値 incremental だけです。 シグナルテーブルに送信する SQL クエリーでの type 値の指定は任意です。 値を指定しない場合には、コネクターは増分スナップショットを実行します。
2	op	イベントタイプを指定します。 スナップショットイベントの値は r で、 READ 操作を示します。

8.2.1.4. 増分スナップショットの停止

ソースデータベースのテーブルにシグナルを送信して、増分スナップショットを停止することもできます。SQL **INSERT** クエリーを送信して、停止スナップショットシグナルをテーブルに送信します。

Debezium はシグナルテーブルの変更を検出した後、シグナルを読み、増分スナップショット操作が進行中であればそれを停止します。

送信するクエリーは、**incremental** のスナップショット操作を指定し、任意で、削除する実行中のスナップショットのテーブルを指定します。

前提条件

- シグナルが有効になっている。
 - ソースデータベースにシグナリングデータコレクションが存在します。
 - シグナルデータコレクションは **signal.data.collection** プロパティーで指定されます。

手順

1. SQL クエリーを送信して、シグナリングテーブルへのアドホックインクリメンタル スナップショットを停止します。

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

以下に例を示します。

```
INSERT INTO myschema.debezium_signal (id, type, data) ①
values ('ad-hoc-1', ②
       'stop-snapshot', ③
       '{"data-collections": ["schema1.table1", "schema2.table2"], ④
       "type":"incremental"}'); ⑤
```

signal コマンドの **id**、**type**、および **data** パラメーターの値は、シグナルテーブルのフィールドに対応します。

以下の表では、この例のパラメーターについて説明しています。

表8.3 シグナリングテーブルに増分スナップショット停止信号を送信するための SQL コマンドのフィールドの説明

項目	値	説明
1	myschema.debezium_signal	ソースデータベースにあるシグナルテーブルの完全修飾名を指定します。
2	ad-hoc-1	id パラメーターは、シグナルリクエストの ID 識別子として割り当てられる任意の文字列を指定します。 この文字列を使用して、シグナルテーブルのエントリーへのログメッセージを特定します。Debezium はこの文字列を使用しません。
3	stop-snapshot	type パラメーターを指定し、シグナルがトリガーする操作を指定します。
4	data-collections	シグナルの data フィールドのオプションコンポーネントで、スナップショットから削除するテーブル名の配列またはテーブル名とマッチする正規表現を指定します。 この配列は、完全修飾名でテーブルをマッチさせる正規表現をリストアップします。 signal.data.collection 設定プロパティでコネクタのシグナリングテーブル名を指定するのと同じ形式を使用します。 data フィールドのこのコンポーネントを省略すると、シグナルは進行中の増分スナップショット全体を停止します。
5	incremental	停止させるスナップショット操作の種類を指定する信号の data フィールドの必須コンポーネント。 現在、有効な唯一のオプションは incremental です。 type の値を指定しない場合、シグナルは増分スナップショットの停止に失敗します。

8.2.2. Debezium SQL Server コネクタによる変更データテーブルの読み取り方法

コネクタが最初に起動すると、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作成し、その情報を内部データベーススキーマ履歴トピックに対して永続化します。その後、コネクタは各ソーステーブルの変更テーブルを特定し、以下の手順を完了します。

1. コネクタは、変更テーブルごとに、最後に保存された最大 LSN と現在の最大 LSN の間に作成された変更をすべて読み取ります。
2. コネクタは、コミット LSN と変更 LSN の値を基にして、読み取る変更を昇順で並び替えます。この並び替えの順序により、変更はデータベースで発生した順序で Debezium によって再生されるようになります。
3. コネクタは、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. コネクタは最大 LSN を保存し、ステップ1からプロセスを再開します。

再開後、コネクタは読み取った最後のオフセット (コミットおよび変更 LSN) から処理を再開します。

コネクタは、含まれるソーステーブルに対して CDC が有効または無効化されているかどうかを検出し、その動作を調整することができます。

8.2.3. データベースでの最大 LSN の記録なし

次の理由により、最大 LSN がデータベースに記録されない場合があります。

1. SQL Server エージェントが実行されていない
2. 変更テーブルにまだ変更が記録されていない
3. データベースのアクティビティが少なく、cdc クリーンアップジョブで cdc テーブルから定期的にエントリが消去される

これらの可能性のうち、実行中の SQL Server エージェントが前提条件であるため、実際には No 1. は問題です (No 2. と 3. は正常です)。

この問題を軽減し、No 1. と他の問題を区別するために、"**SELECT CASE WHEN dss.[status]=4 THEN 1 ELSE 0 END AS isRunning FROM [#db].sys.dm_server_services dss WHERE dss.[servicename] LIKE N'SQL Server Agent (%)';"** のクエリーを使用して SQL Server エージェントのステータスをチェックします。SQL Server Agent が実行されていない場合に、ログに "No maximum LSN recorded in the database; SQL Server Agent is not running" というエラーが書き込まれます。



重要

ステータスクエリーを実行する SQL Server には、**VIEW SERVER STATE** のサーバーパーミッションが必要です。設定したユーザーにこのパーミッションを付与する必要がない場合は、**database.sqlserver.agent.status.query** プロパティで独自のクエリーを設定できます。SQL Server Agent が実行中 (false または 0) で、[What minimum permissions do I need to provide to a user so that it can check the status of SQL Server Agent Service?](#) または [Safely and Easily Use High-Level Permissions Without Granting Them to Anyone: Server-level](#) で説明されているように、高度なパーミッションを付与せずに安全に使用している場合に、True または 1 を返す関数を定義できます。クエリープロパティの設定は、**database.sqlserver.agent.status.query=SELECT #db.func_is_sql_server_agent_running()** のようになります。**#db** は、データベース名のプレースホルダーとして使用する必要があります。

8.2.4. Debezium SQL Server コネクターの制限事項

SQL Server では、変更キャプチャのインスタンスを作成するために、ベースオブジェクトがテーブルであることが特に必要です。そのため、インデックス付きビュー (別名: マテリアライズドビュー) からの変更の取り込みは、SQL Server ではサポートされておらず、したがって Debezium SQL Server コネクタもサポートされていません。

8.2.5. Debezium SQL Server 変更イベントレコードを受信する Kafka トピックのデフォルト名

デフォルトでは、SQL Server コネクタは、テーブルで発生するすべての **INSERT**、**UPDATE**、**DELETE** 操作のイベントを、そのテーブルに固有の単一の Apache Kafka トピックに書き込みます。コネクタは、**<topicPrefix>.<schemaName>.<tableName>** の規則を使用して変更イベントトピックに名前を付けます。

以下のリストは、デフォルト名のコンポーネントの定義を示しています。

topicPrefix

topic.prefix 設定プロパティで指定したサーバーの論理名です。

schemaName

変更イベントが発生したデータベーススキーマの名前。

tableName

変更イベントが発生したデータベーステーブルの名前。

たとえば、**fulfillment** が論理サーバー名、**dbo** がスキーマ名で、データベースに **products**、**products_on_hand**、**customers**、**orders** という名前のテーブルがある場合、コネクタは変更イベントレコードを次の Kafka トピックにストリーミングします。

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

コネクタは同様の命名規則を適用して、内部データベーススキーマの履歴トピック ([スキーマ変更トピック](#) と [トランザクションメタデータトピック](#)) にラベルを付けます。

デフォルトのトピック名が要件を満たさない場合は、カスタムトピック名を設定できます。カスタムト

ピック名を設定するには、論理トピックルーティング SMT に正規表現を指定します。論理トピックルーティング SMT を使用してトピックの命名をカスタマイズする方法は、[トピックルーティング](#) を参照してください。

8.2.6. Debezium SQL Server コネクタによるスキーマ変更トピックの使用方法

CDC が有効になっている各テーブルについて、Debezium SQL Server コネクタは、データベース内のキャプチャしたテーブルに適用されたスキーマ変更イベントの履歴を保存します。コネクタはスキーマ変更イベントを `<topicPrefix>` という名前の Kafka トピックに書き込みます。ここで、`topicPrefix` は `topic.prefix` 設定プロパティで指定された論理サーバー名です。

コネクタがスキーマ変更トピックに送信するメッセージには、ペイロードと、任意で変更イベントメッセージのスキーマが含まれます。スキーマ変更イベントメッセージのペイロードには、以下の要素が含まれます。

databaseName

ステートメントが適用されるデータベースの名前。`databaseName` の値は、メッセージキーとして機能します。

tableChanges

スキーマの変更後のテーブルスキーマ全体の構造化表現。`tableChanges` フィールドには、テーブルの各列のエントリなどのアレイが含まれます。構造化された表現は JSON または Avro 形式でデータを表示するため、コンシューマーは DDL パーサーを介して最初にメッセージを処理しなくてもメッセージを簡単に読み取りできます。



重要

コネクタがテーブルをキャプチャするように設定されている場合、テーブルのスキーマ変更の履歴は、スキーマ変更トピックだけでなく、内部データベーススキーマの履歴トピックにも格納されます。内部データベーススキーマ履歴トピックはコネクタのみの使用を対象としており、使用するアプリケーションによる直接使用を目的としていません。スキーマ変更に関する通知が必要なアプリケーションが、スキーマ変更トピックからの情報のみを使用するようにしてください。



警告

コネクタがスキーマ変更トピックに出力するメッセージの形式は、初期の状態であり、通知なしに変更される可能性があります。

Debezium は、以下のイベントの発生時にスキーマ変更トピックにメッセージを出力します。

- テーブルの CDC を有効にします。
- テーブルの CDC を無効にします。
- [スキーマの進化手順](#) に従って、CDC が有効になっているテーブルの構造を変更します。

例: SQL Server コネクタのスキーマ変更トピックに送信されるメッセージ

以下の例は、スキーマ変更トピックのメッセージを示しています。メッセージには、テーブルスキーマの論理表現が含まれます。

```
{
  "schema": {
    ...
  },
  "payload": {
    "source": {
      "version": "2.1.4.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 0,
      "snapshot": "true",
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "ts_ms": 1588252618953, ①
    "databaseName": "testDB", ②
    "schemaName": "dbo",
    "ddl": null, ③
    "tableChanges": [ ④
      {
        "type": "CREATE", ⑤
        "id": "\"testDB\".\"dbo\".\"customers\"", ⑥
        "table": { ⑦
          "defaultCharsetName": null,
          "primaryKeyColumnNames": [ ⑧
            "id"
          ],
          "columns": [ ⑨
            {
              "name": "id",
              "jdbcType": 4,
              "nativeType": null,
              "typeName": "int identity",
              "typeExpression": "int identity",
              "charsetName": null,
              "length": 10,
              "scale": 0,
              "position": 1,
              "optional": false,
              "autoIncremented": false,
              "generated": false
            },
            {
              "name": "first_name",
              "jdbcType": 12,
              "nativeType": null,
              "typeName": "varchar",
              "typeExpression": "varchar",
              "charsetName": null,
              "length": 255,
              "scale": null,
```



```

    "position": 2,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "last_name",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "varchar",
    "typeExpression": "varchar",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 3,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  },
  {
    "name": "email",
    "jdbcType": 12,
    "nativeType": null,
    "typeName": "varchar",
    "typeExpression": "varchar",
    "charsetName": null,
    "length": 255,
    "scale": null,
    "position": 4,
    "optional": false,
    "autoIncremented": false,
    "generated": false
  }
],
"attributes": [ 10
  {
    "customAttribute": "attributeValue"
  }
]
}
]
}
}

```

表8.4 スキーマ変更トピックに出力されたメッセージのフィールドの説明

項目	フィールド名	説明
----	--------	----

項目	フィールド名	説明
1	ts_ms	<p>コネクタ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。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。</p> <p>ソースオブジェクトの ts_ms は、データベースで変更が行われた時刻を示す。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>
2	databaseName schemaName	変更が含まれるデータベースとスキーマを識別します。
3	ddl	SQL Server コネクタの場合は常に null です。その他のコネクタでは、このフィールドにスキーマの変更を行う DDL が含まれます。この DDL は SQL Server コネクタでは使用できません。
4	tableChanges	DDL コマンドによって生成されるスキーマの変更が含まれる1つ以上の項目の配列。
5	type	<p>変更の種類を説明します。値は以下のいずれかになります。</p> <ul style="list-style-type: none"> ● CREATE - テーブルの作成 ● ALTER - テーブルの変更 ● DROP - テーブルの削除
6	id	作成、変更、または破棄されたテーブルの完全な識別子。
7	table	適用された変更後のテーブルメタデータを表します。
8	primaryKeyColumnNames	テーブルのプライマリーキーを設定する列のリスト。
9	列	変更されたテーブルの各列のメタデータ。
10	attributes	各テーブル変更のカスタム属性メタデータ。

コネクタがスキーマ変更トピックに送信するメッセージでは、キーはスキーマの変更が含まれるデータベースの名前です。以下の例では、**payload** フィールドにキーが含まれます。

```
{
  "schema": {
```

```

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

```

8.2.7. Debezium SQL Server コネクタのデータ変更イベントの説明

Debezium SQL Server コネクタは、行レベルの **INSERT**、**UPDATE**、および **DELETE** 操作ごとにデータ変更イベントを生成します。各イベントにはキーと値が含まれます。キーと値の構造は、変更されたテーブルによって異なります。

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** を中心として設計されています。ただし、これらのイベントの構造は時間の経過とともに変化する可能性があり、コンシューマーによる処理が困難になることがあります。これに対応するために、各イベントにはコンテンツのスキーマが含まれます。スキーマレジストリーを使用している場合は、コンシューマーがレジストリーからスキーマを取得するために使用できるスキーマ ID が含まれます。これにより、各イベントが自己完結型になります。

以下のスケルトン JSON は、変更イベントの基本となる 4 つの部分を示しています。ただし、アプリケーションで使用するために選択した Kafka Connect コンバータの設定方法によって、変更イベントのこれら 4 部分の表現が決定されます。**schema** フィールドは、変更イベントが生成されるようにコンバータを設定した場合のみ変更イベントに含まれます。同様に、イベントキーおよびイベントペイロードは、変更イベントが生成されるようにコンバータを設定した場合のみ変更イベントに含まれます。JSON コンバータを使用し、変更イベントの基本となる 4 つの部分すべてを生成するように設定すると、変更イベントの構造は次のようになります。

```

{
  "schema": { ❶
    ...
  },
  "payload": { ❷
    ...
  },
  "schema": { ❸
    ...
  },
  "payload": { ❹
    ...
  },
}

```

表8.5 変更イベントの基本内容の概要

項目	フィールド名	説明
1	schema	最初の schema フィールドはイベントキーの一部です。イベントキーの payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、最初の schema フィールドは、変更されたテーブルのプライマリーキーの構造、またはテーブルにプライマリーキーがない場合は変更されたテーブルの一意キーの構造を記述します。 message.key.columns コネクタ設定プロパティを設定すると、テーブルのプライマリーキーをオーバーライドできます。この場合、最初の schema フィールドはそのプロパティによって識別されるキーの構造を記述します。
2	payload	最初の payload フィールドはイベントキーの一部です。前述の schema フィールドによって記述された構造を持ち、変更された行のキーが含まれます。
3	schema	2つ目の schema フィールドはイベント値の一部です。イベント値の payload の部分の内容を記述する Kafka Connect スキーマを指定します。つまり、2つ目の schema は変更された行の構造を記述します。通常、このスキーマには入れ子になったスキーマが含まれます。
4	payload	2つ目の payload フィールドはイベント値の一部です。前述の schema フィールドによって記述された構造を持ち、変更された行の実際のデータが含まれます。

デフォルトでは、コネクタによって、変更イベントレコードがイベントの元のテーブルと同じ名前を持つトピックにストリーミングされます。詳細は、[トピック名](#)を参照してください。



警告

SQL Server コネクタは、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。つまり、論理サーバー名はアルファベットまたはアンダースコア (a-z、A-Z、または _) で始まる必要があります。論理サーバー名の残りの各文字と、データベース名とテーブル名の各文字は、アルファベット、数字、またはアンダースコア (a-z、A-Z、0-9、または _) でなければなりません。無効な文字がある場合は、アンダースコアに置き換えられます。

論理サーバー名、データベース名、またはテーブル名に無効な文字が含まれ、名前を区別する唯一の文字が無効であると、無効な文字はすべてアンダースコアに置き換えられるため、予期せぬ競合が発生する可能性があります。

変更イベントの詳細は、以下を参照してください。

- 「[Debezium SQL Server 変更イベントのキー](#)」
- 「[Debezium SQL Server 変更イベントの値](#)」

8.2.7.1. Debezium SQL Server 変更イベントのキー

変更イベントのキーには、変更されたテーブルのキーのスキーマと、変更された行の実際のキーのスキーマが含まれます。スキーマとそれに対応するペイロードの両方には、コネクターによってイベントが作成された時点において、変更されたテーブルのプライマリーキー (または一意なキー制約) に存在した各列のフィールドが含まれます。

以下の **customers** テーブルについて考えてみましょう。この後に、このテーブルの変更イベントキーの例を示します。

テーブルの例

```
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
);
```

変更イベントキーの例

customers テーブルへの変更をキャプチャーする変更イベントのすべてに、イベントキースキーマがあります。**customers** テーブルに前述の定義がある限り、**customers** テーブルへの変更をキャプチャーする変更イベントのキー構造は、JSON では以下のようになります。

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

表8.6 変更イベントキーの説明

項目	フィールド名	説明
1	schema	キーのスキーマ部分は、キーの payload 部分の内容を記述する Kafka Connect スキーマを指定します。
2	fields	各フィールドの名前、型、および必要かどうかなど、 payload で想定される各フィールドを指定します。この例では、型が int32 の id という名前の必須フィールドが1つあります。

項目	フィールド名	説明
3	任意	イベントキーの payload フィールドに値が含まれる必要があるかどうかを示します。この例では、キーのペイロードに値が必要です。テーブルにプライマリーキーがない場合は、キーの payload フィールドの値は任意です。
4	server1.dbo.test DB.customers.Key	キーのペイロードの構造を定義するスキーマの名前。このスキーマは、変更されたテーブルのプライマリーキーの構造を記述します。キースキーマ名の形式は connector-name.database-schema-name.table-name.Key です。この例では、以下のようになります。 <ul style="list-style-type: none"> ● server1 はこのイベントを生成したコネクタの名前です。 ● dbo は変更されたテーブルのデータベーススキーマです。 ● customers は更新されたテーブルです。
5	payload	この変更イベントが生成された行のキーが含まれます。この例では、キーには値が 1004 の id フィールドが1つ含まれます。

8.2.7.2. Debezium SQL Server 変更イベントの値

変更イベントの値はキーよりも若干複雑です。キーと同様に、値には **schema** セクションと **payload** セクションがあります。**schema** セクションには、入れ子のフィールドを含む、**Envelope** セクションの **payload** 構造を記述するスキーマが含まれています。データを作成、更新、または削除する操作のすべての変更イベントには、Envelope 構造を持つ値 payload があります。

変更イベントキーの例を紹介するために使用した、同じサンプル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
);
```

このテーブルへの変更に対する変更イベントの値部分には、以下の各イベント型について記述されています。

- [作成イベント](#)
- [更新イベント](#)
- [削除イベント](#)

作成 イベント

以下の例は、**customers** テーブルにデータを作成する操作に対して、コネクタによって生成される変更イベントの値の部分を示しています。

```
{
  "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": "server1.dbo.testDB.customers.Value", 2
    "field": "before"
  },
  {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "id"
      },
      {
        "type": "string",
        "optional": false,
        "field": "first_name"
      },
      {
        "type": "string",
        "optional": false,
        "field": "last_name"
      },
      {
        "type": "string",
        "optional": false,
        "field": "email"
      }
    ],
    "optional": true,
    "name": "server1.dbo.testDB.customers.Value",
```

```
"field": "after"
},
{
  "type": "struct",
  "fields": [
    {
      "type": "string",
      "optional": false,
      "field": "version"
    },
    {
      "type": "string",
      "optional": false,
      "field": "connector"
    },
    {
      "type": "string",
      "optional": false,
      "field": "name"
    },
    {
      "type": "int64",
      "optional": false,
      "field": "ts_ms"
    },
    {
      "type": "boolean",
      "optional": true,
      "default": false,
      "field": "snapshot"
    },
    {
      "type": "string",
      "optional": false,
      "field": "db"
    },
    {
      "type": "string",
      "optional": false,
      "field": "schema"
    },
    {
      "type": "string",
      "optional": false,
      "field": "table"
    },
    {
      "type": "string",
      "optional": true,
      "field": "change_lsn"
    },
    {
      "type": "string",
      "optional": true,
      "field": "commit_lsn"
    },
  ],
}
```



```

    {
      "type": "int64",
      "optional": true,
      "field": "event_serial_no"
    }
  ],
  "optional": false,
  "name": "io.debezium.connector.sqlserver.Source", 3
  "field": "source"
},
{
  "type": "string",
  "optional": false,
  "field": "op"
},
{
  "type": "int64",
  "optional": true,
  "field": "ts_ms"
}
],
"optional": false,
"name": "server1.dbo.testDB.customers.Envelope" 4
},
"payload": { 5
  "before": null, 6
  "after": { 7
    "id": 1005,
    "first_name": "john",
    "last_name": "doe",
    "email": "john.doe@example.org"
  },
  "source": { 8
    "version": "2.1.4.Final",
    "connector": "sqlserver",
    "name": "server1",
    "ts_ms": 1559729468470,
    "snapshot": false,
    "db": "testDB",
    "schema": "dbo",
    "table": "customers",
    "change_lsn": "00000027:00000758:0003",
    "commit_lsn": "00000027:00000758:0005",
    "event_serial_no": "1"
  },
  "op": "c", 9
  "ts_ms": 1559729471739 10
}
}

```

表8.7 作成 イベント値フィールドの説明

項目	フィールド名	説明
1	schema	値のペイロードの構造を記述する、値のスキーマ。変更イベントの値スキーマは、コネクタが特定のテーブルに生成するすべての変更イベントで同じになります。
2	name	スキーマ セクションで、各 name フィールドは、値のペイロードのフィールドのスキーマを指定します。 server1.dbo.testDB.customers.Value はペイロードの before および after フィールドのスキーマです。このスキーマは customers テーブルに固有です。 before および after フィールドのスキーマ名は logicalName.database-schemaName.tableName.Value の形式を取るため、スキーマ名がデータベースで一意になるようにします。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 スキーマは、それぞれ独自に進化し、独自の履歴を持つことになります。
3	name	io.debezium.connector.sqlserver.Source は、ペイロードの source フィールドのスキーマです。このスキーマは、SQL Server コネクタに固有です。コネクタは生成するすべてのイベントにこれを使用します。
4	name	server1.dbo.testDB.customers.Envelope は、ペイロードの全体的な構造のスキーマで、 server1 はコネクタ名、 dbo はデータベーススキーマ名、 customers はテーブルを指します。
5	payload	値の実際のデータ。これは、変更イベントが提供する情報です。 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。これは、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です。しかし、 Avro コンバーター を使用すると、コネクタが Kafka トピックにストリーミングするメッセージのサイズを大幅に小さくすることができます。
6	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。この例のように、 op フィールドが create (作成) の c である場合、この変更イベントは新しい内容に対するものであるため、 before は null になります。
7	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。この例では、 after フィールドには、新しい行の id 、 first_name 、 last_name 、および email 列の値が含まれます。

項目	フィールド名	説明
8	source	<p>イベントのソースメタデータを記述する必須のフィールド。このフィールドには、イベントの発生元、イベントの発生順序、およびイベントが同じトランザクションの一部であるかどうかなど、このイベントと他のイベントを比較するために使用できる情報が含まれています。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクタ型および名前 ● データベースおよびスキーマ名 ● データベースに変更が加えられた時点のタイムスタンプ ● イベントがスナップショットの一部であるか ● 新しい行が含まれるテーブルの名前 ● サーバーログオフセット
9	op	<p>コネクタによってイベントが生成される原因となった操作の型を記述する必須文字列。この例では、c は操作によって行が作成されたことを示しています。有効な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● c = create ● u = update ● d = delete ● r = read (読み取り、スナップショットのみに適用)
10	ts_ms	<p>コネクタ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。イベントメッセージエンベロープでは、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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースにコミットされた時刻を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

更新イベント

サンプル **customers** テーブルにある更新の変更イベントの値には、そのテーブルの **作成** イベントと同じスキーマがあります。同様に、イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。以下は、コネクタによって **customers** テーブルでの更新に生成されるイベントの変更イベント値の例になります。

```
{
  "schema": { ... },
  "payload": {
    "before": { ①
      "id": 1005,
```

```

    "first_name": "john",
    "last_name": "doe",
    "email": "john.doe@example.org"
  },
  "after": { ②
    "id": 1005,
    "first_name": "john",
    "last_name": "doe",
    "email": "noreply@example.org"
  },
  "source": { ③
    "version": "2.1.4.Final",
    "connector": "sqlserver",
    "name": "server1",
    "ts_ms": 1559729995937,
    "snapshot": false,
    "db": "testDB",
    "schema": "dbo",
    "table": "customers",
    "change_lsn": "00000027:00000ac0:0002",
    "commit_lsn": "00000027:00000ac0:0007",
    "event_serial_no": "2"
  },
  "op": "u", ④
  "ts_ms": 1559729998706 ⑤
}
}

```

表8.8 更新 イベント値フィールドの説明

項目	フィールド名	説明
1	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。 更新 イベント値の before フィールドには、各テーブル列のフィールドと、データベースのコミット前にその列にあった値が含まれます。この例では、 email の値は john.doe@example.org です。
2	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。 before と after の構造を比較すると、この行への更新内容を判断できます。この例では、 email の値は noreply@example.org です。

項目	フィールド名	説明
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。source フィールド構造には create イベントと同じフィールドがありますが、一部の値が異なります。たとえば、更新 イベントサンプルのオフセットは異なります。ソースメタデータには以下が含まれています。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベースおよびスキーマ名 ● データベースに変更が加えられた時点のタイムスタンプ ● イベントがスナップショットの一部であるか ● 新しい行が含まれるテーブルの名前 ● サーバーログオフセット <p>event_serial_no フィールドは、同じコミットおよび変更 LSN を持つイベントを区別します。このフィールドの値が 1 以外である場合に典型的な状況です。</p> <ul style="list-style-type: none"> ● 更新によって SQL Server の CDC 変更テーブルに 2 つのイベントが生成されるため、更新 イベントの値は 2 に設定されています (詳細はソースドキュメントを参照してください)。最初のイベントには古い値が含まれ、2 番目のイベントには新しい値が含まれます。コネクターは最初のイベントの値を使用して 2 つ目のイベントを作成します。コネクターは最初のイベントを破棄します。 ● プライマリーキーが更新されると、SQL Server は 2 つのイベントを生成します。古いプライマリーキーを持つレコードを削除するための 削除 イベントと、新しいプライマリーキーを持つレコードを追加するための 作成 イベント。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。イベント番号はそれぞれ 1 および 2 です。
4	op	<p>操作の型を記述する必須の文字列。更新 イベントの値では、op フィールドの値は u で、更新によってこの行が変更したことを示します。</p>
5	ts_ms	<p>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。イベントメッセージエンベロープでは、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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースにコミットされた時刻を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>



注記

行のプライマリーキー/一意キーの列を更新すると、行のキーの値が変更されます。キーが変更されると、3つのイベントが Debezium によって出力されます。3つのイベントとは、**delete** イベント、行に古いキーが含まれる **tombstone** イベント、および行に新しいキーが含まれる **create** イベントを指します。

delete イベント

削除 イベントの値は、同じテーブルの **作成** および **更新** イベントと同じ **schema** の部分になります。サンプル **customers** テーブルの **削除** イベントの **payload** 部分は以下のようになります。

```
{
  "schema": { ... },
},
"payload": {
  "before": { <>
    "id": 1005,
    "first_name": "john",
    "last_name": "doe",
    "email": "noreply@example.org"
  },
  "after": null, ①
  "source": { ②
    "version": "2.1.4.Final",
    "connector": "sqlserver",
    "name": "server1",
    "ts_ms": 1559730445243,
    "snapshot": false,
    "db": "testDB",
    "schema": "dbo",
    "table": "customers",
    "change_lsn": "00000027:00000db0:0005",
    "commit_lsn": "00000027:00000db0:0007",
    "event_serial_no": "1"
  },
  "op": "d", ③
  "ts_ms": 1559730450205 ④
}
}
```

表8.9 削除 イベント値フィールドの説明

項目	フィールド名	説明
1	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。 削除 イベント値の before フィールドには、データベースのコミットで削除される前に行にあった値が含まれます。
2	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。 削除 イベント値の after フィールドは null で、行が存在しないことを示します。

項目	フィールド名	説明
3	source	<p>イベントのソースメタデータを記述する必須のフィールド。削除 イベント値の source フィールド構造は、同じテーブルの作成 および更新 イベントと同じになります。多くの source フィールドの値も同じです。削除 イベント値では、ts_ms および pos フィールドの値や、その他の値が変更された可能性があります。ただし、削除 イベント値の source フィールドは、同じメタデータを提供します。</p> <ul style="list-style-type: none"> ● Debezium バージョン ● コネクター型および名前 ● データベースおよびスキーマ名 ● データベースに変更が加えられた時点のタイムスタンプ ● イベントがスナップショットの一部であるか ● 新しい行が含まれるテーブルの名前 ● サーバログオフセット
4	op	<p>操作の型を記述する必須の文字列。op フィールドの値は d で、行が削除されたことを示します。</p>
5	ts_ms	<p>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。イベントメッセージエンベロープでは、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。</p> <p>source オブジェクトで、ts_ms は変更がデータベースに加えられた時間を示します。payload.source.ts_ms の値を payload.ts_ms の値と比較することにより、ソースデータベースの更新と Debezium との間の遅延を判断できます。</p>

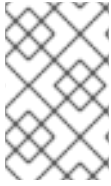
SQL Server コネクターイベントは、[Kafka ログコンパクション](#) と動作するように設計されています。ログコンパクションにより、少なくとも各キーの最新のメッセージが保持される限り、一部の古いメッセージを削除できます。これにより、トピックに完全なデータセットが含まれ、キーベースの状態のリロードに使用できるようにするとともに、Kafka がストレージ領域を確保できるようにします。

廃棄 (tombstone) イベント

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、**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、Kafka が同じキーを持つすべてのメッセージを削除するには、メッセージの値が **null** である必要があります。これを可能にするために、Debezium の SQL Server コネクターは **削除** イベントを出力した後に、**null** 値以外の同じキーを持つ、特別な廃棄 (tombstone) イベントを出力します。

8.2.8. トランザクション境界を表す Debezium SQL Server コネクターによって生成されたイベント

Debezium は、トランザクション境界を表し、データ変更イベントメッセージをエンリッチするイベントを生成できます。



DEBEZIUM がトランザクションメタデータを受信する場合の制限

Debezium は、コネクターのデプロイ後に発生するトランザクションに対してのみメタデータを登録し、受信します。コネクターをデプロイする前に発生するトランザクションのメタデータは利用できません。

データベーストランザクションは、キーワード **BEGIN** および **END** で囲まれたステートメントブロックによって表されます。Debezium は、すべてのトランザクションで **BEGIN** および **END** 区切り文字のトランザクション境界イベントを生成します。トランザクション境界イベントには以下のフィールドが含まれます。

status

BEGIN または **END**

id

一意のトランザクション識別子の文字列表現。

ts_ms

データソースでのトランザクション境界イベント (**BEGIN** または **END** イベント) の時間。もしデータソースが Debezium にイベント時間を提供しないなら、このフィールドは代わりに Debezium がイベントを処理する時間を表します。

event_count (END イベント用)

トランザクションによって出力されるイベントの合計数。

data_collections (END イベント用)

data_collection と **event_count** 要素のペアの配列。これは、コネクターがデータコレクションから発信された変更に対して出力するイベントの数を示します。



警告

Debezium には、トランザクションがいつ終了したかを確実に識別する方法がありません。このように、トランザクション **END** マーカーは、別のトランザクションの最初のイベントが到着した後にのみ発行されます。これにより、トラフィックの少ないシステムの場合、**END** マーカーの配信が遅れる可能性があります。

以下の例は、典型的なトランザクション境界メッセージを示しています。

例: SQL Server コネクタートランザクション境界イベント

```
{
  "status": "BEGIN",
  "id": "00000025:00000d08:0025",
  "ts_ms": 1486500577125,
  "event_count": null,
  "data_collections": null
}
```



```
{
  "status": "END",
  "id": "00000025:00000d08:0025",
  "ts_ms": 1486500577691,
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "testDB.dbo.testDB.tablea",
      "event_count": 1
    },
    {
      "data_collection": "testDB.dbo.testDB.tableb",
      "event_count": 1
    }
  ]
}
```

topic.transaction オプションで上書きされない限り、トランザクションイベントは **<topic.prefix>.transaction** という名前のトピックに書き込まれます。

8.2.8.1. 変更データイベントのエンリッチメント

トランザクションメタデータを有効にすると、データメッセージ **Envelope** は新しい **transaction** フィールドでエンリッチされます。このフィールドは、複合フィールドの形式ですべてのイベントに関する情報を提供します。

id

一意のトランザクション識別子の文字列表現。

total_order

トランザクションによって生成されたすべてのイベントを対象とするイベントの絶対位置。

data_collection_order

トランザクションによって出力されたすべてのイベントを対象とするイベントのデータコレクションごとの位置。

以下の例は、典型的なメッセージの例を示しています。

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

8.2.9. Debezium SQL Server コネクタによるデータ型のマッピング方法

Debezium SQL Server コネクタは、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生成して、テーブル行データへの変更を表します。各イベントには、行の列値を表すフィールドが含まれます。イベントが操作の列値を表す方法は、列の SQL データ型によって異なります。このイベントで、コネクタは各 SQL Server データ型のフィールドを **リテラル型** と **セマンティック型** の両方にマップします。

コネクタは SQL Server のデータ型を **リテラル型** および **セマンティック型** の両方にマップできます。

リテラル型

Kafka Connect のスキーマタイプ

(**INT8**、**INT16**、**INT32**、**INT64**、**FLOAT32**、**FLOAT64**、**BOOLEAN**、**STRING**、**BYTES**、**ARRAY**、**MAP**、**STRUCT**) を使用して、値が文字通りどのように表現されるかを記述します。

セマンティック型

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。

デフォルトのデータ型変換が要件に合わない場合は、コネクタ用の **カスタムコンバータの作成** が可能です。

データ型マッピングの詳細については、以下を参照してください。

- [基本型](#)
- [時間値](#)
- [10 進数値](#)
- [タイムスタンプ値](#)

基本型

以下の表は、コネクタによる基本的な SQL Server データ型のマッピング方法を示しています。

表8.10 SQL Server コネクタによって使用されるデータ型マッピング

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

SQL Server のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
CHAR[(N)]	STRING	該当なし
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。

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** 設定プロパティの値によって異なります。**time.precision.mode** 設定プロパティが **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 のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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 のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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 設定プロパティが **connect** に設定された場合、コネクタは事前定義された 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 のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
DATE	INT32	org.apache.kafka.connect.data.Date エポックからの日数を表します。

SQL Server のデータ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
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、範囲が 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 が 3 よりも大きい場合は、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、範囲が 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 が 3 よりも大きい場合は、このモードでは精度が失われます。

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。

10 進数値

Debezium コネクターは、**decimal.handling.mode** コネクター設定プロパティの設定にしたがって 10 進数进行处理します。

decimal.handling.mode=precise

表8.11 **decimal.handling.mode=precise** の場合のマッピング

SQL Server タイプ	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名)
----------------	---------------	------------------

SQL Server タイプ	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名)
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。
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。
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。
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。

decimal.handling.mode=double

表8.12 decimal.handling.mode=double の場合のマッピング

SQL Server タイプ	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
SMALLMONEY[(M[,D])]	FLOAT64	該当なし
MONEY[(M[,D])]	FLOAT64	該当なし

decimal.handling.mode=string

表8.13 decimal.handling.mode=string の場合のマッピング

SQL Server タイプ	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
SMALLMONEY[(M[,D])]	STRING	該当なし

SQL Server タイプ	リテラル型	セマンティック型
MONEY[(M[,D])]	STRING	該当なし

8.3. DEBEZIUM コネクターを実行するための SQL SERVER のセットアップ

Debezium が SQL Server テーブルから変更イベントをキャプチャーするには、必要な権限を持つ SQL Server の管理者が最初にクエリを実行してデータベースで CDC を有効にします。その後、管理者は Debezium がキャプチャーする各テーブルに対して、CDC を有効にする必要があります。



注記

デフォルトでは、Microsoft SQL Server への JDBC 接続は SSL 暗号化によって保護されています。SQL Server データベースで SSL が有効になっていない場合、または SSL を使用せずにデータベースに接続する場合は、コネクター設定で **database.encrypt** プロパティの値を **false** に設定して SSL を無効にできます。

Debezium コネクターと使用するための SQL Server の設定に関する詳細は、以下を参照してください。

- [「SQL Server データベースでの CDC の有効化」](#)
- [「SQL Server テーブルでの CDC の有効化」](#)
- [「ユーザーが CDC テーブルにアクセスできることの確認」](#)
- [「Azure 上の SQL Server」](#)
- [「SQL Server キャプチャジョブエージェント設定のサーバー負荷およびレイテンシーへの影響」](#)
- [「SQL Server のキャプチャジョブエージェントの設定パラメーター」](#)

CDC の適用後、CDC が有効になっているテーブルにコミットされる **INSERT**、**UPDATE**、および **DELETE** 操作がすべてキャプチャーされます。その後、Debezium コネクターはこれらのイベントをキャプチャーして Kafka トピックに出力できます。

8.3.1. SQL Server データベースでの CDC の有効化

テーブルの CDC を有効にする前に、SQL Server データベースに対して CDC を有効にする必要があります。SQL Server 管理者は、システムストアプロシージャを実行して CDC を有効にします。システムストアプロシージャは、SQL Server Management Studio または Transact-SQL を使用すると実行できます。

前提条件

- SQL Server の **sysadmin** 固定サーバーロールのメンバーである。
- データベースの **db_owner** である。
- SQL Server Agent が稼働している。



注記

SQL Server の CDC 機能は、ユーザーが作成したテーブルでのみ発生する変更を処理します。SQL Server **master** データベースで CDC を有効にすることはできません。

手順

1. SQL Server Management Studio の **View** メニューから **Template Explorer** をクリックします。
2. **Template Browser** で、**SQL Server Templates** を展開します。
3. **Change Data Capture > Configuration** を展開した後、**Enable Database for CDC** をクリックします。
4. テンプレートで、**USE** ステートメントのデータベース名を、CDC に対して有効にするデータベースの名前に置き換えます。
5. ストアドプロシージャ **sys.sp_cdc_enable_db** を実行して、CDC 用のデータベースを有効にします。
データベースが CDC に対して有効になったら、**cdc** という名前のスキーマ、CDC ユーザー、メタデータテーブル、およびその他のシステムオブジェクトが作成されます。

以下の例は、データベース **MyDB** に対して CDC を有効にする方法を示しています。

例: CDC テンプレートに対する SQL Server データベースの有効化

```
USE MyDB
GO
EXEC sys.sp_cdc_enable_db
GO
```

8.3.2. SQL Server テーブルでの CDC の有効化

SQL Server 管理者は、Debezium がキャプチャーするソーステーブルで変更データキャプチャー (CDC) を有効にする必要があります。データベースが CDC に対してすでに有効になっている必要があります。テーブルで CDC を有効にするには、SQL Server 管理者はストアドプロシージャ **sys.sp_cdc_enable_table** をテーブルに対して実行します。ストアドプロシージャは、SQL Server Management Studio または Transact-SQL を使用すると実行できます。キャプチャーするすべてのテーブルに対して SQL Server の CDC を有効にする必要があります。

前提条件

- CDC が SQL Server データベースで有効になっている。
- SQL Server Agent が稼働している。
- データベースの **db_owner** 固定データベース出力のメンバーである。

手順

1. SQL Server Management Studio の **View** メニューから **Template Explorer** をクリックします。
2. **Template Browser** で、**SQL Server Templates** を展開します。

3. **Change Data Capture > Configuration**を展開した後、**Enable Table Specifying Filegroup Option** をクリックします。
4. テンプレートで、**USE** ステートメントのテーブル名を、キャプチャーするテーブルの名前に置き換えます。
5. ストアドプロシージャ **sys.sp_cdc_enable_table** を実行します。
以下の例は、テーブル **MyTable** に対して CDC を有効にする方法を示しています。

例: SQL Server テーブルに対する CDC の有効化

```
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
```

<> キャプチャーするテーブルの名前を指定します。<> ソーステーブルのキャプチャーされた列で **SELECT** 権限を付与するユーザーを追加できる **MyRole** ロールを指定します。 **sysadmin** または **db_owner** ロールのユーザーも、指定された変更テーブルにアクセスできます。 **@role_name** の値を **NULL** に設定して、 **sysadmin** または **db_owner** のメンバーのみがキャプチャーされた情報に完全にアクセスできるようにします。<> SQL Server がキャプチャーされたテーブルの変更テーブルを配置する **filegroup** を指定します。指定された **filegroup** は、すでに存在している必要があります。ソーステーブルに使用するのと同じ **filegroup** に変更テーブルを置かないことが推奨されます。

8.3.3. ユーザーが CDC テーブルにアクセスできることの確認

SQL Server 管理者は、システムストアドプロシージャを実行してデータベースまたはテーブルをクエリーし、その CDC 設定情報を取得できます。ストアドプロシージャは、SQL Server Management Studio または Transact-SQL を使用すると実行できます。

前提条件

- キャプチャーインスタンスのキャプチャーされたすべての列に対して **SELECT** 権限を持っている。 **db_owner** データベース出力のメンバーは、定義されたすべてのキャプチャーインスタンスの情報を確認できます。
- クエリーに含まれるテーブル情報に定義したゲーティングロールへのメンバーシップがある。

手順

1. SQL Server Management Studio の **View** メニューから **Object Explorer** をクリックします。
2. Object Explorer から **Databases** を展開し、 **MyDB** などのデータベースオブジェクトを展開します。
3. **Programmability > Stored Procedures > System Stored Procedures**を展開します。

4. **sys.sp_cdc_help_change_data_capture** ストアドプロシージャを実行して、テーブルを問い合わせます。
クエリーは空の結果を返しません。

次の例では、データベース **MyDB** 上でストアドプリファレンス **sys.sp_cdc_help_change_data_capture** を実行します。

例: CDC 設定情報のテーブルのクエリー

```
USE MyDB;
GO
EXEC sys.sp_cdc_help_change_data_capture
GO
```

クエリーは、CDC に対して有効になっているデータベースの各テーブルの設定情報を返し、呼び出し元のアクセスが許可される変更データが含まれます。結果が空の場合は、ユーザーに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できる権限があることを確認します。

8.3.4. Azure 上の SQL Server

Debezium SQL Server コネクターは Azure の SQL Server と併用できます。CDC for SQL Server on Azure の設定と Debezium での使用は、[こちらの例](#) を参考にしてください。

8.3.5. SQL Server キャプチャジョブエージェント設定のサーバー負荷およびレイテンシーへの影響

データベース管理者がソーステーブルに対して変更データキャプチャーを有効にすると、キャプチャジョブエージェントの実行が開始されます。エージェントは新しい変更イベントレコードをトランザクションログから読み取り、イベントレコードを変更データテーブルに複製します。変更がソーステーブルにコミットされてから、対応する変更テーブルに変更が反映される間、常に短いレイテンシーが間隔で発生します。この遅延間隔は、ソーステーブルで変更が発生したときから、Debezium がその変更を Apache Kafka にストリーミングできるようになるまでの差を表します。

データの変更に素早く対応する必要があるアプリケーションについては、ソースと変更テーブル間で密接に同期を維持するのが理想的です。キャプチャーエージェントを実行してできるだけ迅速に変更イベントを継続的に処理すると、スループットが増加し、レイテンシーが減少するため、イベントの発生後にほぼリアルタイムで新しいイベントレコードが変更テーブルに入力されることを想像するかもしれません。しかし、これは必ずしもそうであるとは限りません。同期を即時に行くとパフォーマンスに影響します。キャプチャジョブエージェントが新しいイベントレコードについてデータベースにクエリーを実行するたびに、データベースホストの CPU 負荷が増加します。サーバーへの負荷が増えると、データベース全体のパフォーマンスに悪影響を及ぼす可能性があり、特にデータベースの使用がピークに達するときにトランザクションの効率が低下する可能性があります。

データベースメトリクスを監視して、サーバーがキャプチャーエージェントのアクティビティをサポートできなくなるレベルにデータベースが達した場合に認識できるようにすることが重要となります。パフォーマンスの問題を認識した場合、データベースホストの全体的な CPU 負荷を許容できるレイテンシーで調整するために、SQL Server のキャプチャーエージェント設定を変更できます。

8.3.6. SQL Server のキャプチャジョブエージェントの設定パラメーター

SQL Server では、キャプチャジョブエージェントの動作を制御するパラメーターは SQL Server テーブル **msdb.dbo.cdc_jobs** に定義されます。キャプチャジョブエージェントの実行中にパフォーマンスの問題が発生した場合は、**sys.sp_cdc_change_job** ストアドプロシージャを実行し、新しい値を指定することで、キャプチャジョブ設定を調整し、CPU の負荷を軽減します。



注記

SQL Server のキャプチャジョブエージェントパラメーターの設定方法に関する具体的なガイダンスは、本書の範囲外となります。

以下のパラメーターは、Debezium SQL Server コネクターと使用するキャプチャーエージェントの動作を変更する場合に最も重要になります。

pollinginterval

- キャプチャーエージェントがログスキャンのサイクルで待機する秒数を指定します。
- 値が大きいほど、データベースホストの負荷が減少し、レイテンシーが増加します。
- **0** を値として指定すると、スキャン間の待ち時間はありません。
- デフォルト値は **5** です。

maxtrans

- 各ログスキャンサイクル中に処理するトランザクションの最大数を指定します。キャプチャジョブが指定の数のトランザクションを処理したら、次のスキャンを開始する前に **pollinginterval** によって指定された期間、一時停止します。
- 値が小さいほど、データベースホストの負荷が減少し、レイテンシーが増加します。
- デフォルト値は **500** です。

maxscans

- キャプチャジョブが、データベーストランザクションログの完全な内容のキャプチャーを試みるスキャンサイクルの数の制限を指定します。 **continuous** パラメーターが **1** に設定されると、ジョブはスキャンを再開する前に **pollinginterval** で指定された期間一時停止します。
- 値が小さいほど、データベースホストの負荷が減少し、レイテンシーが増加します。
- デフォルト値は **10** です。

関連情報

- キャプチャーエージェントパラメーターの詳細は、SQL Server のドキュメントを参照してください。

8.4. DEBEZIUM SQL SERVER コネクターのデプロイ

以下の方法のいずれかを使用して Debezium SQL Server コネクターをデプロイできます。

- [AMQ Streams](#) を使用して、コネクタープラグインが含まれるイメージを自動的に作成します。これは推奨される方法です。
- [Dockerfile](#) からカスタム Kafka Connect コンテナイメージをビルドします。

関連情報

- [「Debezium SQL Server コネクター設定プロパティの説明」](#)

8.4.1. AMQ Streams を使用した SQL Server コネクターデプロイメント

Debezium 1.7 以降、Debezium コネクターのデプロイに推奨される方法は、AMQ Streams を使用してコネクタープラグインが含まれる Kafka Connect コンテナイメージをビルドすることです。

デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- Kafka Connect インスタンスを定義し、コネクターアーティファクトに関する情報をイメージに含める必要がある **KafkaConnect** CR。
- コネクターがソースデータベースにアクセスするために使用する情報を提供する **KafkaConnector** CR。AMQStreams が Kafka Connect Pod を開始した後、**KafkaConnector** CR を適用してコネクターを開始します。

Kafka Connect イメージのビルド仕様では、デプロイ可能なコネクターを指定できます。各コネクタープラグインに対して、デプロイメントに利用可能にする他のコンポーネントを指定することもできます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。AMQ Streams が Kafka Connect イメージをビルドすると、指定のアーティファクトをダウンロードし、イメージに組み込みます。

Kafka Connect CR の **spec.build.output** パラメーターは、生成される **KafkaConnect** コンテナイメージを格納する場所を指定します。コンテナイメージは Docker レジストリーまたは OpenShift ImageStream に保存できます。イメージを ImageStream に保存するには、Kafka Connect をデプロイする前に ImageStream を作成する必要があります。イメージストリームは自動的に作成されません。



注記

KafkaConnect リソースを使用してクラスターを作成する場合は、Kafka Connect REST API を使用してコネクターを作成または更新できません。ただし、REST API を使用して情報を取得できます。

関連情報

- OpenShift での AMQ Streams の設定ガイドの [Kafka Connect の設定](#) を参照してください。
- [AMQ Streams を使用した新しいコンテナイメージの自動作成と OpenShift での AMQ Streams のアップグレード](#)

8.4.2. AMQ Streams を使用した Debezium SQL Server コネクターのデプロイ

以前のバージョンの AMQ Streams では、OpenShift に Debezium コネクターをデプロイするには、最初にコネクター用の Kafka Connect イメージをビルドする必要がありました。コネクターを OpenShift にデプロイする場合に現在推奨される方法は、AMQ Streams でビルド設定を使用して、使用する Debezium コネクタープラグインが含まれる Kafka Connect コンテナイメージを自動的にビルドすることです。

ビルドプロセス中、AMQ Streams Operator は Debezium コネクター定義を含む **KafkaConnect** カスタムリソースの入力パラメーターを Kafka Connect コンテナイメージに変換します。このビルドは、Red Hat Maven リポジトリまたは別の設定済みの HTTP サーバーから必要なアーティファクトをダウンロードします。

新規に作成されたコンテナは **.spec.build.output** に指定されるコンテナレジストリーにプッシュされ、Kafka Connect クラスターのデプロイに使用されます。AMQ Streams が Kafka Connect イメージ

をビルドしたら、**KafkaConnector** カスタムリソースを作成し、ビルドに含まれるコネクタを起動します。

前提条件

- クラスタ Operator がインストールされている OpenShift クラスタにアクセスできる。
- AMQ Streams Operator が稼働している。
- Kafka クラスタが [OpenShift での AMQ Streams のデプロイおよびアップグレード](#) に記載されているようにデプロイされている。
- [Kafka Connect が AMQ Streams にデプロイされている](#)。
- Red Hat Integration ライセンスがある。
- [OpenShift oc CLI](#) クライアントがインストールされている、または OpenShift Container Platform Web コンソールにアクセスできる。
- Kafka Connect ビルドイメージの保存方法に応じて、レジストリーのパーミッションを用意するか、ImageStream リソースを作成しておく。

ビルドイメージを Red Hat Quay.io または Docker Hub などのイメージレジストリーに保存する場合は、以下が必要です。

- レジストリーでイメージを作成し、管理するためのアカウントおよびパーミッション

ビルドイメージをネイティブ OpenShift ImageStream として保存する場合

- [ImageStream](#) リソースを新規コンテナイメージを保存するためにクラスタにデプロイします。クラスタの ImageStream を明示的に作成する必要があります。ImageStreams はデフォルトでは利用できません。ImageStreams の詳細は、[OpenShift Container Platform でのイメージストリームの管理](#) を参照してください。

手順

1. OpenShift クラスタにログインします。
2. コネクタの Debezium **KafkaConnect** カスタムリソース (CR) を作成するか、既存のリソースを変更します。たとえば、**metadata.annotations** および **spec.build** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

例8.1 Debezium コネクタを含む KafkaConnect カスタムリソースを定義する dbz-connect.yaml ファイル

次の例では、カスタムリソースは、次のアーティファクトをダウンロードするように設定されています。

- Debezium SQL Server コネクタアーカイブ。
- サービスレジストリーアーカイブ。Service Registry はオプションのコンポーネントです。コネクタで Avro シリアル化を使用する場合にのみ、Service Registry コンポーネントを追加します。
- Debezium スクリプティング SMT アーカイブと、Debezium コネクタで使用する関連

スクリプティングエンジン。SMT アーカイブとスクリプト言語の依存関係はオプションのコンポーネントです。Debezium [コンテンツベースのルーティング SMT](#) または [フィルター SMT](#) を使用する場合にのみ、これらのコンポーネントを追加します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ❶
spec:
  version: 3.3.1
  build: ❷
  output: ❸
    type: imagestream ❹
    image: debezium-streams-connect:latest
  plugins: ❺
    - name: debezium-connector-sqlserver
      artifacts:
        - type: zip ❻
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-sqlserver/2.1.4.Final-redhat-00001/debezium-connector-sqlserver-2.1.4.Final-redhat-00001-plugin.zip ❼
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.3.0.Final-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.3.0.Final-redhat-<build-number>.zip ❽
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/2.1.4.Final-redhat-00001/debezium-scripting-2.1.4.Final-redhat-00001.zip ❾
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy/3.0.11/groovy-3.0.11.jar ❿
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-jsr223/3.0.11/groovy-jsr223-3.0.11.jar
        - type: jar
          url: https://repo1.maven.org/maven2/org/codehaus/groovy/groovy-json/3.0.11/groovy-json-3.0.11.jar

  bootstrapServers: debezium-kafka-cluster-kafka-bootstrap:9093

  ...

```

表8.14 Kafka Connect 設定の説明

項目	説明
1	strimzi.io/use-connector-resources アノテーションを true に設定して、クラスター Operator が KafkaConnector リソースを使用してこの Kafka Connect クラスター内のコネクタを設定できるようにします。

項目	説明
2	spec.build 設定は、ビルドイメージの保存場所を指定し、プラグインアーティファクトの場所と共にイメージに追加するプラグインを一覧表示します。
3	build.output は、新たにビルドされたイメージが保存されるレジストリーを指定します。
4	イメージ出力の名前およびイメージ名を指定します。 output.type の有効な値は、Docker Hub や Quay などのコンテナレジストリーにプッシュする場合は docker 、内部の OpenShift ImageStream にイメージをプッシュする場合は imagestream です。ImageStream を使用するには、ImageStream リソースをクラスターにデプロイする必要があります。KafkaConnect 設定で build.output を指定する方法の詳細は、OpenShift での AMQ Streams の設定の AMQ Streams ビルドスキーマ参照 を参照してください。
5	plugins 設定は、Kafka Connect イメージに追加するすべてのコネクタを一覧表示します。一覧の各エントリーについて、プラグイン name と、コネクタのビルドに必要なアーティファクトに関する情報を指定します。任意で、各コネクタプラグインに対して、コネクタと使用できる他のコンポーネントを含めることができます。たとえば、Service Registry アーティファクトまたは Debezium スクリプトコンポーネントを追加できます。
6	artifacts.type の値は、 artifacts.url で指定したアーティファクトのファイルタイプを指定します。有効なタイプは zip 、 tgz 、または jar です。Debezium コネクタアーカイブは、 .zip ファイル形式で提供されます。 type の値は、 url フィールドで参照されるファイルのタイプと一致する必要があります。
7	artifacts.url の値は、コネクタアーティファクトのファイルを格納する Maven リポジトリなどの HTTP サーバーのアドレスを指定します。Debezium コネクタアーティファクトは Red Hat リポジトリで入手できます。OpenShift クラスターは指定されたサーバーにアクセスする必要があります。
8	(オプション) サービスレジストリーコンポーネントをダウンロードするためのアーティファクト type と url を指定します。デフォルトの JSON コンバーターを使用する代わりに、コネクタが Apache Avro を使用して、Service Registry を使用してイベントのキーと値をシリアル化する場合にのみ、ServiceRegistry アーティファクトを含めます。
9	(オプション) Debezium コネクタで使用する Debezium スクリプト SMT アーカイブのアーティファクト type と url を指定します。Debezium コンテンツベースルーティング SMT または フィルター SMT を使用する場合にのみ、スクリプト SMT を含めます。スクリプト SMT を使用するには、groovy などの JSR 223 準拠のスクリプト実装もデプロイする必要があります。

項目	説明
10	<p>(オプション) JSR 223 準拠のスクリプト実装の JAR ファイルのアーティファクト type と url を指定します。これは、Debezium スクリプト SMT で必要です。</p> <div style="display: flex; align-items: flex-start;"> <div style="width: 40px; height: 40px; background-color: black; margin-right: 10px;"></div> <div> <p>重要</p> <p>AMQ Streams を使用して Kafka Connect イメージにコネクタプラグインを組み込む場合、必要なスクリプト言語コンポーネントごとに、artifacts.url に JAR ファイルの場所を指定し、artifacts.type の値も jar に設定する必要があります。値が無効な場合、実行時にコネクタが失敗します。</p> </div> </div> <p>スクリプト SMT で Apache Groovy 言語を使用できるようにするために、この例のカスタムリソースは、次のライブラリーの JAR ファイルを取得します。</p> <ul style="list-style-type: none"> ● groovy ● groovy-jsr223 (スクリプトエージェント) ● groovy-json (JSON 文字列を解析するためのモジュール) <p>別の方法として、Debezium スクリプト SMT は、GraalVM JavaScript の JSR 223 実装の使用もサポートします。</p>

- 以下のコマンドを入力して、**KafkaConnect** ビルド仕様を OpenShift クラスタに適用します。

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

Streams Operator はカスタムリソースで指定された設定に基づいて、デプロイする Kafka Connect イメージを準備します。

ビルドが完了すると、Operator はイメージを指定されたレジストリーまたは ImageStream にプッシュし、Kafka Connect クラスタを起動します。設定に一覧表示されているコネクタアーティファクトはクラスタで利用できます。

- KafkaConnector** リソースを作成し、デプロイする各コネクタのインスタンスを定義します。たとえば、以下の **KafkaConnector** CR を作成し、**sqlserver-inventory-connector.yaml** として保存します。

例8.2 Debezium コネクタの KafkaConnector カスタムリソースを定義する sqlserver-inventory-connector.yaml ファイル

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
name: inventory-connector-sqlserver ❶
spec:
  class: io.debezium.connector.sqlserver.SqlServerConnector ❷
  tasksMax: 1 ❸
```



```

config: 4
  schema.history.internal.kafka.bootstrap.servers: debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092
  schema.history.internal.kafka.topic: schema-changes.inventory
  database.hostname: sqlserver.debezium-sqlserver.svc.cluster.local 5
  database.port: 1433 6
  database.user: debezium 7
  database.password: dbz 8
  topic.prefix: inventory-connector-sqlserver 9
  table.include.list: dbo.customers 10
  ...

```

表8.15 コネクター設定の説明

項目	説明
1	Kafka Connect クラスターに登録するコネクターの名前。
2	コネクタークラスの名前。
3	同時に動作できるタスクの数。
4	コネクターの設定。
5	ホストデータベースインスタンスのアドレス。
6	データベースインスタンスのポート番号。
7	Debezium がデータベースへの接続に使用するアカウントの名前。
8	Debezium がデータベースユーザーアカウントに接続するために使用するパスワード。
9	データベースインスタンスまたはクラスターのトピック接頭辞。 指定する名前は、英数字またはアンダースコアのみで設定する必要があります。 トピック接頭辞は、このコネクターから変更イベントを受信する Kafka トピックの接頭辞として使用されるため、名前はクラスターのコネクター間で一意である必要があります。 コネクターを Avro コネクター と統合する場合、この名前空間は関連する Kafka Connect スキーマの名前や、対応する Avro スキーマの名前空間でも使用されます。
10	コネクターが変更イベントをキャプチャーするテーブルの一覧

5. 以下のコマンドを実行してコネクターリソースを作成します。

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

以下に例を示します。

■

```
oc create -n debezium -f sqlserver-inventory-connector.yaml
```

コネクタは Kafka Connect クラスターに登録され、**KafkaConnector** CR の **spec.config.database.dbname** で指定されたデータベースに対して実行を開始します。コネクタ Pod の準備ができると、Debezium が実行されます。

これで、[Debezium SQL デプロイメントを検証する](#) 準備が整いました。

8.4.3. Dockerfile からカスタム Kafka Connect コンテナイメージをビルドして Debezium SQL Server コネクタのデプロイ

Debezium SQL Server コネクタをデプロイするには、Debezium コネクタアーカイブが含まれるカスタム Kafka Connect コンテナイメージをビルドし、このコンテナイメージをコンテナレジストリーにプッシュする必要があります。次に、以下のカスタムリソース (CR) を作成する必要があります。

- Kafka Connect インスタンスを定義する **KafkaConnect** CR。 **image** は Debezium コネクタを実行するために作成したイメージの名前を指定します。この CR を、[Red Hat AMQ Streams](#) がデプロイされている OpenShift インスタンスに適用します。AMQ Streams は、Apache Kafka を OpenShift に取り入れる operator およびイメージを提供します。
- Debezium SQL Server コネクタを定義する **KafkaConnector** CR。この CR を **KafkaConnect** CR を適用するのと同じ OpenShift インスタンスに適用します。

前提条件

- SQL Server が稼働し、[Debezium コネクタと連携するように SQL Server を設定する手順](#) が完了済みである必要があります。
- AMQ Streams は OpenShift にデプロイされ、Apache Kafka および Kafka Connect が稼働している必要があります。詳細は、[Deploying and Upgrading AMQ Streams on OpenShift](#) を参照してください。
- Podman または Docker がインストールされている。
- Debezium コネクタを実行するコンテナを追加する予定のコンテナレジストリー ([quay.io](#) や [docker.io](#) など) でコンテナを作成および管理するアカウントとパーミッションを持っている。

手順

1. Kafka Connect の Debezium SQL Server コンテナを作成します。
 - a. [registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12](#) をベースイメージとして使用して、新規の Dockerfile を作成します。例えば、ターミナルウィンドウから、以下のコマンドを入力します。

```
cat <<EOF >debezium-container-for-sqlserver.yaml 1
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
RUN mkdir -p /opt/kafka/plugins/debezium 2
RUN cd /opt/kafka/plugins/debezium/ \
&& curl -O https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-sqlserver/2.1.4.Final-redhat-00001/debezium-connector-sqlserver-2.1.4.Final-redhat-00001-plugin.zip \
```

```
&& unzip debezium-connector-sqlserver-2.1.4.Final-redhat-00001-plugin.zip \
&& rm debezium-connector-sqlserver-2.1.4.Final-redhat-00001-plugin.zip
RUN cd /opt/kafka/plugins/debezium/
USER 1001
EOF
```

項目	説明
1	任意のファイル名を指定できます。
2	Kafka Connect プラグインディレクトリーへのパスを指定します。Kafka Connect のプラグインディレクトリーが別の場所にある場合は、このパスを実際のディレクトリーのパスに置き換えてください。

このコマンドは、現在のディレクトリーに **debezium-container-for-sqlserver.yaml** という名前の Dockerfile を作成します。

- b. 前のステップで作成した **debezium-container-for-sqlserver.yaml** Docker ファイルからコンテナイメージをビルドします。ファイルが含まれるディレクトリーから、ターミナルウィンドウを開き、以下のコマンドのいずれかを入力します。

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

```
docker build -t debezium-container-for-sqlserver:latest .
```

上記のコマンドは、**debezium-container-for-sqlserver** という名前のコンテナイメージを構築します。

- c. カスタムイメージを quay.io などのコンテナレジストリーまたは内部のコンテナレジストリーにプッシュします。コンテナレジストリーは、イメージをデプロイする OpenShift インスタンスで利用できる必要があります。以下のいずれかのコマンドを実行します。

```
podman push <myregistry.io>/debezium-container-for-sqlserver:latest
```

```
docker push <myregistry.io>/debezium-container-for-sqlserver:latest
```

- d. 新しい Debezium SQL Server KafkaConnect カスタムリソース (CR) を作成します。たとえば、**annotations** および **image** プロパティを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。以下の例は、**KafkaConnect** カスタムリソースを記述する **dbz-connect.yaml** ファイルからの抜粋を示しています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
annotations:
  strimzi.io/use-connector-resources: "true" 1
spec:
  #...
```

```
image: debezium-container-for-sqlserver 2
```

```
...
```

項目	説明
1	KafkaConnector リソースはこの Kafka Connect クラスターでコネクタを設定するために使用されることを、 metadata.annotations は Cluster Operator に示します。
2	spec.image は Debezium コネクタを実行するために作成したイメージの名前を指定します。設定された場合、このプロパティによって Cluster Operator の STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE 変数がオーバーライドされます。

- e. 以下のコマンドを入力して、**KafkaConnect** CR を OpenShift Kafka Connect 環境に適用します。

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

このコマンドは、Debezium コネクタを実行するために作成したイメージの名前を指定する Kafka Connect インスタンスを追加します。

2. Debezium SQL Server コネクタインスタンスを設定する **KafkaConnector** カスタムリソースを作成します。

通常、コネクタに使用できる設定プロパティを使用して、**.yaml** ファイルに Debezium SQL Server コネクタを設定します。コネクタ設定は、Debezium に対して、スキーマおよびテーブルのサブセットにイベントを生成するよう指示する可能性があり、または機密性の高い、大きすぎる、または不必要な指定の列で Debezium が値を無視、マスク、または切り捨てるようにプロパティを設定する可能性もあります。

以下の例では、ポート **1433** で PostgreSQL サーバーホスト **192.168.99.100** に接続する Debezium コネクタを設定します。このホストには、**testDB** という名前のデータベース、**customers** という名前のテーブル、**inventory-connector-sqlserver** という論理名のサーバーがあります。

SQL Server inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: inventory-connector-sqlserver 1
  labels:
    strimzi.io/cluster: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: 'true'
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.names: testDB1,testDB2 7
topic.prefix: inventory-connector-sqlserver 8
table.include.list: dbo.customers 9
schema.history.internal.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 10
schema.history.internal.kafka.topic: schemahistory.fullfillment 11
database.ssl.truststore: path/to/trust-store 12
database.ssl.truststore.password: password-for-trust-store 13

```

表8.16 コネクタ設定の説明

項目	説明
1	Kafka Connect サービスに登録する場合のコネクタの名前。
2	この SQL Server コネクタクラスの名前。
3	SQL Server インスタンスのアドレス。
4	SQL Server インスタンスのポート番号。
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	名前空間を形成する SQL Server インスタンス/クラスターのトピック接頭辞で、コネクタが書き込む Kafka トピックの名前、Kafka Connect スキーマ名、および Arvo コンバーター が使用される場合に対応する Avro スキーマの名前空間すべてに使用されません。
9	コネクタは dbo.customers テーブルからの変更のみをキャプチャーします。
10	DDL ステートメントをデータベーススキーマ履歴トピックに書き込み、復元するためにコネクタによって使用される Kafka ブローカーのリスト。
11	コネクタが DDL ステートメントを書き、復元するデータベーススキーマ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。
12	サーバーの署名者証明書を格納する SSL トラストストアへのパス。データベースの暗号化が無効になっている場合 (database.encrypt=false) を除き、このプロパティは必須です。
13	SSL トラストストアのパスワード。データベースの暗号化が無効になっている場合 (database.encrypt=false) を除き、このプロパティは必須です。

3. Kafka Connect でコネクタインスタンスを作成します。たとえば、**KafkaConnector** リソースを **inventory-connector.yaml** ファイルに保存した場合は、以下のコマンドを実行します。

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

上記のコマンドは **inventory-connector** を登録し、コネクタは **KafkaConnector** CR に定義されている **testDB** データベースに対して実行を開始します。

Debezium SQL Server コネクタが実行していることの確認

コネクタがエラーなしで正常に起動すると、コネクタがキャプチャーするように設定された各テーブルのトピックが作成されます。ダウストリームアプリケーションは、これらのトピックをサブスクライブして、ソースデータベースで発生する情報イベントを取得できます。

コネクタが実行されていることを確認するには、OpenShift Container Platform Web コンソールまたは OpenShift CLI ツール (oc) から以下の操作を実行します。

- コネクタのステータスを確認します。
- コネクタがトピックを生成していることを確認します。
- 各テーブルの最初のスナップショットの実行中にコネクタが生成する読み取り操作 ("op":"r") のイベントがトピックに反映されていることを確認します。

前提条件

- Debezium コネクタが AMQ Streams on OpenShift にデプロイされている。
- OpenShift **oc** CLI クライアントがインストールされている。
- OpenShift Container Platform Web コンソールへのアクセスがある。

手順

1. 以下の方法のいずれかを使用して **KafkaConnector** リソースのステータスを確認します。
 - OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaConnector** を入力します。
 - c. **KafkaConnectors** リストから確認するコネクタの名前をクリックします (例: **inventory-connector-sqlserver**)。
 - d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
 - ターミナルウィンドウから以下を実行します。
 - a. 以下のコマンドを入力します。

```
oc describe KafkaConnector <connector-name> -n <project>
```

以下に例を示します。

```
oc describe KafkaConnector inventory-connector-sqlserver -n debezium
```

このコマンドは、以下の出力のようなステータス情報を返します。

例8.3 KafkaConnector リソースのステータス

```
Name:      inventory-connector-sqlserver
Namespace: debezium
Labels:    strimzi.io/cluster=debezium-kafka-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:      KafkaConnector

...

Status:
Conditions:
  Last Transition Time: 2021-12-08T17:41:34.897153Z
  Status:              True
  Type:                Ready
Connector Status:
Connector:
  State:  RUNNING
  worker_id: 10.131.1.124:8083
Name:      inventory-connector-sqlserver
Tasks:
  Id:      0
  State:   RUNNING
  worker_id: 10.131.1.124:8083
  Type:    source
Observed Generation: 1
Tasks Max: 1
Topics:
inventory-connector-sqlserver.inventory
inventory-connector-sqlserver.inventory.addresses
inventory-connector-sqlserver.inventory.customers
inventory-connector-sqlserver.inventory.geom
inventory-connector-sqlserver.inventory.orders
inventory-connector-sqlserver.inventory.products
inventory-connector-sqlserver.inventory.products_on_hand
Events: <none>
```

2. コネクターによって Kafka トピックが作成されたことを確認します。

- OpenShift Container Platform Web コンソールから以下を実行します。
 - a. **Home** → **Search** に移動します。
 - b. **Search** ページで **Resources** をクリックし、**Select Resource** ボックスを開き、**KafkaTopic** を入力します。
 - c. **KafkaTopics** リストから確認するトピックの名前をクリックします (例: **inventory-connector-sqlserver.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**)。

- d. **Conditions** セクションで、**Type** および **Status** 列の値が **Ready** および **True** に設定されていることを確認します。
- ターミナルウィンドウから以下を実行します。
 - 以下のコマンドを入力します。

```
oc get kafkatopics
```

このコマンドは、以下の出力のようなステータス情報を返します。

例8.4 KafkaTopic リソースのステータス

```

NAME                                     CLUSTER
PARTITIONS REPLICATION FACTOR  READY
connect-cluster-configs                 debezium-kafka-cluster  1
1             True
connect-cluster-offsets                 debezium-kafka-cluster  25
1             True
connect-cluster-status                   debezium-kafka-cluster  5
1             True
consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
debezium-kafka-cluster 50             1             True
inventory-connector-sqlserver--a96f69b23d6118ff415f772679da623fbbb99421
debezium-kafka-cluster 1             1             True
inventory-connector-sqlserver.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480     debezium-kafka-cluster
1             1             True
inventory-connector-sqlserver.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b     debezium-kafka-cluster  1
1             True
inventory-connector-sqlserver.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5     debezium-kafka-cluster
1             1             True
inventory-connector-sqlserver.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d     debezium-kafka-cluster
1             1             True
inventory-connector-sqlserver.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef     debezium-kafka-cluster  1
1             True
inventory-connector-sqlserver.inventory.products_on_hand---
8649e0f17fcc9212e266e31a7aeaa4585e5c6b5     debezium-kafka-cluster  1
1             True
schema-changes.inventory                 debezium-kafka-cluster
1             1             True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55     debezium-
kafka-cluster 1             1             True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b     debezium-kafka-cluster  1  1
True

```

- トピックの内容を確認します。

- ターミナルウィンドウから、以下のコマンドを入力します。


```
oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=<topic-name>
```

以下に例を示します。

```
oc exec -n debezium -it debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
> --bootstrap-server localhost:9092 \
> --from-beginning \
> --property print.key=true \
> --topic=inventory-connector-sqlserver.inventory.products_on_hand
```

トピック名を指定する形式は、手順1で返された **oc describe** コマンドと同じです (例: **inventory-connector-sqlserver.inventory.addresses**)。

トピックの各イベントについて、このコマンドは、以下の出力のような情報を返します。

例8.5 Debezium 変更イベントの内容

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "int32",
        "optional": false,
        "field": "product_id",
        "optional": false,
        "name": "inventory-connector-sqlserver.inventory.products_on_hand.Key",
        "payload": {
          "product_id": 101
        }
      }
    ]
  },
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "product_id",
            "optional": true,
            "name": "inventory-connector-sqlserver.inventory.products_on_hand.Value",
            "field": "before"
          },
          {
            "type": "struct",
            "fields": [
              {
                "type": "int32",
                "optional": false,
                "field": "product_id",
                "optional": true,
                "name": "inventory-connector-sqlserver.inventory.products_on_hand.Value",
                "field": "after"
              },
              {
                "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": "string",
                "optional": true,
                "name": "io.debezium.data.Enum",
                "version": 1,
                "parameters": {
                  "allowed": "true,last,false",
                  "default": "false",
                  "field": "snapshot"
                }
              },
              {
                "type": "string",
                "optional": false,
                "field": "db"
              },
              {
                "type": "string",
                "optional": true,
                "field": "sequence"
              },
              {
                "type": "string",
                "optional": true,
                "field": "table"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "server_id"
              },
              {
                "type": "string",
                "optional": true,
                "field": "gtid"
              },
              {
                "type": "string",
                "optional": false,
                "field": "file"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "pos"
              },
              {
                "type": "int32",
                "optional": false,
                "field": "row"
              },
              {
                "type": "int64",
                "optional": true,
                "field": "thread"
              },
              {
                "type": "string",
                "optional": true,
                "field": "query"
              }
            ],
            "optional": false,
            "name": "io.debezium.connector.sqlserver.Source",
            "field": "source"
          },
          {
            "type": "string",
            "optional": false,
            "field": "op"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
          },
          {
            "type": "struct",
            "fields": [
              {
                "type": "string",
                "optional": false,
                "field": "id"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "total_order"
              },
              {
                "type": "int64",
                "optional": false,
                "field": "data_collection_order"
              }
            ],
            "optional": true,
            "field": "transaction"
          },
          {
            "optional": false,
            "name": "inventory-connector-sqlserver.inventory.products_on_hand.Envelope",
            "payload": {
              "before": null,
              "after": {
                "product_id": 101,
                "quantity": 3,
                "source": {
                  "version": "2.1.4.Final-redhat-00001",
                  "connector": "sqlserver",
                  "name": "inventory-connector-"
                }
              }
            }
          }
        ]
      }
    ]
  }
}
```

```
sqlserver","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":
:"products_on_hand","server_id":0,"gtid":null,"file":"sqlserver-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"t
ransaction":null}}
```

上記の例では、**payload** 値は、コネクタースナップショットがテーブル **inventory.products_on_hand** から読み込み (**op** = "r") イベントを生成したことを示しています。**product_id** レコードの **before** 状態は **null** であり、レコードに以前の値が存在しないことを示します。**"after"** 状態が **product_id 101** で項目の **quantity** を **3** で示しています。

Debezium SQL Server コネクタに設定できる設定プロパティの完全リストは [SQL Server コネクタプロパティ](#) を参照してください。

結果

コネクタが起動すると、コネクタが設定された SQL Server データベースの [整合性スナップショット](#) が実行されます。その後、コネクタは行レベルの操作のデータ変更イベントの生成を開始し、変更イベントレコードを Kafka トピックにストリーミングします。

8.4.4. Debezium SQL Server コネクタ設定プロパティの説明

Debezium SQL Server コネクタには、アプリケーションに適したコネクタ動作を実現するために使用できる設定プロパティが多数あります。多くのプロパティにはデフォルト値があります。

プロパティに関する情報は、以下のように設定されています。

- [必要なコネクタ設定プロパティ](#)
- [高度なコネクタ設定プロパティ](#)
- Debezium がデータベース履歴トピックから読み取るイベントを処理する方法を制御する [データベース履歴コネクタ設定プロパティ](#)。
 - [パススルーデータベーススキーマ履歴プロパティ](#)
- データベースドライバーの動作を制御する [パススルーデータベースドライバープロパティ](#)。

Debezium SQL Server コネクタ設定プロパティ (必須)

以下の設定プロパティは、デフォルト値がない場合は**必須**です。

プロパティ	デフォルト	説明
name	デフォルトなし	コネクタの一意名。同じ名前で再登録を試みると失敗します。(このプロパティはすべての Kafka Connect コネクタに必要です)
connector.class	デフォルトなし	コネクタの Java クラスの名前。SQL Server コネクタには、常に io.debezium.connector.sqlserver.SqlServerConnector の値を使用してください。

プロパティ	デフォルト	説明
tasks.max	1	コネクターがデータベースインスタンスからデータをキャプチャーするために使用できるタスクの最大数を指定します。
database.hostname	デフォルトなし	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	デフォルトなし	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。Kerberos 認証を使用する場合は省略可能で、 パススループロパティ を使用して設定することができます。
database.password	デフォルトなし	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.instance	デフォルトなし	SQL Server の名前付きインスタンス のインスタンス名を指定します。

プロパティ	デフォルト	説明
topic.prefix	デフォルトなし	<p>Debezium にキャプチャーさせる SQL Server データベースサーバーの名前空間を提供するトピック接頭辞。接頭辞は、他のコネクタ全体で一貫した必要があります。これは、このコネクタからレコードを受信するすべての Kafka トピックの接頭辞として使用されるためです。データベースサーバーの論理名には英数字とハイフン、ドット、アンダースコアのみを使用する必要があります。</p> <div data-bbox="884 622 1428 1218" style="background-color: #fff9c4; padding: 10px; border: 1px solid #ccc;"> <p style="text-align: center;"> 警告</p> <p>このプロパティの値を変更しないでください。名前を変更すると、再起動後に、元のトピックにイベントを発行し続けるのではなく、新しい値に基づいた名前のトピックに後続のイベントを発行します。また、コネクタはデータベーススキーマ履歴トピックを復元できません。</p> </div>
schema.include.list	デフォルトなし	<p>変更をキャプチャーする対象とするスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。schema.include.list に含まれていないスキーマ名は、変更をキャプチャーする対象から除外されます。デフォルトでは、コネクタはシステム以外のすべてのスキーマの変更をキャプチャーします。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、schema.exclude.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
schema.exclude.list	デフォルトなし	<p>変更をキャプチャーする対象としないスキーマの名前と一致する正規表現のコンマ区切りリスト (任意)。システムスキーマ以外で、schema.exclude.list に名前が含まれていないスキーマの変更がキャプチャーされます。</p> <p>スキーマの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定した式は、スキーマ名に存在する可能性のある部分文字列とは一致しない、スキーマの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、schema.include.list プロパティを設定しないでください。</p>
table.include.list	デフォルトなし	<p>Debezium にキャプチャーさせるテーブルの完全修飾テーブル識別子に一致する、コンマ区切りの正規表現のリスト (任意)。デフォルトでは、コネクターは指定のスキーマのシステム以外のテーブルをすべてキャプチャーします。このプロパティが設定されている場合、コネクターは指定されたテーブルからのみ変更をキャプチャします。各識別子の形式は schemaName.tableName です。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.exclude.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
table.exclude.list	デフォルトなし	<p>キャプチャーから除外するテーブルの完全修飾テーブル識別子に一致するコンマ区切りの正規表現のリスト (任意)。Debezium は table.exclude.list に含まれていないテーブルをすべてキャプチャーします。各識別子の形式は <code>schemaName.tableName</code> です。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、table.include.list プロパティも設定しないでください。</p>
column.include.list	空の文字列	<p>変更イベントメッセージの値に含ま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 <code>schemaName.tableName.columnName</code> です。プライマリーキー列は、値に含まれていない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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、column.exclude.list プロパティを設定しないでください。</p>
column.exclude.list	空の文字列	<p>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 <code>schemaName.tableName.columnName</code> です。プライマリーキー列は、値から除外される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。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列名に存在する可能性のある部分文字列とは一致しない、列の名前文字列全体と照合されます。</p> <p>このプロパティを設定に含める場合は、column.include.list プロパティも設定しないでください。</p>

プロパティ	デフォルト	説明
<p><code>column.mask.hash.hashAlgorithm.with.salt.salt;</code> <code>column.mask.hash.v2.hashAlgorithm.with.salt.salt</code></p>	<p>該当なし</p>	<p>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)。列の完全修飾名の形式は <code>`<schemaName>.<tableName>.<columnName>`</code> です。</p> <p>列の名前を一致させるために、Debezium は指定した正規表現を <code>_アンカー正規表現</code> として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。作成された変更イベントレコードでは、指定された列の値は仮名に置き換えられます。</p> <p>仮名は、指定された <code>hashAlgorithm</code> と <code>salt</code> を適用すると得られるハッシュ化された値で設定されます。使用されるハッシュ関数に基づいて、参照整合性は保持され、列値は仮名に置き換えられます。サポートされるハッシュ関数は、Java Cryptography Architecture Standard Algorithm Name Documentation の MessageDigest セクション に説明されています。</p> <p>以下の例では、CzQMA0cB5K が無作為に選択された salt になります。</p> <pre>column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName</pre> <p>必要な場合は、仮名は自動的に列の長さに短縮されます。コネクター設定には、異なるハッシュアルゴリズムと salt を指定する複数のプロパティを含めることができます。</p> <p>使用される <code>hashAlgorithm</code>、選択された <code>salt</code>、および実際のデータセットによっては、結果のデータセットが完全にマスクされない場合があります。</p> <p>値が異なる場所やシステムでハッシュ化されている場合は、ハッシュ化ストラテジーバージョン 2 を使用する必要があります。</p>

プロパティ	デフォルト	説明
<code>time.precision.mode</code>	<code>adaptive</code>	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。詳細は、 一時的な値 を参照してください。
<code>decimal.handling.mode</code>	<code>precise</code>	コネクタによる DECIMAL 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に表される java.math.BigDecimal 値を使用して正確に表します。 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、簡単に使用できます。 string 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。簡単に使用できますが、本来の型に関するセマンティック情報は失われます。
<code>include.schema.changes</code>	<code>true</code>	コネクタ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と、スキーマ更新を記述する JSON 構造である値で記録されます。これは、コネクタがデータベーススキーマ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。

プロパティ	デフォルト	説明
<code>tombstones.on.delete</code>	<code>true</code>	<p>削除 イベントの後に廃棄 (tombstone) イベントが続くかどうかを制御します。</p> <p>true: 削除操作は、削除 イベントと後続の破棄 (tombstone) イベントで表されます。</p> <p>false - delete イベントのみが出力されます。</p> <p><code>log compaction</code> がトピックで有効になっている場合には、ソースレコードの削除後に廃棄 (tombstone) イベントを出力すると (デフォルト動作)、Kafka は削除された行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。</p>
<code>column.truncate.to.length.chars</code>	該当なし	<p>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)。プロパティ名の 長さ で指定された文字数を超えた場合に、一連の列のデータを切り捨てる場合は、このプロパティを設定します。length を正の整数値に設定します (例: <code>column.truncate.to.20.chars</code>)。</p> <p>列の完全修飾名は、<code><schemaName>.<tableName>.<columnName></code> の形式に従います。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>

プロパティ	デフォルト	説明
<p>column.mask.with.length.chars</p>	<p>n/a 列の完全修飾名の形式は <code>schemaName.tableName.columnName</code> です。</p>	<p>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)。一連の列の値をコネクタでマスクする場合 (たとえば、列に機密データが含まれている場合) は、このプロパティを設定します。 length を正の整数に設定して、指定された列のデータをプロパティ名の長さで指定されたアスタリスク (*) 文字数で置き換えます。指定した列のデータを空の文字列に置き換えるには、長さを 0 (ゼロ) に設定します。</p> <p>列の完全修飾名は、次の形式に従います: <code>schemaName.tableName.columnName</code> 列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p> <p>単一の設定で、異なる長さを持つ複数のプロパティを指定できます。</p>
<p>column.propagate.source.type</p>	<p>該当なし</p>	<p>列のメタデータを表す追加パラメーターをコネクタに発行させたい列の完全修飾名に一致する、オプションのコンマ区切りの正規表現のリスト。このプロパティが設定されている場合、コネクタは次のフィールドをイベントレコードのスキーマに追加します。</p> <ul style="list-style-type: none"> ● <code>__debezium.source.column.type</code> ● <code>__debezium.source.column.length</code> ● <code>__debezium.source.column.scale</code> <p>これらのパラメーターは、列の元の型名と長さ (可変幅型の場合) をそれぞれ伝達します。コネクタがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、次の形式に従います: <code>schemaName.tableName.columnName</code>。列の名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、列の名前文字列全体に対して照合されます。式は、列名に存在する可能性のある部分文字列と一致しません。</p>

プロパティ	デフォルト	説明
<p>datatype.propagate.source.type</p>	<p>該当なし</p>	<p>データベース内の列に対して定義されているデータ型の完全修飾名を指定する正規表現のオプションのコンマ区切りリスト。このプロパティが設定されている場合、データ型が一致する列に対して、コネクターはスキーマに次の追加フィールドを含むイベントレコードを発行します。</p> <ul style="list-style-type: none"> ● __debezium.source.column.type ● __debezium.source.column.length ● __debezium.source.column.scale <p>これらのパラメーターは、列の元の型名と長さ(可変幅型の場合)をそれぞれ伝達します。コネクターがこの余分なデータを発行できるようにすると、シンクデータベース内の特定の数値または文字ベースの列のサイズを適切に設定するのに役立ちます。</p> <p>列の完全修飾名は、schemaName.tableName.typeName の形式に従います。データ型の名前を照合するために、Debeziumは指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、データ型の名前文字列全体に対して照合されます。式は、型名に存在する可能性のある部分文字列と一致しません。</p> <p>SQL Server 固有のデータ型名の一覧は、SQL Server データ型マッピング を参照してください。</p>

プロパティ	デフォルト	説明
<p><code>message.key.columns</code></p>	<p>該当なし</p>	<p>指定のテーブルの Kafka トピックに公開する変更イベントレコードのカスタムメッセージキーを形成するためにコネクタが使用する列を指定する式のリスト。</p> <p>デフォルトでは、Debezium はテーブルのプライマリーキー列を、出力するレコードのメッセージキーとして使用します。デフォルトの代わりに、またはプライマリーキーのないテーブルのキーを指定するには、1つ以上の列をもとにカスタムメッセージキーを設定できます。</p> <p>テーブルにカスタムメッセージキーを設定するには、テーブルを列挙した後、メッセージキーとして使用する列を列挙します。各リストエントリは、</p> <p><fully-qualified_tableName>:<keyColumn>,<keyColumn></p> <p>の形式を取ります。複数の列名をベースにテーブルキーを作成するには、列名の間コンマを挿入します。</p> <p>各完全修飾テーブル名は、</p> <p><schemaName>.<tableName> の形式の正規表現です。</p> <p>プロパティには複数のテーブルのエントリを含めることができます。セミコロンを使用して、リスト内のテーブルエントリを区切ります。</p> <p>以下の例は、テーブル inventory.customers および purchase.orders:</p> <p>inventory.customers:pk1,pk2; (.*)purchaseorders:pk3,pk4</p> <p>のメッセージキーを設定します。テーブル inventory.customer の場合、列 pk1 と pk2 がメッセージキーとして指定されます。どのスキーマの purchaseorders テーブルでも、pk3 と pk4 の列がメッセージキーとして使用されます。</p> <p>カスタムメッセージキーの作成に使用する列の数に制限はありません。ただし、一意の鍵を指定するために必要な最小数を使用することが推奨されます。</p>

プロパティ	デフォルト	説明
binary.handling.mode	bytes	バイナリー (binary, varbinary) 列を変更イベントで表す方法を指定します。 bytes はバイナリーデータをバイト配列として表します (デフォルト)。 base64 はバイナリーデータを base64 でエンコードされた文字列として表します。 base64-url-safe は base64-url-safe でエンコードされた文字列としてバイナリーデータを表現します。 hex はバイナリーデータを 16 進エンコード (base16) 文字列として表します。
schema.name.adjustment.mode	none	コネクターで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。設定可能: <ul style="list-style-type: none"> ● none は、調整を適用しません。 ● Avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。

高度な SQL Server コネクター設定プロパティ

以下の **高度な** 設定プロパティには、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	デフォルト	説明
-------	-------	----

プロパティ	デフォルト	説明
<p>converters</p>	<p>デフォルトなし</p>	<p>コネクタが使用できる カスタムコンバーター インスタンスのシンボリック名のコンマ区切りリストを列挙します。以下に例を示します。</p> <p>isbn</p> <p>コネクタがカスタムコンバータを使用できるようにするには、converters タブプロパティを設定する必要があります。</p> <p>コネクタに設定するコンバーターごとに、コンバーターインターフェイスを実装するクラスの完全修飾名を指定する .type プロパティも追加する必要があります。.type プロパティでは、以下の形式を使用します。</p> <p><converterSymbolicName>.type</p> <p>以下に例を示します。</p> <pre> isbn.type: io.debezium.test.IsbnConverter </pre> <p>設定されたコンバータの動作をさらに制御したい場合は、1つ以上の設定パラメーターを追加して、コンバータに値を渡すことができます。追加の設定パラメーターとコンバーターを関連付けるには、パラメーター名の前にコンバーターのシンボリック名を付けます。以下に例を示します。</p> <pre> isbn.schema.name: io.debezium.sqlserver.type.Isbn </pre>

プロパティ	デフォルト	説明
<code>snapshot.mode</code>	Initial	<p>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。以下の値を使用できます。</p> <ul style="list-style-type: none"> ● initial: キャプチャーされたテーブルの構造と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。 ● initial_only: initial のように構造やデータのスナップショットを作成しますが、スナップショットの完了後に変更のストリーミングに移行しません。 ● 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ます。
<code>snapshot.include.collection.list</code>	<code>table.include.list</code> に指定したすべてのテーブル	<p>スナップショットに含めるテーブルの完全修飾名 (<code><dbName>.<schemaName>.<tableName></code>) と一致する正規表現のコンマ区切りリスト (オプション) です。指定する項目は、コネクターの <code>table.include.list</code> プロパティで名前を付ける必要があります。このプロパティは、コネクターの <code>snapshot.mode</code> プロパティが never 以外の値に設定されている場合にのみ有効です。</p> <p>このプロパティは増分スナップショットの動作には影響しません。</p> <p>テーブルの名前を一致させるために、Debezium は指定した正規表現を アンカー 正規表現として適用します。つまり、指定された式は、テーブル名に存在する可能性のある部分文字列とは一致しない、テーブルの名前文字列全体と照合されます。</p>

プロパティ	デフォルト	説明
snapshot.isolation.mode	repeatable_read	<p>使用されるトランザクション分離レベルと、キャプチャー用に指定されたテーブルをコネクターがロックする期間を制御するモード。以下の値を使用できます。</p> <ul style="list-style-type: none"> ● read_uncommitted ● read_committed ● repeatable_read ● snapshot ● exclusive (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、読み取りにはすべてのテーブルで排他ロックが必要です。) <p>snapshot、read_committed、read_uncommitted の各モードでは、最初のスナップショット中に他のトランザクションがテーブルの行を更新することができません。exclusive と repeatable_read モードでは、同時更新ができません。</p> <p>モードの選択は、データの整合性にも影響します。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</p> <p>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)。</p>
event.processing.failure.handling.mode	fail	<p>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。</p> <p>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。</p> <p>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。</p>

プロパティ	デフォルト	説明
poll.interval.ms	500	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 500 ミリ秒 (0.5 秒) です。
max.queue.size	8192	ブロッキングキューが保持できるレコードの最大数を指定する正の整数値。Debezium はデータベースからストリームされたイベントを読み込む際、Kafka に書き込む前にブロッキングキューにイベントを配置します。ブロッキングキューは、コネクターが Kafka に書き込むよりも速くメッセージを取り込む場合、または Kafka が利用できなくなった場合に、データベースから変更イベントを読み込むためのバックプレッシャーを提供することができます。コネクターがオフセットを定期的に記録すると、キューに保持されるイベントは無視されます。 max.queue.size の値を、 max.batch.size の値よりも大きくなるように設定します。
max.queue.size.in.bytes	0	ブロッキングキューの最大容量をバイト単位で指定する長整数値。デフォルトでは、ブロックキューにはボリューム制限は指定されません。キューが使用できるバイト数を指定するには、このプロパティを正の long 値に設定します。 max.queue.size も設定されている場合、キューのサイズがどちらかのプロパティで指定された上限に達すると、キューへの書き込みがブロックされます。例えば、 max.queue.size=1000 、 max.queue.size.in.bytes=5000 と設定した場合、キューに 1000 レコードが入った後、あるいはキュー内のレコードの量が 5000 バイトに達した後、キューへの書き込みがブロックされます。
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。

プロパティ	デフォルト	説明
heartbeat.interval.ms	0	<p>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。</p> <p>このプロパティには、コネクタがメッセージをハートビートピックに送信する頻度を定義する間隔(ミリ秒単位)が含まれます。このプロパティは、コネクタ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確認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。デフォルトでは無効にされています。</p>
snapshot.delay.ms	デフォルトなし	<p>コネクタの起動後、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)。</p> <p>クラスター内で複数のコネクタ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、コネクタのリバランスが実行される可能性があります。</p>
snapshot.fetch.size	2000	<p>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。</p>
query.fetch.size	デフォルトなし	<p>指定のクエリーのデータベースのラウンドトリップごとにフェッチされる行の数を指定します。デフォルトは、JDBCドライバーのデフォルトのフェッチサイズです。</p>
snapshot.lock.timeout.ms	10000	<p>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)。</p> <p>0に設定すると、コネクタ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。</p>

プロパティ	デフォルト	説明
<p>snapshot.select.statement.overrides</p>	<p>デフォルトなし</p>	<p>スナップショットに追加するテーブル行を指定します。スナップショットにテーブルの行のサブセットのみを含める場合は、プロパティを使用します。このプロパティはスナップショットにのみ影響します。コネクターがログから読み取るイベントには影響しません。</p> <p>プロパティには、<schemaName>、<tableName> の形式で完全修飾テーブル名のコンマ区切りリストが含まれます。たとえば、</p> <pre>"snapshot.select.statement.overrides": "inventory.products,customers.orders"</pre> <p>をリスト内の各テーブルに対して、スナップショットを作成する場合には、その他の設定プロパティを追加して、コネクターがテーブルで実行するように SELECT ステートメントを指定します。指定した SELECT ステートメントは、スナップショットに追加するテーブル行のサブセットを決定します。以下の形式を使用して、この SELECT ステートメントプロパティの名前 (</p> <pre>snapshot.select.statement.overrides.<schemaName>.<tableName></pre> <p>を指定します。例:</p> <pre>snapshot.select.statement.overrides.customers.orders.</pre> <p>例:</p> <p>スナップショットにソフト削除以外のレコードのみを含める場合は、soft-delete 列 (delete_flag) を含む customers.orders テーブルから、以下のプロパティを追加します。</p> <pre>"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"</pre> <p>作成されるスナップショットでは、コネクターには delete_flag = 0 のレコードのみが含まれます。</p>

プロパティ	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。詳細は、 Avro の命名 をご覧ください。
provide.transaction.meta.data	false	true に設定すると、Debezium はトランザクション境界でイベントを生成し、トランザクションメタデータでデータイベントエンベロープをエンリッチします。
retriable.restart.connector.wait.ms	10000 (10 秒)	再試行可能なエラーが発生した後にコネクターを再起動するまで待機する時間 (ミリ秒単位)。
skipped.operations	t	ストリーミング中にスキップされる oplog 操作のコンマ区切りリスト。挿入/作成は c 、更新は u 、削除は d 、切り捨ては t 、操作をスキップしない場合は none となります。デフォルトでは、省略操作はスキップされます (このコネクターによる出力ではない)。
signal.data.collection	デフォルト値なし	シグナルをコネクターに送信するために使用されるデータコレクションの完全修飾名。コレクション名の指定には、 <databaseName>.<schemaName>.<tableNameName> の形式を使用します。
incremental.snapshot.allow.schema.changes	false	<p>増分スナップショット時のスキーマの変更を許可します。有効にすると、コネクターは増分スナップショットの実行中にスキーマの変更を検出し、ロック DDL を回避するために現在のチャンクを再選択します。</p> <p>プライマリーキーへの変更はサポートされず、増分スナップショットの実行時に実行された場合には誤った結果が生じる可能性があります。もう1つの制限は、スキーマの変更が列のデフォルト値のみに影響する場合、DDL が binlog ストリームから処理されるまで変更が検出されないことです。これはスナップショットイベントの値には影響しませんが、スナップショットイベントのスキーマにはデフォルト値が古くなっている可能性があります。</p>

プロパティ	デフォルト	説明
<code>incremental.snapshot.chunk.size</code>	1024	増分スナップショットのチャンクの実行中にコネクターがメモリーを取得して読み取る行の最大数。スナップショットは、サイズが大きいスナップショットの場合にはクエリーが少なくなるため、チャンクサイズを増やすと効率が上がります。ただし、チャンクサイズが大きい場合には、スナップショットデータのバッファーにより多くのメモリーが必要になります。チャンクサイズは、環境で最適なパフォーマンスを発揮できる値に、調整します。
<code>max.iteration.transactions</code>	0	データベースの複数のテーブルからの変更をストリーミングする際に、メモリーの使用量を削減するために使用する、反復ごとの最大トランザクション数を指定します。 0 (デフォルト) に設定すると、コネクターは現在の最大 LSN を変更をフェッチする範囲として使用します。ゼロより大きい値に設定すると、コネクターはこの設定で指定された n 番目の LSN を変更をフェッチする範囲として使用します。
<code>incremental.snapshot.option.recompile</code>	false	増分スナップショット時に使用するすべての SELECT ステートメントに <code>OPTION(RECOMPILE)</code> クエリーオプションを使用します。これは、発生しうるパラメータスニффィング問題を解決するのに役立ちますが、クエリーの実行頻度によっては、ソースデータベースの CPU 負荷が増加する可能性があります。
<code>topic.naming.strategy</code>	<code>io.debezium.schema.SchemaTopicNamingStrategy</code>	データ変更、スキーマ変更、トランザクション、ハートビートイベントなどのトピック名を決定するために使用する <code>TopicNamingStrategy</code> クラスの名前。デフォルトは <code>SchemaTopicNamingStrategy</code> 。
<code>topic.delimiter</code>	.	トピック名の区切り文字を指定します。デフォルトは . です。
<code>topic.cache.size</code>	10000	トピック名を保持するために使用されるサイズ (bounded concurrent hash map)。このキャッシュは、与えられたデータコレクションに対応するトピック名を決定するのに役立つ。

プロパティ	デフォルト	説明
topic.heartbeat.prefix	<code>__debezium-heartbeat</code>	<p>コネクターがハートビートメッセージを送信するトピックの名前を制御します。トピック名のパターンは、</p> <p><code>topic.prefix.topic.transaction</code></p> <p>です。たとえば、トピックの接頭辞が fulfillment の場合は、デフォルトのトピック名は <code>__debezium-heartbeat.fulfillment</code> になります。</p>
topic.transaction	<code>transaction</code>	<p>コネクターがトランザクションのメタデータメッセージを送信するトピックの名前を制御します。トピック名のパターンは、</p> <p><code>topic.prefix.topic.transaction</code></p> <p>です。たとえば、トピックの接頭辞が fulfillment の場合、デフォルトのトピック名は <code>fulfillment.transaction</code> になります。</p> <p>詳しくは、トランザクションメタデータ を参照してください。</p>

Debezium SQL Server コネクターデータベーススキーマ履歴の設定プロパティ

Debezium には、コネクターがスキーマ履歴トピックと対話する方法を制御する `schema.history.internal.*` プロパティのセットが含まれています。

以下の表は、Debezium コネクターを設定するための `schema.history.internal` プロパティについて説明しています。

表8.17 コネクターデータベーススキーマ履歴の設定プロパティ

プロパティ	デフォルト	説明
schema.history.internal.kafka.topic	デフォルトなし	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。
schema.history.internal.kafka.bootstrap.servers	デフォルトなし	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。この接続は、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。各ペアは、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	デフォルト	説明
<code>schema.history.internal.kafka.a.recovery.poll.interval.ms</code>	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。
<code>schema.history.internal.kafka.a.query.timeout.ms</code>	3000	Kafka 管理クライアントを使用してクラスター情報を取得する際に、コネクターが待機すべき最大ミリ秒数を指定する整数値です。
<code>schema.history.internal.kafka.a.create.timeout.ms</code>	30000	Kafka 管理クライアントを使用して kafka 履歴トピックを作成する間、コネクターが待機する最大ミリ秒数を指定する整数値。
<code>schema.history.internal.kafka.a.recovery.attempts</code>	100	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 × recovery.poll.interval.ms です。
<code>schema.history.internal.skip.unparseable.ddl</code>	false	コネクターが不正または不明なデータベースのステートメントを無視するかどうか、または人が問題を修正するために処理を停止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 の処理中にデータの損失や分割を引き起こす可能性があるため、必ず注意して使用する必要があります。
<code>schema.history.internal.store.only.captured.tables.ddl</code>	false	コネクターがすべての DDL ステートメントを記録するかどうかを指定するブール値 true は、変更が Debezium によってキャプチャーされるテーブルに関連する DDL ステートメントのみを記録します。変更がキャプチャーされたテーブルを変更すると、不足しているデータが必要になる可能性があるため、 true に設定するには注意が必要です。 安全なデフォルトは false です。

プロデューサーおよびコンシューマクライアントを設定するためのパススルーデータベーススキーマ履歴プロパティ

Debezium は、Kafka プロデューサーを使用して、データベーススキーマ履歴トピックにスキーマの変更を書き込みます。同様に、コネクターが起動すると、データベーススキーマ履歴トピックから読み取る Kafka コンシューマーに依存します。 **schema.history.internal.producer.*** および **schema.history.internal.consumer.*** 接頭辞で始まるパススルー設定プロパティのセットに値を割り当てて、Kafka プロデューサーおよびコンシューマクライアントの設定を定義します。パススルー

ロデューサーおよびコンシューマーデータベーススキーマ履歴プロパティは、以下の例のように Kafka ブローカーとのこれらのクライアントの接続をセキュアにする方法など、さまざまな動作を制御します。

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

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

Debezium は、プロパティを Kafka クライアントに渡す前に、プロパティ名から接頭辞を削除します。

[Kafka プロデューサー設定プロパティ](#) および [Kafka コンシューマー設定プロパティ](#) の詳細は、Kafka のドキュメントを参照してください。

Debezium SQL Server コネクターススルーデータベースドライバー設定プロパティ

Debezium コネクタースでは、データベースドライバーのパススルー設定が可能です。パススルーデータベースプロパティは接頭辞 **driver.*** で始まります。たとえば、コネクタースは **driver.foofoo=false** などのプロパティを JDBC URL に渡します。

[データベーススキーマ履歴クライアントのパススループロパティ](#) の場合のように、Debezium はプロパティから接頭辞を削除してからデータベースドライバーに渡します。

8.5. スキーマ変更後のキャプチャーテーブルの更新

SQL Server テーブルに対して変更データキャプチャー (CDC) が有効になっている場合、テーブルで変更が行われると、イベントレコードがサーバーのキャプチャーテーブルに永続化されます。たとえば、新しい列を追加するなどして、ソーステーブル変更の構造に変更を加えても、その変更は変更テーブルに動的に反映されません。キャプチャーテーブルが古いスキーマを使用し続ける限り、Debezium コネクタースはテーブルのデータ変更イベントを正しく出力できません。コネクタースが変更イベントの処理を再開できるようにするには、介入してキャプチャーテーブルを更新する必要があります。

CDC を SQL Server に実装する方法により、Debezium を使用してキャプチャーテーブルを更新することはできません。キャプチャーテーブルを更新するには、1つが昇格された権限を持つ SQL Server データベースオペレーターである必要があります。Debezium ユーザーとして、SQL Server データベース operator とタスクを調整して、スキーマの更新を完了し、Kafka トピックへのストリーミングを復元する必要があります。

以下の方法のいずれかを使用すると、スキーマの変更後にキャプチャーテーブルを更新できます。

- [オフラインでスキーマ](#) を更新するには、キャプチャーテーブルを更新する前に Debezium コネクタースを停止する必要があります。
- [オンラインスキーマの更新](#) は、Debezium コネクタースの実行中にキャプチャーテーブルを更新できます。

各手順には長所と短所があります。



警告

オンライン更新またはオフライン更新のどちらを使用する場合でも、同じソーステーブルに後続のスキーマ更新を適用する前に、スキーマ更新プロセス全体を完了する必要があります。手順を一度に実行できるようにするため、すべての DDL を 1 つのバッチで実行することがベストプラクティスとなります。



注記

CDC が有効になっているソーステーブルでは、一部のスキーマの変更がサポートされていません。たとえば、CDC がテーブルで有効になっている場合、SQL Server で列の名前を変更したり、列型を変更すると、テーブルのスキーマを変更できません。



注記

ソーステーブルの列を **NULL** から **NOT NULL** またはその逆に変更した後、SQL Server コネクターは新しいキャプチャーインスタンスが作成されるまで変更された情報を正しくキャプチャーできません。列指定の変更後に新しいキャプチャーテーブルを作成しないと、コネクターによって出力される変更イベントレコードは列が任意であるかどうかを正しく示しません。つまり、以前はオプション (または **NULL**) として定義されていた列が、現在は **NOT NULL** として定義されているにもかかわらず、引き続きオプションとして定義されているということです。同様に、必要 (**NOT NULL**) として定義された列は、**NULL** として定義されても、以前の指定が保持されます。



注記

sp_rename 関数を使用してテーブルの名前を変更すると、コネクターが再起動されるまで、古いソーステーブル名で変更が発行され続けます。コネクターを再起動すると、新しいソーステーブル名で変更が発行されます。

8.5.1. スキーマの変更後のオフライン更新の実行

オフラインでスキーマを更新すると、最も安全にキャプチャーテーブルを更新できます。ただし、オフラインでの更新は、高可用性を必要とするアプリケーションでは使用できないことがあります。

前提条件

- CDC が有効になっている SQL Server テーブルのスキーマに更新がコミット済みである。
- 昇格された権限を持つ SQL Server データベース operator である。

手順

1. データベースを更新するアプリケーションを一時停止します。
2. Debezium コネクターがストリーミングされていない変更イベントレコードをすべてストリーミングするまで待ちます。

3. Debezium コネクタを停止します。
4. すべての変更をソーステーブルスキーマに適用します。
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. ステップ1で一時停止したアプリケーションを再開します。
7. Debezium コネクタを起動します。
8. Debezium コネクタ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したら、古いキャプチャーインスタンス名に設定されたパラメーター `@capture_instance` でストアードプロシージャ `sys.sp_cdc_disable_table` を実行して、古いキャプチャーテーブルを削除します。

8.5.2. スキーマの変更後のオンライン更新の実行

オンラインでスキーマの更新を完了する手順は、オフラインでスキーマの更新を実行する手順よりも簡単です。また、アプリケーションやデータ処理のダウンタイムなしで完了できます。ただし、オンラインでスキーマを更新すると、ソースデータベースでスキーマを更新した後、新しいキャプチャーインスタンスを作成するまでに、処理の差が生じる可能性があります。この間、変更イベントは変更テーブルの古いインスタンスによって引き続きキャプチャーされ、古いテーブルに保存された変更データは、以前のスキーマの構造を保持します。たとえば、新しい列をソーステーブルに追加した場合は、新しいキャプチャーテーブルの準備が整う前に生成された変更イベントには新しい列のフィールドは含まれません。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、オフラインでスキーマの更新を行うことが推奨されます。

前提条件

- CDC が有効になっている SQL Server テーブルのスキーマに更新がコミット済みである。
- 昇格された権限を持つ SQL Server データベース operator である。

手順

1. すべての変更をソーステーブルスキーマに適用します。
2. パラメーター `@capture_instance` に一意の値を指定して `sys.sp_cdc_enable_table` ストアドプロシージャを実行し、更新元テーブルに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のストリーミングを開始したら、パラメーター `@capture_instance` に古いキャプチャーインスタンス名を設定して、`sys.sp_cdc_disable_table` ストアドプロシージャを実行することで、古いキャプチャーテーブルを削除することができます。

例: データベーススキーマの変更後のオンラインスキーマ更新の実行

次の例は、`customers` ソーステーブルに `phone_number` 列が追加された後、`change` テーブルでオンラインスキーマ更新を完了する方法を示しています。

1. 次のクエリーを実行して `customers` ソーステーブルのスキーマを変更し、`phone_number` フィールドを追加します。

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

2. **sys.sp_cdc_enable_table** ストアドプロシージャを実行して、新しいキャプチャーインスタンスを作成します。

```
EXEC sys.sp_cdc_enable_table @source_schema = 'dbo', @source_name = 'customers',
@role_name = NULL, @supports_net_changes = 0, @capture_instance =
'dbo_customers_v2';
GO
```

3. 次のクエリーを実行して、**customers** テーブルに新しいデータを挿入します。

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

Kafka Connect ログは、以下のメッセージのようなエントリで設定の更新を報告します。

```
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]
[jio.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]
[jio.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]
[jio.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
```

最終的には、**phone_number** フィールドがスキーマに追加され、その値が Kafka トピックに書き込まれたメッセージに表示されます。

```
...
{
  "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"
},
```

4. **sys.sp_cdc_disable_table** ストアドプロシージャーを実行して、古いキャプチャーインスタンスを削除します。

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

8.6. DEBEZIUM SQL SERVER コネクタのパフォーマンスの監視

Debezium SQL Server コネクタは、Zookeeper、Kafka、および Kafka Connect によって提供される JMX メトリクスの組み込みサポートに加えて、3種類のメトリクスを提供します。コネクタは以下のメトリクスを提供します。

- スナップショットの実行時にコネクタを監視するための、[スナップショットメトリクス](#)。
- CDC テーブルデータの読み取り時にコネクタを監視するための、[ストリーミングメトリクス](#)。
- コネクタ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](#)。

JMX 経由で前述のメトリクスを公開する方法については、[Debezium の監視に関するドキュメント](#) を参照してください。

8.6.1. Debezium SQL Server コネクタのスナップショットメトリクス

MBean は `debezium.sql_server:type=connector-metrics,server=<sqlserver.server.name>,task=<task.id>,context=snapshot` です。

スナップショット操作がアクティブでない場合や、最後のコネクタの起動後にスナップショットの作成が発生した場合に、スナップショットメトリクスは公開されません。

以下の表は、利用可能なスナップショットのメトリックの一覧です。

属性	タイプ	説明
LastEvent	string	コネクタ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で確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクタに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。

属性	タイプ	説明
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	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。
QueueRemainingCapacity	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。
SnapshotPaused	boolean	スナップショットが一時停止されたかどうか。
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)。スナップショットが一時停止された時間も含まれます。
SnapshotPausedDurationInSeconds	long	スナップショットが一時停止された合計秒数。スナップショットが数回一時停止された場合は、一時停止時間が加算されます。

属性	タイプ	説明
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せん。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは、 max.queue.size.in.bytes が正の長さの値に設定されている場合に利用できます。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

コネクタは、増分スナップショットの実行時に、以下の追加のスナップショットメトリクスも提供します。

属性	タイプ	説明
ChunkId	string	現在のスナップショットチャンクの識別子。
ChunkFrom	string	現在のチャンクを定義するプライマリーキーセットの下限。
ChunkTo	string	現在のチャンクを定義するプライマリーキーセットの上限。
TableFrom	string	現在スナップショットされているテーブルのプライマリーキーセットの下限。
TableTo	string	現在スナップショットされているテーブルのプライマリーキーセットの上限。

8.6.2. Debezium SQL Server コネクタのストリーミングメトリクス

MBean は `debezium.sql_server:type=connector-metrics,server=<sqlserver.server.name>,task=<task.id>,context=streaming` です。

以下の表は、利用可能なストリーミングメトリクスの一覧です。

属性	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	このコネクターが前回の起動またはメトリックリセット以降に見たイベントの合計数。
TotalNumberOfCreateEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た、作成イベントの合計数。
TotalNumberOfUpdateEventsSeen	long	最後の起動またはメトリックリセット以降にこのコネクターが見た更新イベントの合計数。
TotalNumberOfDeleteEventsSeen	long	このコネクターが最後に起動またはメトリックリセットされてから見た削除イベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルタリングルールによってフィルターされたイベントの数。
CapturedTables	string[]	コネクターによって取得されるテーブルの一覧。
QueueTotalCapacity	int	ストリーマーとメイン 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。
QueueRemainingCapacity	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。
Connected	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。

属性	タイプ	説明
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との間の期間 (ミリ秒単位)。この値は、データベースサーバーとコネクタが稼働しているマシンのクロック間の差異に対応します。
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。
MaxQueueSizeInBytes	long	キューの最大バッファ (バイト単位)。このメトリクスは、 max.queue.size.in.bytes が正の長さの値に設定されている場合に利用できます。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在の容量 (バイト単位)。

8.6.3. Debezium SQL Server コネクタのスキーマ履歴メトリクス

MBean は、`debezium.sql_server:type=connector-metrics,context=schema-history,server=<sqlserver.server.name>` です。

以下の表は、利用可能なスキーマ履歴メトリクスの一覧です。

属性	タイプ	説明
Status	string	データベーススキーマ履歴の状態を示す STOPPED 、 RECOVERING (ストレージから履歴を復元)、または 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。

属性	タイプ	説明
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。

第9章 DEBEZIUM の監視

Apache Zookeeper、Apache Kafka、および Kafka Connect が提供する 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。

9.1. DEBEZIUM コネクタを監視するためのメトリクス

Kafka、Zookeeper、および Kafka Connect に組み込まれた JMX メトリクスのサポートに加えて、それぞれのコネクタには動作を監視するための追加のメトリクスが用意されています。

- [MySQL コネクタメトリクス](#)
- [MongoDB コネクタメトリクス](#)
- [Postgre SQL コネクタのメトリクス](#)
- [SQL Server コネクタメトリクス](#)
- [Db2 コネクタメトリクス](#)

9.2. ローカルインストールでの JMX の有効化

Zookeeper、Kafka、および Kafka Connect では、各サービスの起動時に適切な環境変数を設定して JMX を有効にします。

9.2.1. 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。

9.2.2. 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**

9.2.3. 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**

9.3. OPENSIFT 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、OpenShift における AMQ Streams の使用で [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、Open Shift での AMQ ストリームのデプロイとアップグレードの [Kafka へのメトリクスの紹介](#) を参照してください。

第10章 DEBEZIUM のログ機能

Debezium のコネクタには、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。

10.1. DEBEZIUM ログの概念

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、固有の **ログレベル** も設定される。

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 (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 とは、基本的にログメッセージの書き込み先を指します。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。

10.2. デフォルトの DEBEZIUM ログ設定

Kafka Connect プロセスで Debezium コネクタ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 ① ① ① ① ① ① ① ① ① ① ① ① ① 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ています。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 ② ② ② ② ② ② ② ② ② ② ② ② ② 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 ③ ③ ③ ③ ③ ③ ③ ③ ③ ③ ③ ③ ③ ③ **stdout** アペンダーは、パターンの照合アルゴリズムを使用してログメッセージをフォーマットします。

④ ④ ④ ④ ④ ④ ④ ④ ④ ④ ④ ④ ④ ④ 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 が使用するすべてのロガーは **rootLogger** 設定を継承します。

10.3. DEBEZIUM ログの設定

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。次のいずれかの方法を使用して、デフォルトのログ設定を変更できます。

- [ロガーを設定してログレベルを設定する](#)
- [Kafka Connect REST API でのロギングレベルの動的な設定](#)
- [マップされた診断コンテキストを追加してログレベルを設定する](#)



注記

他の方法を使用して、Log4j による Debezium のログを設定することができます。詳細については、アペンダーを設定および使用し、特定の宛先にログメッセージを送信する方法についてのチュートリアルを検索してください。

10.3.1. ロガーを設定して Debezium の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。

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スキーマ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 ❶
log4j.logger.io.debezium.relational.history=DEBUG, stdout ❷

log4j.additivity.io.debezium.connector.mysql=false ❸
log4j.additivity.io.debezium.storage.kafka.history=false ❹
...
```

- ❶ **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。
- ❷ **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。
- ❸ ❹ **additivity** を無効にします。これにより、ログメッセージが親ロガーのアペンダーに送信されなくなります (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表示され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.storage.kafka.history=false
log4j.additivity.io.debezium.connector.mysql.BinlogReader=false
...
```

10.3.2. Kafka Connect API を使用して Debezium のログレベルを動的に変更する

Kafka Connect REST API を使用して、実行時にコネクタのログレベルを動的に設定できます。**log4j.properties** で設定されるログレベルの変更とは異なり、API 経由で行った変更は即座に反映されるため、ワーカーを再起動する必要はありません。

API に指定するログレベル設定は、要求を受信するエンドポイントのワーカーにのみ適用されます。クラスター内の他のワーカーのログレベルは変更されません。

指定されたレベルは、ワーカーの再起動後に維持されません。ロギングレベルを永続的に変更するには、[ロガーを設定するか](#)、[マップされた診断コンテキストを追加](#)して **log4j.properties** でログレベルを設定します。

手順

- 以下の情報を指定する **admin/loggers** エンドポイントに PUT 要求を送信してログレベルを設定します。
 - ログレベルを変更するパッケージ。
 - 設定するログレベル。

```
curl -s -X PUT -H "Content-Type:application/json"
http://localhost:8083/admin/loggers/io.debezium.connector.<connector_package> -d
{"level": "<log_level>"}
```

たとえば、Debezium MySQL コネクタのデバッグ情報をログに記録するには、以下のリクエストを Kafka Connect に送信します。

```
curl -s -X PUT -H "Content-Type:application/json"
http://localhost:8083/admin/loggers/io.debezium.connector.mysql -d {"level": "DEBUG"}
```

10.3.3. マッピングされた診断コンテキストを追加して Debezium のロギングレベルの変更

ほとんどの Debezium コネクタ（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る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 プロパティを利用することができます。

dbz.connectorType

コネクタ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によって生成されたすべてのログメッセージを探ることができます。

dbz.connectorName

コネクタの設定で定義されているコネクタ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ることができます。

dbz.connectorContext

コネクタ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ることができます。

コネクタの 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 プロパティを使用するように **stdout** アペンダーを設定します。

log4j.properties

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

前述の例の設定では、以下の出力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&useUnicode=true
&characterEncoding=UTF-8&characterSetResults=UTF-
8&zeroDateTimeBehavior=convertToNull with user 'debezium'
[jio.debezium.connector.mysql.SnapshotReader]
2017-02-07 20:49:37,696 INFO MySQL|dbserver1|snapshot Snapshot is using user
'debezium' with these MySQL grants: [jio.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'@%' [jio.debezium.connector.mysql.SnapshotReader]
...
```

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

10.4. OPENSIFT 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。Kafka Connect スキーマでのロギングプロパティの設定に関する詳細は、[Using AMQ Streams on OpenShift](#) を参照してください。

第11章 アプリケーション用 DEBEZIUM コネクタの設定

Debezium コネクタのデフォルトの動作がアプリケーションに適していない場合、以下の Debezium 機能を使用して必要な動作を設定することができます。

Kafka Connect 自動トピック作成

Connect が実行時にトピックを作成し、トピックの名前に基づいて設定設定をトピックに適用するのを許可します。

Avro シリアライゼーション

Debezium PostgreSQL、MongoDB、または SQL Server コネクタが Avro を使用してメッセージのキーと値をシリアライズする設定をサポートします。これにより、変更イベントレコードのコンシューマーがレコードスキーマの変更に容易に適応できるようにします。

CloudEvents コンバーター

Debezium コネクタが CloudEvents 仕様に準拠する変更イベントレコードを出力できるようにします。

Debezium コネクタへのシグナル送信

コネクタの動作を変更したり、アドホックスナップショットの開始などのアクションをトリガーしたりする方法を提供します。

11.1. KAFKA CONNECT 自動トピック作成のカスタマイズ

Kafka には、トピックを自動的に作成するメカニズムが2つ用意されています。Kafka ブローカーの自動トピック作成を有効にすることができます。また、Kafka 2.6.0 以降では、Kafka Connect のトピック作成を有効にすることもできます。Kafka ブローカーは、**auto.create.topics.enable** プロパティを使用してトピックの自動作成を制御します。Kafka Connect では、**topic.creation.enable** プロパティで、Kafka Connect がトピックを作成することを許可するかどうかを指定します。いずれの場合も、プロパティのデフォルト設定ではトピックの自動作成が有効です。

トピックの自動作成が有効な場合、Debezium ソースコネクタがまだルーティング先トピックが存在しないテーブルの変更イベントレコードを出力すると、イベントレコードが Kafka に取り込まれる際にトピックが実行時に作成されます。

ブローカーと Kafka Connect での自動トピック作成の違い

ブローカーが作成するトピックは、1つのデフォルト設定の共有に制限されます。ブローカーは、異なるトピックまたはトピックのセットに一意の設定を適用することはできません。対照的に、Kafka Connect では、トピックの作成時に任意のさまざまな設定を適用し、Debezium コネクタ設定で指定したレプリケーション係数、パーティション数、およびその他のトピック固有の設定を定義することができます。コネクタ設定はトピック作成グループのセットを定義し、トピック設定のプロパティセットを各グループに関連付けます。

ブローカー設定と Kafka Connect 設定は、互いに独立しています。ブローカーでトピック作成を無効にしたかどうかに関係なく、Kafka Connect はトピックを作成することができます。ブローカーと Kafka Connect の両方でトピックの自動作成を有効にすると Connect の設定が優先され、Kafka Connect のいずれの設定も適用されない場合に限り、ブローカーはトピックを作成します。

詳細については、以下のトピックを参照してください。

- [「Kafka ブローカーの自動トピック作成の無効化」](#)
- [「Kafka Connect の自動トピック作成の設定」](#)
- [「自動的に作成されたトピックの設定」](#)

- [「トピック作成グループ」](#)
- [「トピック作成グループの設定プロパティ」](#)
- [「Debezium デフォルトトピック作成グループ設定の指定」](#)
- [「Debezium カスタムトピック作成グループ設定の指定」](#)
- [「Debezium カスタムトピック作成グループの登録」](#)

11.1.1. Kafka ブローカーの自動トピック作成の無効化

デフォルトでは、トピックがまだ存在しない場合、Kafka ブローカー設定によりブローカーは実行時にトピックを作成することができます。ブローカーによって作成されたトピックにカスタムプロパティを設定することはできません。2.6.0 より前のバージョンの Kafka を使用し、特定の設定でトピックを作成する場合は、ブローカーの自動トピック作成を無効にし、手動またはカスタムデプロイプロセスのいずれにより明示的にトピックを作成する必要があります。

手順

- ブローカーの設定で、`auto.create.topics.enable` の値を `false` にします。

11.1.2. Kafka Connect の自動トピック作成の設定

Kafka Connect でのトピックの自動作成は、`topic.creation.enable` プロパティによって制御されます。次の例に示すように、プロパティのデフォルト値は `true` であり、トピックの自動作成を有効にします。

```
topic.creation.enable = true
```

`topic.creation.enable` プロパティの設定は、Connect クラスター内のすべてのワーカーに適用されます。

Kafka Connect の自動トピック作成では、トピックの作成時に Kafka Connect が適用する設定プロパティを定義する必要があります。トピックグループを定義して Debezium コネクター設定でトピックの設定プロパティを指定し、続いてそれぞれのグループに適用するプロパティを指定します。コネクター設定では、デフォルトのトピック作成グループ、およびオプションで1つまたは複数のカスタムトピック作成グループを定義します。カスタムトピック作成グループは、トピック名パターンのリストを使用してグループの設定が適用されるトピックを指定します。

Kafka Connect がどのようにトピックをトピック作成グループと照合するかについての詳細は、[トピック作成グループ](#) を参照してください。設定プロパティがどのようにグループに割り当てられるかについての詳細は、[トピック作成グループの設定プロパティ](#) を参照してください。

デフォルトでは、Kafka Connect が作成するトピックは、パターン `server.schema.table` に基づいて名前が付けられます (例: `dbserver.myschema.inventory`)。

手順

- Kafka Connect がトピックを自動的に作成しないようにするには、次の例のように、**Kafka Connect** カスタムリソースで `topic.creation.enable` の値を `false` に設定します。

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
```

```
name: my-connect-cluster
...
spec:
  config:
    topic.creation.enable: "false"
```



注記

Kafka Connect の自動トピック作成では、**replication.factor** プロパティと **partitions** プロパティを少なくとも **default** のトピック作成グループに設定する必要があります。グループは、Kafka ブローカーのデフォルト値から必要なプロパティの値を取得することができます。

11.1.3. 自動的に作成されたトピックの設定

Kafka Connect でトピックを自動的に作成するには、トピックの作成時に適用する設定プロパティに関するソースコネクタからの情報が必要です。それぞれの Debezium コネクタの設定で、トピックの作成を制御するプロパティを定義します。Kafka Connect がコネクタから出力されるイベントレコード用のトピックを作成すると、作成されるトピックは該当するグループから設定を取得します。この設定は、そのコネクタによって出力されたイベントレコードにのみ適用されます。

11.1.3.1. トピック作成グループ

トピックプロパティのセットが、トピック作成グループに関連付けられます。少なくとも **default** トピック作成グループを定義し、その設定プロパティを指定する必要があります。それ以外に、オプションとして1つまたは複数のカスタムトピック作成グループを定義し、それぞれに一意のプロパティを指定することができます。

カスタムトピック作成グループを作成する場合、トピック名パターンに基づいて各グループにメンバートピックを定義します。各グループに含めるトピックまたはグループから除外するトピックを記述する命名パターンを指定することができます。**include** および **exclude** プロパティには、トピック名パターンを定義する正規表現のコンマ区切りリストを指定します。たとえば、文字列 **dbserver1.inventory** で始まるすべてのトピックをグループに含める場合は、その **topic.creation.inventory.include** プロパティの値を **dbserver1\\.inventory\\.*** に設定します。



注記

カスタムトピックグループに **include** および **exclude** プロパティの両方を指定すると、除外ルールが優先され包含ルールがオーバーライドされます。

11.1.3.2. トピック作成グループの設定プロパティ

default トピック作成グループおよびそれぞれのカスタムグループは、一意の設定プロパティのセットに関連付けられます。任意の **Kafka トピックレベルの設定プロパティ** をグループの設定に含めることができます。たとえば、トピックグループに **古いトピックセグメントのクリーンアップポリシー**、**保持時間**、または **トピックの圧縮タイプ** を指定することができます。少なくとも、作成するトピックの設定を記述するプロパティの最小セットを定義する必要があります。

カスタムグループが登録されていない場合、または登録されているグループの **include** パターンが作成するトピックの名前とマッチしない場合、Kafka Connect は **default** グループの設定を使用してトピックを作成します。

トピック設定についての概要は、Debezium の OpenShift へのインストールの [Kafka トピック作成に関する推奨事項](#) を参照してください。

11.1.3.3. Debezium デフォルトトピック作成グループ設定の指定

Kafka Connect の自動トピック作成を使用するためには、デフォルトのトピック作成グループを作成し、その設定を定義する必要があります。デフォルトのトピック作成グループの設定は、カスタムトピック作成グループの **include** リストのパターンにマッチしない名前のすべてのトピックに適用されません。

前提条件

- Kafka Connect のカスタムリソースで、**metadata.annotations** の **use-connector-resources** の値により、クラスターの Operator が KafkaConnector カスタムリソースを使用してクラスター内のコネクタを設定するように指定されている。以下に例を示します。

```
...
  metadata:
    name: my-connect-cluster
    annotations: strimzi.io/use-connector-resources: "true"
  ...
```

手順

- **topic.creation.default** グループのプロパティを定義するには、以下の例に示すように、コネクタのカスタムリソースの **spec.config** にプロパティを追加します。

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

  config:
  ...
    topic.creation.default.replication.factor: 3 1
    topic.creation.default.partitions: 10 2
    topic.creation.default.cleanup.policy: compact 3
    topic.creation.default.compression.type: lz4 4
  ...
```

任意の [Kafka トピックレベルの設定プロパティ](#) を **default** グループの設定に含めることができます。

表11.1 default のトピック作成グループのコネクタ設定

項目	説明
----	----

項目	説明
1	<p>topic.creation.default.replication.factor は、デフォルトグループによって作成されるトピックのレプリケーション係数を定義します。</p> <p>replication.factor グループの場合 default の設定は必須ですが、カスタムグループの場合は任意です。カスタムグループは、設定されていない場合、default グループの値にフォールバックします。Kafka ブローカーのデフォルト値を使用する場合は -1 を使用します。</p>
2	<p>topic.creation.default.partitions は、デフォルトグループによって作成されるトピックのパーティション数を定義します。</p> <p>default グループの場合 partitions の設定は必須ですが、カスタムグループの場合は任意です。カスタムグループは、設定されていない場合、default グループの値にフォールバックします。Kafka ブローカーのデフォルト値を使用する場合は -1 を使用します。</p>
3	<p>topic.creation.default.cleanup.policy は トピックレベルの設定パラメーターの cleanup.policy プロパティにマッピングされ、ログの保存ポリシーを定義します。</p>
4	<p>topic.creation.default.compression.type は、 トピックレベルの設定パラメーターの compression.type プロパティにマッピングされており、メッセージをハードディスク上でどのように圧縮するかを定義します。</p>



注記

カスタムグループは、必要な **replication.factor** および **partitions** プロパティのみに対して、**default** グループの設定が戻ります。カスタムトピックグループ設定の他のプロパティが定義されていない場合、**default** グループで指定された値は適用されません。

11.1.3.4. Debezium カスタムトピック作成グループ設定の指定

複数のカスタムトピックグループを、それぞれ個別の設定で定義することができます。

手順

- カスタムトピックグループを定義するには、コネクタのカスタムリソースの **spec.config** に **topic.creation.<group_name>.include** プロパティを追加し、続いてカスタムグループのトピックに適用する設定プロパティを定義します。
 次の例では、カスタムトピック作成グループ **inventory** と **applicationlogs** を定義するカスタムリソースの抜粋を示しています。

```

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

config:
... 1
  topic.creation.inventory.include: dbserver1\\inventory\\.* 2
  topic.creation.inventory.partitions: 20

```

```
topic.creation.inventory.cleanup.policy: compact
topic.creation.inventory.delete.retention.ms: 7776000000
```

3

```
topic.creation.applicationlogs.include: dbserver1\\.logs\\.applog-* 4
topic.creation.applicationlogs.exclude": dbserver1\\.logs\\.applog-old-* 5
topic.creation.applicationlogs.replication.factor: 1
topic.creation.applicationlogs.partitions: 20
topic.creation.applicationlogs.cleanup.policy: delete
topic.creation.applicationlogs.retention.ms: 7776000000
topic.creation.applicationlogs.compression.type: lz4
```

...

...

表11.2 inventory および applicationlogs カスタムトピック作成グループのコネクター設定

項目	説明
1	inventory グループの設定を定義します。 カスタムグループでは、 replication.factor および partitions プロパティはオプションです。値が設定されていない場合、カスタムグループは、 default グループに設定されている値にフォールバックします。Kafka ブローカーに設定されている値を使用する場合は、 -1 に設定します。
2	topic.creation.inventory.include は、 dbserver1.inventory. で始まるすべてのトピックにマッチする正規表現を定義します。 inventory グループに定義された設定は、指定した正規表現にマッチする名前のトピックにのみ適用されます。
3	applicationlogs グループの設定を定義します。 カスタムグループでは、 replication.factor および partitions プロパティはオプションです。値が設定されていない場合、カスタムグループは、 default グループに設定されている値にフォールバックします。Kafka ブローカーに設定されている値を使用する場合は、 -1 に設定します。
4	topic.creation.applicationlogs.include では、 dbserver1.logs.applog- で始まるすべてのトピックにマッチする正規表現を定義します。 applicationlogs グループに定義された設定は、指定された正規表現にマッチする名前のトピックにのみ適用されます。このグループには、 exclude プロパティも定義されているため、 include 正規表現に一致するトピックは、 exclude プロパティによってさらに制限される可能性があります。
5	topic.creation.applicationlogs.exclude では、 dbserver1.logs.applog-old- で始まるすべてのトピックに一致する正規表現を定義します。 applicationlogs グループに定義された設定は、指定された正規表現にマッチしない名前のトピックにのみ適用されます。このグループには include プロパティも定義されているため、 applicationlogs グループの設定は、指定された include 正規表現にマッチし、指定された exclude 正規表現にマッチしない名前のトピックにのみ適用されます。

11.1.3.5. Debezium カスタムトピック作成グループの登録

カスタムトピック作成グループの設定を指定したら、グループを登録します。

手順

- カスタムグループを登録するには、コネクタのカスタムリソースに **topic.creation.groups** プロパティを追加し、カスタムトピック作成グループをコンマで区切って指定します。カスタムトピック作成グループ **inventory** と **applicationlogs** を登録するコネクタカスタムリソースの抜粋を以下に示します。

```

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

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

```

設定の完了

default トピックグループの設定に加えて **inventory** および **applicationlogs** カスタムトピック作成グループの設定が含まれる完了した設定の例を以下に示します。

例: デフォルトのトピック作成グループおよび2つのカスタムグループの設定

```

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

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

```

11.2. AVRO シリアライゼーションを使用する DEBEZIUM コネクタの設定

Debezium コネクタは Kafka Connect のフレームワークで動作し、変更イベントレコードを生成することでデータベース内の各行レベルの変更をキャプチャーします。それぞれの変更イベントレコードについて、Debezium コネクタは以下のアクションを完了します。

1. 設定された変換を適用する。
2. 設定された [Kafka Connect コンバーター](#) を使用して、レコードのキーと値をバイナリー形式にシリアル化する。
3. レコードを正しい Kafka トピックに書き込む。

個々の Debezium コネクタインスタンスごとにコンバーターを指定することができます。Kafka Connect は、レコードのキーと値を JSON ドキュメントにシリアル化する JSON コンバーターを提供します。デフォルトの動作では、JSON コンバーターはレコードのメッセージスキーマを含めるので、それぞれのレコードが非常に冗長になります。[Debezium スタートガイド](#) に、ペイロードとスキーマの両方が含まれる場合にレコードがどのように見えるかが示されています。レコードを JSON でシリアル化したい場合は、以下のコネクタ設定プロパティを **false** に設定することを検討してください。

- **key.converter.schemas.enable**
- **value.converter.schemas.enable**

これらのプロパティを **false** に設定すると、冗長なスキーマ情報がそれぞれのレコードから除外されます。

あるいは、[Apache Avro](#) を使用してレコードのキーと値をシリアル化することもできます。Avro のバイナリー形式はコンパクトで効率的です。Avro スキーマを使用すると、それぞれのレコードが正しい構造を持つようにすることができます。Avro のスキーマ進化メカニズムにより、スキーマを進化させることが可能です。変更されたデータベーステーブルの構造と一致するように各レコードのスキーマを動的に生成するこのメカニズムは、Debezium コネクタに不可欠です。時間の経過と共に、同じ Kafka トピックに書き込まれる変更イベントレコードが、同じスキーマの別バージョンとなる場合があります。Avro シリアルイゼーションを使用すると、変更イベントレコードのコンシューマーはレコードスキーマの変化に対応することができます。

Apache Avro シリアルイゼーションを使用するには、Avro メッセージスキーマおよびそのバージョンを管理するスキーマレジストリーをデプロイする必要があります。このレジストリーの設定については、[OpenShift でのサービスレジストリーのインストールとデプロイ](#) のドキュメントを参照してください。

11.2.1. Service Registry の概要

サービスレジストリー

[Service Registry](#) は、Avro と共に動作する以下のコンポーネントを提供します。

- Debezium コネクタ設定で指定することができる Avro コンバーター。このコンバーターは、Kafka Connect スキーマを Avro スキーマにマッピングします。続いて、コンバーターは Avro スキーマを使用してレコードのキーと値を Avro のコンパクトなバイナリー形式にシリアル化します。
- API および以下の項目を追跡するスキーマレジストリー。
 - Kafka トピックで使用される Avro スキーマ
 - Avro コンバーターが生成した Avro スキーマを送信する先

Avro スキーマはこのレジストリーに保管されるため、各レコードには小さなスキーマ識別子だけを含める必要があります。これにより、各レコードが非常にコンパクトになります。Kafka など I/O 律速のシステムの場合、これはプロデューサーおよびコンシューマーのトータルスループットが向上することを意味します。

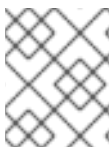
- Kafka プロデューサーおよびコンシューマー用 Avro **Serdes** (シリアライザー/デシリアライザー)。変更イベントレコードを使用するために作成する Kafka コンシューマーアプリケーションは、Avro Serdes を使用して変更イベントレコードをデシリアライズすることができます。

Debezium で Service Registry を使用するには、Debezium コネクタを実行するのに使用している Kafka Connect コンテナイメージに Service Registry コンバーターおよびその依存関係を追加します。



注記

Service Registry プロジェクトは、JSON コンバーターも提供します。このコンバーターは、メッセージが冗長ではないというメリットを持つのに加えて、人間が判読できる JSON を扱うことができます。メッセージ自体にはスキーマ情報は含まれず、スキーマ ID だけが含まれます。



注記

サービスレジストリーが提供するコンバータを使用するには、**apicurio.registry.url** を指定する必要があります。

11.2.2. Avro シリアライゼーションを使用する Debezium コネクタのデプロイの概要

Avro シリアライゼーションを使用する Debezium コネクタをデプロイするには、以下の 3 つの主要タスクを完了する必要があります。

1. [OpenShift でのサービスレジストリーのインストールとデプロイ](#) の手順に従って、サービスレジストリーインスタンスをデプロイします。
2. Debezium [Service Registry Kafka Connect](#) の zip ファイルをダウンロードして Debezium コネクタのディレクトリーに展開し、Avro コンバーターをインストールする。
3. 以下のように設定プロパティを設定して、Avro シリアライゼーションを使用するように Debezium コネクタインスタンスを設定する。

```
key.converter=io.apicurio.registry.utils.converter.AvroConverter
key.converter.apicurio.registry.url=http://apicurio:8080/apis/registry/v2
key.converter.apicurio.registry.auto-register=true
key.converter.apicurio.registry.find-latest=true
value.converter=io.apicurio.registry.utils.converter.AvroConverter
value.converter.apicurio.registry.url=http://apicurio:8080/apis/registry/v2
value.converter.apicurio.registry.auto-register=true
value.converter.apicurio.registry.find-latest=true
schema.name.adjustment.mode=avro
```

内部的には、Kafka Connect は常に JSON キー/値コンバーターを使用して設定およびオフセットを保管します。

11.2.3. Debezium コンテナで Avro を使用するコネクタのデプロイ

ご使用の環境で、提供された Debezium コンテナを使用して、Avro シリアライゼーションを使用す

る Debezium コネクタをデプロイしなければならない場合があります。Debezium 用のカスタム Kafka Connect コンテナイメージをビルドし、Avro コンバーターを使用するように Debezium コネクタを設定するには、以下の手順を完了します。

前提条件

- コンテナを作成および管理するのに十分な権限と共に Docker をインストールしている。
- Avro シリアライゼーションと共にデプロイする Debezium コネクタプラグインをダウンロードしている。

手順

1. Service Registry のインスタンスをデプロイします。[Installing and deploying Service Registry on Open Shift](#)では、以下の手順を説明しています。

- Service Registry のインストール
- AMQ Streams のインストール
- AMQ Streams ストレージのセットアップ

2. Debezium コネクタのアーカイブを展開して、コネクタプラグインのディレクトリ構造を作成します。複数の Debezium コネクタのアーカイブをダウンロードして展開した場合、作成されるディレクトリ構造は以下の例のようになります。

```
tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   └── ...
├── debezium-connector-mysql
│   └── ...
├── debezium-connector-postgres
│   └── ...
└── debezium-connector-sqlserver
    └── ...
```

3. Avro シリアライゼーションを使用するように設定する Debezium コネクタが含まれるディレクトリに Avro コンバーターを追加します。
 - a. [Red Hat Integration のダウンロードサイト](#) に移動し、Service Registry Kafka Connect の zip ファイルをダウンロードします。
 - b. 目的の Debezium コネクタディレクトリにアーカイブを展開します。

複数のタイプの Debezium コネクタを Avro シリアライゼーションを使用するように設定するには、該当するそれぞれのコネクタタイプのディレクトリにアーカイブを展開します。それぞれのディレクトリにアーカイブを抽出するとファイルが重複しますが、これにより依存関係の競合が生じる可能性がなくなります。

4. Avro コンバーターを使用するように設定する Debezium コネクタを実行するためのカスタムイメージを作成して公開します。
 - a. [registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12](#) をベースイメージとして使用して、新しい **Dockerfile** を作成します。以下の例の **my-plugins** を、実際のプラグインディレクトリの名前に置き換えてください。

```
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.0-12
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Kafka Connect は、コネクタの実行を開始する前に、`/opt/kafka/plugins` ディレクトリーにあるサードパーティープラグインをロードします。

- b. docker コンテナイメージをビルドします。例えば、前のステップで作成した docker ファイルを **debezium-container-with-avro** として保存した場合、以下のコマンドを実行します。

```
docker build -t debezium-container-with-avro:latest
```

- c. カスタムイメージをコンテナレジストリーにプッシュします。例を以下に示します。

```
docker push <myregistry.io>/debezium-container-with-avro:latest
```

- d. 新しいコンテナイメージを示します。次のいずれかを行います。

- **KafkaConnect** カスタムリソースの **KafkaConnect.spec.image** プロパティを編集します。このプロパティが設定されていると、クラスターオペレータの **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  image: debezium-container-with-avro
```

- **install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml** ファイルの **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数を編集し、新しいコンテナイメージを示すようにした後、Cluster Operator を再インストールします。このファイルを編集する場合は、これを OpenShift クラスターに適用する必要があります。

5. Avro コンバーターを使用するように設定されたそれぞれの Debezium コネクタをデプロイします。それぞれの Debezium コネクタについて、以下の設定を行います。

- a. Debezium コネクタインスタンスを作成します。次の **inventory-connector.yaml** ファイルの例では、Avro コンバーターを使用するように設定された MySQL コネクタインスタンスを定義する **Kafka Connector** カスタムリソースを作成しています。

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

```

database.password: dbz
database.server.id: 184054
topic.prefix: dbserver1
database.include.list: inventory
schema.history.internal.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092
schema.history.internal.kafka.topic: schema-changes.inventory
schema.name.adjustment.mode: avro
key.converter: io.apicurio.registry.utils.converter.AvroConverter
key.converter.apicurio.registry.url: http://apicurio:8080/api
key.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
value.converter: io.apicurio.registry.utils.converter.AvroConverter
value.converter.apicurio.registry.url: http://apicurio:8080/api
value.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy

```

- b. コネクタインスタンスを適用します。以下に例を示します。

oc apply -f inventory-connector.yaml

これにより **inventory-connector** が登録され、コネクタが **inventory** データベースに対して実行されるようになります。

6. コネクタが作成され、指定されたデータベース内の変更の追跡を開始したことを確認します。例えば **inventory-connector** が起動したときの Kafka Connect のログ出力を見ることで、コネクタのインスタンスを確認することができます。

- a. Kafka Connect のログ出力を表示します。

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

- b. ログの出力を確認し、初回のスナップショットが実行されたことを確認します。以下のような行が表示されるはずです。

```

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

```

スナップショットは、複数のステップを経て作成されます。

```

...
2020-02-21 17:57:30,822 INFO Step 0: disabling autocommit, enabling repeatable read
transactions, and setting lock wait timeout to 10
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,836 INFO Step 1: flush and obtain global read lock to prevent
writes to database (io.debezium.connector.mysql.SnapshotReader) [debezium-
mysqlconnector-dbserver1-snapshot]

```

```

2020-02-21 17:57:30,839 INFO Step 2: start transaction with consistent snapshot
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,840 INFO Step 3: read binlog position of MySQL primary server
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,843 INFO using binlog 'mysql-bin.000003' at position '154' and gtid
" (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...
2020-02-21 17:57:34,423 INFO Step 9: committing transaction
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:34,424 INFO Completed snapshot in 00:00:03.632
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...

```

スナップショットの作成が完了した後、Debezium は (例として) **inventory** データベースの **binlog** に生じる変更の追跡を開始し、変更イベントの有無を監視します。

```

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

```

11.2.4. Avro の名前の要件について

Avro の [ドキュメント](#) に記載されているように、名前は以下のルールに従う必要があります。

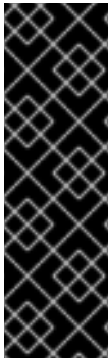
- **[A-Za-z]** で始まる
- その後に **[A-Za-z0-9]** の文字のみが含まれる

Debezium は、対応する Avro フィールドのベースとして列の名前を使用します。これにより、列の名前も Avro の命名規則に従わないと、シリアル化中に問題が発生する可能性があります。列の名前が Avro の命名規則に従わない場合は、各 Debezium コネクタの設定プロパティ **sanitize.field.names** を **true** に設定することができます。 **sanitize.field.names** を **true** に設定すると、スキーマを実際に変更することなく、適さないフィールドをシリアル化することができます。

11.3. CLOUDEVENTS フォーマットでの DEBEZIUM 変更イベントレコードの出力

[CloudEvents](#) は、共通の方法でイベントデータを記述するための仕様です。その目的は、サービス、プ

ラットフォーム、およびシステム間の相互運用性を提供することです。Debezium では、MongoDB、MySQL、PostgreSQL、または SQL Server コネクタを設定して、CloudEvents 仕様に準拠した変更イベントレコードを出力することができます。



重要

CloudEvents フォーマットでの変更イベントレコードの出力は、テクノロジープレビュー機能です。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の機能は、最新の技術をいち早く提供して、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。

CloudEvents 仕様は、以下の項目を定義します。

- 標準化されたイベント属性のセット
- カスタム属性を定義するためのルール
- イベントフォーマットを JSON や Avro 等のシリアライズした表現にマッピングするためのエンコード規則
- Apache Kafka、HTTP、または AMQP 等のトランスポート層のプロトコルバインディング

CloudEvents 仕様に準拠する変更イベントレコードを出力するように Debezium コネクタを設定するために、Debezium では Kafka Connect メッセージコンバーターである **io.debezium.converters.CloudEventsConverter** を利用することができます。

現時点では、構造化マッピングモードだけがサポートされています。Cloud Events の変更イベントのエンベロープは、JSON または Avro であり、各エンベロープタイプは **data** フォーマットとして JSON または Avro をサポートしています。今後の Debezium リリースでは、バイナリーマッピングモードがサポートされる計画です。

CloudEvents フォーマットでの変更イベントの出力に関する情報は、以下のように整理されます。

- [「CloudEvents フォーマットでの Debezium 変更イベントレコードの例」](#)
- [「Debezium CloudEvents コンバーターの設定例」](#)
- [「Debezium CloudEvents コンバーター設定オプション」](#)

Avro 使用の詳細については、以下を参照してください。

- [Avro シリアライゼーション](#)
- [Apicurio Registry](#)

11.3.1. CloudEvents フォーマットでの Debezium 変更イベントレコードの例

以下の例は、PostgreSQL コネクタから出力される CloudEvents 変更イベントレコードを示しています。この例では、PostgreSQL コネクタは CloudEvents フォーマットエンベロープおよび **data** フォーマットとして JSON を使用するように設定されています。

```
{
```

```

{id": "name:test_server;lsn:29274832;txId:565", 1
"source": "/debezium/postgresql/test_server", 2
"specversion": "1.0", 3
"type": "io.debezium.postgresql.datachangeevent", 4
"time": "2020-01-13T13:55:39.738Z", 5
"datacontenttype": "application/json", 6
"iodebeziumop": "r", 7
"iodebeziumversion": "2.1.4.Final", 8
"iodebeziumconnector": "postgresql",
"iodebeziumname": "test_server",
"iodebeziumtsms": "1578923739738",
"iodebeziumsnapshot": "true",
"iodebeziumdb": "postgres",
"iodebeziumschema": "s1",
"iodebeziumtable": "a",
"iodebeziumtxId": "565",
"iodebeziumlsn": "29274832",
"iodebeziumxmin": null,
"iodebeziumtxid": "565", 9
"iodebeziumtxtotalorder": "1",
"iodebeziumtxdatacollectionorder": "1",
"data": { 10
  "before": null,
  "after": {
    "pk": 1,
    "name": "Bob"
  }
}
}

```

- 1 1 1 変更イベントの内容に基づいてコネクタが変更イベントに生成する一意の ID。
- 2 2 2 イベントのソースで、コネクタ設定の **topic.prefix** プロパティで指定されたデータベースの論理名です。
- 3 3 3 CloudEvents 仕様のバージョン。
- 4 4 4 変更イベントを生成したコネクタタイプ。このフィールドの形式は **io.debezium.CONNECTOR_TYPE.datachangeevent** です。CONNECTOR_TYPE の値は **mongodb**、**mysql**、**postgresql**、または **sqlserver** です。
- 5 5 ソースデータベースの変更時刻。
- 6 data 属性のコンテンツタイプ (この例では JSON) を記述します。それ以外には Avro のみ有効です。
- 7 操作の ID。許容値は、**r** (読み取り)、**c** (作成)、**u** (更新)、または **d** (削除) です。
- 8 Debezium 変更イベントから認識されるすべての **source** 属性は、属性名の前に **iodebezium** を追加して CloudEvents エクステンション属性にマッピングされます。
- 9 コネクタで有効にすると、Debezium 変更イベントから認識されるそれぞれの **transaction** 属性は、属性名の前に **iodebeziumtx** を追加して CloudEvents エクステンション属性にマッピングされます。

- 10 実際のデータ変更。操作およびコネクタによって、データに **before**、**after** または **patch** フィールドが含まれる場合があります。

以下の例も、PostgreSQL コネクタから出力される CloudEvents 変更イベントレコードを示しています。この例でも、PostgreSQL コネクタは CloudEvents フォーマットエンベロープとして JSON を使用するように設定されていますが、ここではコネクタは **data** フォーマットに Avro を使用するように設定されています。

```
{
  "id" : "name:test_server;lsn:33227720;txld:578",
  "source" : "/debezium/postgresql/test_server",
  "specversion" : "1.0",
  "type" : "io.debezium.postgresql.datachangeevent",
  "time" : "2020-01-13T14:04:18.597Z",
  "datacontenttype" : "application/avro",
  "dataschema" : "http://my-registry/schemas/ids/1",
  "iodebeziumop" : "r",
  "iodebeziumversion" : "2.1.4.Final",
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578924258597",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxld" : "578",
  "iodebeziumlsn" : "33227720",
  "iodebeziumxmin" : null,
  "iodebeziumtxid" : "578",
  "iodebeziumtxtotalorder" : "1",
  "iodebeziumtxdatacollectionorder" : "1",
  "data" : "AAAAAAEAAgICAg=="
}
```

- 1 **data** 属性に Avro バイナリーデータが含まれていることを示します。
- 2 Avro データが準拠するスキーマの URI。
- 3 **data** 属性には、base64 でエンコードされた Avro バイナリーデータが含まれます。

data 属性に加えてエンベロープに Avro を使用することもできます。

11.3.2. Debezium CloudEvents コンバーターの設定例

Debezium コネクタ設定で **io.debezium.converters.CloudEventsConverter** を設定します。次の特性を持つ変更イベントレコードを出力するように CloudEvents コンバーターを設定する方法を以下の例に示します。

- エンベロープとして JSON を使用する。
- **http://my-registry/schemas/ids/1** のスキーマレジストリーを使用して、**データ** 属性をバイナリー Avro データとしてシリアライズする。


```

...
"value.converter": "io.debezium.converters.CloudEventsConverter",
"value.converter.serializer.type" : "json",
"value.converter.data.serializer.type" : "avro",
"value.converter.avro.schema.registry.url": "http://my-registry/schemas/ids/1"
...

```

① **json** はデフォルトであるため、**serializer.type** の指定は任意です。

CloudEvents コンバーターは、Kafka レコードの値を変換します。レコードのキーを操作する場合は、同じコネクタ設定で **key.converter** を指定することができます。たとえば、**StringConverter**、**LongConverter**、**JsonConverter**、または **AvroConverter** を指定できます。

11.3.3. Debezium CloudEvents コンバーター設定オプション

CloudEvent コンバーターを使用するように Debezium コネクタを設定する場合、以下のオプションを指定できます。

表11.3 CloudEvents コンバーター設定オプションの説明

オプション	デフォルト	説明
serializer.type	json	CloudEvents エンベロープ構造に使用するエンコーディングタイプ。値は json または avro に指定できます。
data.serializer.type	json	data 属性に使用するエンコーディングタイプ。値は json または avro に指定できます。
json. ...	該当なし	JSON を使用する際に、ベースとなるコンバーターに渡される任意の設定オプション。 json. 接頭辞が削除されます。
avro. ...	該当なし	Avro を使用する際に、ベースとなるコンバーターに渡される任意の設定オプション。 avro. 接頭辞が削除されます。たとえば、Avro データの場合は、 avro.schema.registry.url オプションを指定します。
schema.name.adjustment.mode	none	コネクタで使用されるメッセージコンバータとの互換性のために、スキーマ名をどのように調整するかを指定します。値は none または avro にすることができます。

11.4. DEBEZIUM コネクタへのシグナル送信

Debezium のシグナルメカニズムを使用すると、コネクタの動作を変更したり、テーブルの **アドホックスナップショット** を起動するなどの1回限りのアクションをトリガーする方法が可能になります。コネクタをトリガーして指定のアクションを実行するには、SQL コマンドを実行して指定のシグナルテーブル (別称: シグナルデータコレクション) にシグナルメッセージを追加します。ソースデータベース

ス上に作成したシグナリングテーブルは、Debezium との通信専用指定されています。Debezium は、新しい [ロギングレコード](#) や [アドホックスナップショットレコード](#) がシグナリングテーブルに追加されたことを検出すると、シグナルを読み込んで、要求された操作を開始します。

シグナリングは、以下の Debezium コネクタで利用可能です。

- Db2
- MongoDB (テクノロジープレビュー)
- MySQL
- Oracle
- PostgreSQL
- SQL Server

11.4.1. Debezium シグナリングの有効化

デフォルトでは、Debezium シグナリングメカニズムは無効になっています。シグナリングを使用したいコネクタごとに、明示的にシグナリングを有効にする必要があります。

手順

1. ソースデータベースで、コネクタへのシグナル送信用にデータコレクションテーブルを作成します。シグナリングデータコレクションの必要な構造については、[シグナリングデータコレクションの構造](#)を参照してください。
2. Db2 や SQL Server など、ネイティブな変更データキャプチャ (CDC) メカニズムを実装しているソースデータベースでは、信号テーブルの CDC を有効にします。
3. Debezium コネクタの設定にシグナリングデータコレクションの名前を追加します。コネクタ設定で、プロパティ **signal.data.collection** を追加し、その値を手順 1 で作成したシグナルデータコレクションの完全修飾名に設定します。

例: **signal.data.collection = inventory.debezium_signals**.

シグナリングコレクションの完全修飾名のフォーマットは、コネクタによって異なります。次の例では、各コネクタに使用する名前フォーマットを示しています。

Db2

<schemaName>.<tableName>

MongoDB

<databaseName>.<collectionName>

MySQL

<databaseName>.<tableName>

Oracle

<databaseName>.<schemaName>.<tableName>

PostgreSQL

<schemaName>.<tableName>

SQL Server

<databaseName>.<schemaName>.<tableName>

signal.data.collection プロパティの設定の詳細は、お使いのコネクタの設定プロパティの表を参照してください。

11.4.1.1. デベジウムシグナリングデータ収集の必須構造

シグナルデータコレクションまたはシグナルテーブルは、コネクタに送信して指定の操作をトリガーするシグナルを保存します。シグナリングテーブルの構造は、以下の標準フォーマットに準拠する必要があります。

- 3つのフィールド(列)があります。
- フィールドは、表1のように決まった順序で配置されています。

表11.4 シグナリングデータ収集の必須構造

フィールド	タイプ	説明
id (必須)	string	シグナルのインスタンスを識別する任意のユニークな文字列です。シグナリングテーブルに登録するシグナルには、それぞれ ID を割り当てます。通常、ID は UUID 文字列です。シグナルインスタンスは、ロギング、デバッグ、デデュープなどに使用できます。シグナルによって Debezium が増分スナップショットを実行すると、任意の id 文字列を持つシグナルメッセージが生成されます。生成されたメッセージに含まれる id 文字列が、送信されたシグナルの id 文字列と一致しません。
type (必須)	string	送信する信号の種類を指定します。信号の種類によっては、信号が利用可能なすべてのコネクタで使用できますが、他の信号の種類は特定のコネクタでのみ使用できます。
data (オプション)	string	シグナルアクションに渡す、JSON 形式のパラメーターを指定します。それぞれの信号タイプには、特定のデータセットが必要です。



注記

データコレクションのフィールド名は任意です。前述の表には、推奨される名称が記載されています。異なる命名規則を使用している場合は、各フィールドの値が期待される内容と一致していることを確認してください。

11.4.1.2. Debezium シグナルのデータコレクションの作成

標準 SQL の DDL クエリーをソースデータベースに送信して、シグナリングテーブルを作成します。

前提条件

- ターゲットデータベースにテーブルを作成するのに十分なアクセス権限があります。

手順

- SQL クエリーをソースデータベースに送信して、[必須の構造](#) と合致するテーブルを作成します。例:

```
CREATE TABLE <tableName> (id VARCHAR(<varcharValue>) PRIMARY KEY, type
VARCHAR(<varcharValue>) NOT NULL, data VARCHAR(<varcharValue>) NULL);
```



注記

id 変数の **VARCHAR** パラメーターに割り当てる容量は、シグナリングテーブルに送信されるシグナルの ID 文字列のサイズを考慮して、十分に確保する必要があります。ID のサイズが使用可能なスペースを超える場合、コネクタは信号を処理できません。

次の例は、3 列の **debezium_signal** テーブルを作成する **CREATE TABLE** コマンドです。

```
CREATE TABLE debezium_signal (id VARCHAR(42) PRIMARY KEY, type VARCHAR(32) NOT
NULL, data VARCHAR(2048) NULL);
```

11.4.2. デベージウムシグナルアクションの種類

シグナリングを使用して、以下のアクションを起こすことができます。

- ログにメッセージを追加する。
- アドホックなスナップショットのトリガー
- アドホックスナップショットの実行を停止
- 増分スナップショットを一時停止
- **incremental** スナップショットを再開

一部の信号はすべてのコネクタに対応していません。

11.4.2.1. ロギング信号

log シグナルタイプのシグナリングテーブルエントリを作成することで、ログにエントリを追加するようコネクタに要求できます。シグナルの処理後、コネクタは指定されたメッセージをログに出力します。オプションとして、結果として得られるメッセージにストリーミング座標が含まれるようにシグナルを設定することもできます。

表11.5 ログメッセージを追加するためのシグナリングレコードの例

列	値	説明
id	924e3ff8-2245-43ca-ba77-2af9af02fa07	
type	log	シグナルのアクションタイプです。

列	値	説明
data	<code>{"message": "Signal message at offset {}}"</code>	message パラメータは、ログに出力する文字列を指定します。 メッセージにプレースホルダー (<code>{}</code>) を追加した場合は、ストリーミング座標に置き換えられます。

11.4.2.2. アドホックなスナップショット信号

シグナルタイプ **execute-snapshot** のシグナリングテーブルエントリを作成することで、アドホックスナップショットの開始をコネクタに要求することができます。信号を処理した後、コネクタは要求されたスナップショットオペレーションを実行します。

コネクタが最初に起動したときに実行される最初のスナップショットとは異なり、アドホックスナップショットは、コネクタがすでにデータベースからの変更イベントのストリーミングを開始した後のランタイム中に発生します。いつでもアドホックなスナップショットを開始することができます。

アドホックスナップショットは、以下の Debezium コネクタで利用可能です。

- Db2
- MySQL
- Oracle
- PostgreSQL
- SQL Server

表11.6 アドホックスナップショットシグナルレコードの例

列	値
id	d139b9b7-7777-4547-917d-e1775ea61d41
type	execute-snapshot
data	<code>{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}</code>

現在、**execute-snapshot** アクションは [増分スナップショット](#) のみをトリガーします。

アドホックスナップショットの詳細については、お使いのコネクタのドキュメントのスナップショットのトピックを参照してください。

関連情報

- [Db2 コネクタのアドホックスナップショット](#)
- [My SQL コネクタのアドホックスナップショット](#)
- [Oracle コネクタのアドホックスナップショット](#)
- [Postgre SQL コネクタのアドホックスナップショット](#)

- [SQL Server コネクターのアドホックスナップショット](#)

アドホックスナップショット停止シグナル

stop-snapshot シグナルタイプでシグナルテーブルエントリを作成することにより、進行中のアドホックスナップショットを停止するようコネクターに要求できます。シグナルを処理した後、コネクターは現在進行中のスナップショット操作を停止します。

次の Debezium コネクターのアドホックスナップショットを停止できます。

- Db2
- MySQL
- Oracle
- PostgreSQL
- SQL Server

表11.7 停止アドホックスナップショットシグナルレコードの例

列	値
id	d139b9b7-7777-4547-917d-e1775ea61d41
type	stop-snapshot
data	{"type":"INCREMENTAL", "data-collections": ["public.MyFirstTable"]}

シグナルの **type** を指定する必要があります。**data-collections** フィールドはオプションです。現在のスナップショットのすべてのアクティビティを停止するようコネクターに要求するには、**data-collections** フィールドを空白のままにします。増分スナップショットを続行したいが、スナップショットから特定のコレクションを除外したい場合は、除外するコレクションまたは正規表現の名前のコンマ区切りリストを指定します。コネクターが信号を処理した後、増分スナップショットが続行されますが、指定したコレクションからデータが除外されます。

11.4.2.3. 増分スナップショット

増分スナップショットはアドホックスナップショットの一種です。増分スナップショットでは、初期スナップショットと同様に、指定したテーブルのベースライン状態をキャプチャします。しかし、初期スナップショットとは異なり、増分スナップショットでは、一度にすべてのテーブルをキャプチャするのではなく、チャンク単位でテーブルをキャプチャします。このコネクターは、スナップショットの進捗状況を追跡するために、電子透かしを使用しています。

増分スナップショットは、指定されたテーブルの初期状態を単一のモノリシックな操作ではなくチャンクでキャプチャすることにより、初期スナップショットのプロセスに比べて以下のような利点があります。

- コネクターが指定されたテーブルのベースライン状態をキャプチャしている間、トランザクションログからのほぼリアルタイムのイベントのストリーミングは中断することなく継続されます。
- 増分スナップショットの処理が中断されても、中断した時点から再開することができます。

- 増分スナップショットはいつでも開始できます。

増分スナップショットの一時停止シグナル

pause-snapshot シグナルタイプを使用してシグナルテーブルエントリを作成することにより、進行中の増分スナップショットを一時停止するようコネクタに要求できます。信号を処理した後、コネクタは現在進行中のスナップショット操作の一時停止を停止します。そのため、信号の処理中の位置でスナップショット処理が一時停止されるため、データ収集を指定することはできません。

次の Debezium コネクタの増分スナップショットを一時停止できます。

- Db2
- MySQL
- Oracle
- PostgreSQL
- SQL Server

表11.8 一時停止増分スナップショットシグナルレコードの例

列	値
id	d139b9b7-7777-4547-917d-e1775ea61d41
type	pause-snapshot

シグナルの **type** を指定する必要があります。 **data** フィールドは無視されます。

増分スナップショットの一時停止シグナル

一時停止した増分スナップショットを再開するようコネクタに要求するには、 **resume-snapshot** シグナルタイプでシグナルテーブルエントリを作成します。信号を処理した後、コネクタは以前に一時停止したスナップショット操作を再開します。

次の Debezium コネクタの増分スナップショットを再開できます。

- Db2
- MySQL
- Oracle
- PostgreSQL
- SQL Server

表11.9 再開増分スナップショットシグナルレコードの例

列	値
id	d139b9b7-7777-4547-917d-e1775ea61d41

列	値
type	resume-snapshot

シグナルの **type** を指定する必要があります。 **data** フィールドは無視されます。

増分スナップショットの詳細については、お使いのコネクターのドキュメントのスナップショットのトピックを参照してください。

関連情報

- [Db2 コネクターのインクリメンタルスナップショット](#)
- [My SQL コネクターの増分スナップショット](#)
- [Oracle コネクターの増分スナップショット](#)
- [Postgre SQL コネクターの増分スナップショット](#)
- [SQL Server コネクターのインクリメンタルスナップショット](#)

第12章 APACHE KAFKA で交換されたメッセージを修正するためのトランスフォームの適用

Debezium には、変更イベントレコードを修正するために使用できるいくつかのシングルメッセージ変換 (SMT) があります。Apache Kafka にレコードを送信する前に、レコードを修正する変換を適用するようにコネクタを設定することができます。また、Debezium SMT をシンクコネクタに適用して、コネクタが Kafka トピックから読み込む前にレコードを修正することもできます。

特定のメッセージのみに選択的に変換を適用したい場合は、SMT を適用する条件を定義する Kafka Connect プレディケートを設定することができます。

Debezium は以下の SMT を提供しています。

トピックルーター SMT

元のトピック名に適用される正規表現に基づいて、変更イベントレコードを特定のトピックに再ルーティングします。

コンテンツベースルーター SMT

イベントの内容に基づいて、指定された変更イベントのレコードを再送信します。

メッセージフィルタリング SMT

イベントレコードのサブセットを宛先の Kafka トピックに伝搬することができます。この変換では、イベントレコードの内容に基づいて、コネクタが発信する変更イベントレコードに正規表現を適用します。式にマッチしたレコードのみが対象のトピックに書き込まれます。その他の記録は無視されます。

新記録の状態抽出 SMT

Debezium の変更イベントレコードの複雑な構造をシンプルなフォーマットにフラット化します。構造を簡略化することで、元の構造を消費できないシンクコネクタでの処理が可能になります。

MongoDB 新記録の状態抽出

Debezium MongoDB コネクタ変更イベントレコードの複雑な構造を簡素化します。構造を簡略化して、元のイベント構造を消費できないシンクコネクタでの処理が可能になります。

送信トレイ (Outbox) イベントルーター SMT

複数のサービス間での安全で信頼性の高いデータ交換を可能にするアウトボックスパターンのサポートを提供します。

MongoDB outbox event router SMT

複数のサービス間で安全で信頼性の高いデータ交換を可能にするために、送信トレイパターンを使用するためのサポートを提供します。

12.1. SMT 述語を使用した変換の選択的適用

コネクタに単一メッセージ変換 (SMT) を設定する場合、変換の述語を定義できます。述語は、コネクタが処理するメッセージのサブセットに変換を条件的に適用する方法を指定します。Debezium などのソースコネクタまたはシンクコネクタに対して設定する変換に、述語を割り当てることができます。

12.1.1. SMT 述語について

Debezium は、Kafka Connect がレコードを Kafka トピックに保存する前に、イベントレコードを変更するために使用できるさまざまな単一メッセージ変換 (SMT) を提供します。デフォルトでは、Debezium コネクタにこれらの SMT のいずれかを設定する場合、Kafka Connect はコネクタが出力するすべてのレコードに変換を適用します。ただし、共通の特徴を共有する変更イベントメッセージのサブセットのみが変更されるように、一部の变換を適用する場合があります。

たとえば、Debezium コネクタでは、特定のテーブルからのイベントメッセージまたは特定のヘッダーキーが含まれるイベントメッセージでのみ変換を実行する必要がある場合があります。Apache Kafka 2.6 以降を実行する環境では、変換に述語ステートメントを追加して、特定のレコードだけに SMT を適用するように Kafka Connect に指示できます。述語では、Kafka Connect が処理する各メッセージを評価するために使用する条件を指定します。Debezium コネクタが変更イベントメッセージを出力すると、Kafka Connect はメッセージを設定済みの述語条件に対して確認します。イベントメッセージで条件が満たされる場合、Kafka Connect は変換を適用し、メッセージを Kafka トピックに書き込みます。条件に一致しないメッセージは、そのまま Kafka に送信されます。

この状況は、シンクコネクタ SMT に定義する述語に類似しています。コネクタは Kafka トピックからメッセージを読み取り、Kafka Connect はメッセージを述語条件に対して評価します。メッセージが条件と一致する場合、Kafka Connect は変換を適用し、メッセージをシンクコネクタに渡します。

述語を定義したら、それを再利用し、複数の変換に適用できます。述語には **negate** オプションがあり、これを使うと述語の条件を反転させて、述語文で定義された条件に一致しないレコードにのみ適用することができます。**negate** オプションを使うと、条件を否定することを前提とした他の変換と述語をペアにすることができます。

述語要素

述語には、以下の要素が含まれます。

- **predicates** 接頭辞
- エイリアス (例:**isOutbox Table**)
- タイプ (例えば、**org.apache.kafka.connect.transforms.predicates.Topic Name Matches**)Kafka Connect は、デフォルトの述語型のセットを提供します。これは、独自のカスタム述語を定義することで補足できます。
- 条件ステートメントと追加の設定プロパティ (述語の型 (正規表現の命名パターンなど) による)

デフォルトの述語型

デフォルトでは、以下の述語型を利用できます。

HasHeaderKey

Kafka Connect が評価するイベントメッセージのヘッダーのキー名を指定します。述語は、指定された名前を持つヘッダーキーが含まれるレコードを **true** と評価します。

RecordIsTombstone

Kafka **廃棄** レコードとマッチします。述語は、**null** 値を持つすべてのレコードに対して **true** と評価されます。この述語をフィルター SMT と組み合わせて使用して廃棄レコードを削除します。この述語には設定パラメータはありません。

Kafka の tombstone は、0 バイトの **null** ペイロードを持つキーを持つレコードです。Debezium コネクタがソースデータベースで削除操作を処理すると、コネクタは削除操作に対して 2 つの変更イベントを出力します。

- データベースレコードの以前の値を提供する削除操作 ("**op**" : "**d**") イベント。
- キーは同じだが、値が **null** の墓石イベント。
tombstone は、行の削除マーカを表します。[ログコンパクション](#) が Kafka に対して有効になっている場合、コンパクト時に Kafka は tombstone と同じキーを共有するすべてのイベントを削除します。ログコンパクションは、トピックの [delete.retention.ms](#) 設定で制御されるコンパクト化の間隔で定期的に行われます。

廃棄 (tombstone) イベントを出力しないように Debezium を設定することは可能ですが、ログコンパクション中に想定される動作を維持するために Debezium が tombstone を出力するのを許可することを推奨します。tombstone を抑制することにより、ログコンパクション中に削除されたキーのレコードを Kafka が削除しなくなります。環境に tombstone を処理できないシンクコネクタが含まれる場合は、**RecordIsTombstone** 述語で SMT を使用して廃棄レコードをフィルタリングするようにシンクコネクタを設定できます。

TopicNameMatches

Kafka Connect が照合するトピックの名前を指定する正規表現。トピック名が指定の正規表現と一致するコネクタレコードの場合、述語は true になります。この述語を使用して、ソーステーブルの名前に基づいてレコードに SMT を適用します。

関連情報

- [KIP-585: Filter and Conditional SMTs](#)
- [Kafka Connect 述語の Apache Kafka ドキュメント](#)

12.1.2. SMT 述語の定義

デフォルトでは、Kafka Connect は Debezium コネクタ設定の各単一メッセージ変換を、Debezium から受け取るすべての変更イベントレコードに適用します。Apache Kafka 2.6 以降では、Kafka Connect による変換の適用方法を制御するコネクタ設定で変換に SMT 述語を定義できます。述語ステートメントは、Kafka Connect が Debezium によって出力されるイベントレコードに変換を適用する条件を定義します。Kafka Connect は述語ステートメントを評価し、SMT を選択的に、述語で定義される条件に一致するレコードのサブセットに適用します。Kafka Connect 述語の設定は、変換の設定と似ています。述語エイリアスを指定し、エイリアスを変換に関連付け、述語の型および設定を定義します。

前提条件

- Debezium 環境が Apache Kafka 2.6 以降を実行している。
- SMT が Debezium コネクタ用に設定されている。

手順

1. Debezium コネクタの設定で、**predicates** パラメーターに、**IsOutbox Table** などの predicate エイリアスを指定します。
2. コネクタ設定の変換エイリアスに述語エイリアスを追加して、条件付きに適用する変換に述語エイリアスを関連付けます。

```
transforms.<TRANSFORM_ALIAS>.predicate=<PREDICATE_ALIAS>
```

以下に例を示します。

```
transforms.outbox.predicate=IsOutboxTable
```

3. 型を指定し、設定パラメーターの値を指定して述語を設定します。
 - a. 型には、Kafka Connect で使用できる以下のデフォルト型のいずれかを指定します。
 - HasHeaderKey

- RecordsIsTombstone
- TopicNameMatches
以下に例を示します。

```
predicates.IsOutboxTable.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
```

- TopicNameMatch または **HasHeaderKey** 述語に対して、照合するトピックまたはヘッダー名の正規表現を指定します。
以下に例を示します。

```
predicates.IsOutboxTable.pattern=outbox.event.*
```

- 条件を反転する場合は、**negate** キーワードを変換エイリアスに追加し、**true** に設定します。
以下に例を示します。

```
transforms.outbox.negate=true
```

前述のプロパティは、述語がマッチするレコードセットを反転し、Kafka Connect は述語で指定された条件に一致しないレコードに変換を適用するようにします。

例: 送信トレイイベントルーターの変換用の TopicNameMatch 述語

以下の例に示す Debezium コネクター設定は、送信トレイイベントルーター変換を Debezium が Kafka **outbox.event.order** トピックに出力するメッセージにだけ適用します。

TopicNameMatch 述語は送信トレイテーブルからのメッセージだけを **true** と評価するため (**outbox.event.***)、データベースの他のテーブルから送信されるメッセージに変換は適用されません。

```
transforms=outbox
transforms.outbox.predicate=IsOutboxTable
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
predicates=IsOutboxTable
predicates.IsOutboxTable.type=org.apache.kafka.connect.transforms.predicates.TopicNameMatches
predicates.IsOutboxTable.pattern=outbox.event.*
```

12.1.3. 廃棄 (tombstone) イベントの無視

Debezium が廃棄 (tombstone) イベントを生成するかどうかや、Kafka がそれらを保持する期間を制御できます。データパイプラインによっては、Debezium が廃棄 (tombstone) イベントを出力しないように、コネクターの **tombstones.on.delete** プロパティを設定する必要がある場合があります。

Debezium が tombstone を出力できるようにするかどうかは、トピックがどのように環境で使用されるかと、シンクコンシューマーの特性によって異なります。一部のシンクコネクターは、廃棄 (tombstone) イベントに依存してダウンストリームデータストアからレコードを削除します。シンクコネクターが廃棄 (tombstone) レコードに依存してダウンストリームデータストアのレコードを削除するタイミングを示す場合は、Debezium がそれらを出力するように設定します。

tombstone を生成するように Debezium を設定する場合、シンクコネクターが廃棄 (tombstone) イベントを受信するように追加の設定が必要になります。ログコンパクション中に Kafka がイベントメッセージを削除する前に、コネクターがイベントメッセージを読み取るために、トピックの保持ポリシーを設定する必要があります。コンパクション前にトピックが tombstone を保持する時間の長さは、トピックの **delete.retention.ms** プロパティによって制御されます。

デフォルトでは、コネクターの `tombstones.on.delete` プロパティが `true` に設定されているため、削除イベントが発生するたびに、コネクターは墓石を生成します。このプロパティを `false` に設定して、Debezium が Kafka トピックに墓石の記録を保存しないようにすると、墓石の記録がないために意図しない結果になる可能性があります。Kafka はログコンパクション時に tombstone に依存して、削除されたキーに関連するレコードを削除します。

null 値のレコードを処理できないシンクコネクターやダウンストリームの Kafka コンシューマーをサポートする必要がある場合、Debezium が廃棄を出力するのを防止するのではなく、コンシューマーが読み取る前に `RecordsTombstone` 述語型を使用して廃棄メッセージを削除する述語を使用するコネクターの SMT を設定することを検討してください。

手順

- Debezium が削除されたデータベースレコードの墓石イベントを発行しないようにするには、コネクターオプション `tombstones.on.delete` を `false` に設定します。以下に例を示します。

```
“tombstones.on.delete”: “false”
```

12.2. 指定したトピックへの DEBEZIUM イベントレコードのルーティング

データ変更イベントが含まれるそれぞれの Kafka レコードは、デフォルトのルーティング先トピックを持ちます。必要に応じて、レコードが Kafka Connect コンバーターに到達する前に、指定したトピックにレコードを再ルーティングすることができます。そのために、Debezium ではトピックルーティング単一メッセージ変換 (SMT) を利用することができます。Debezium コネクターの Kafka Connect 設定でこの変換を設定します。設定オプションにより、以下の項目を指定することができます。

- 再ルーティングするレコードを識別するための式。
- ルーティング先トピックに解決する式。
- 宛先トピックに再ルーティングされるレコード間でキーの一意性を確保する方法。

変換の設定により必要な動作が得られるようにするのは、ユーザー側の範疇です。Debezium は、変換の設定により得られる動作を検証しません。

トピックルーティング変換は [Kafka Connect SMT](#) です。

詳細は以下のセクションを参照してください。

- [「指定したトピックに Debezium レコードをルーティングするユースケース」](#)
- [「複数テーブルの Debezium レコードを1つのトピックにルーティングする例」](#)
- [「同一トピックにルーティングされる Debezium レコード間でのキーの一意性確保」](#)
- [「Debezium トピックルーティング変換設定用のオプション」](#)

12.2.1. 指定したトピックに Debezium レコードをルーティングするユースケース

Debezium コネクターのデフォルト動作では、それぞれの変更イベントレコードは、名前がデータベースおよび変更が加えられたテーブルの名前から作られるトピックに送信されます。つまり、トピックは1つの物理テーブルのレコードを受け取ります。トピックが複数の物理テーブルのレコードを受け取るようにするには、Debezium コネクターを設定してレコードをそのトピックに再ルーティングする必要があります。

論理テーブル

論理テーブルは、複数の物理テーブルのレコードを1つのトピックにルーティングする場合の一般的なユースケースです。論理テーブル内には、すべて同じスキーマを持つ複数の物理テーブルがあります。たとえば、シャーディングされたテーブルのスキーマは同一です。論理テーブルは、**db_shard1.my_table** および **db_shard2.my_table** という2つ以上のシャード化されたテーブルで設定されているかもしれません。テーブルは異なるシャードにあり物理的に別個のものですが、1つにまとめ論理テーブルを形成します。任意のシャード内のテーブルの変更イベントレコードを、同じトピックに再ルーティングすることができます。

パーティションで分割された PostgreSQL テーブル

Debezium PostgreSQL コネクタがパーティションで分割されたテーブルの変更をキャプチャーする場合、デフォルトの動作では、変更イベントレコードはパーティションごとに異なるトピックにルーティングされます。すべてのパーティションからのレコードを1つのトピックに出力するには、トピックルーティング SMT を設定します。パーティションで分割されたテーブルの各キーは必ず一意であるため、キーの一意性を確保するために SMT がキーフィールドを追加しないように **key.enforce.uniqueness=false** を設定します。デフォルトの動作では、キーフィールドが追加されません。

12.2.2. 複数テーブルの Debezium レコードを1つのトピックにルーティングする例

複数の物理テーブルの変更イベントレコードを同じトピックにルーティングするには、Debezium コネクタの Kafka Connect 設定でトピックルーティング変換を設定します。トピックルーティング SMT を設定するには、以下の項目を決定する正規表現を指定する必要があります。

- レコードをルーティングするテーブル。これらのテーブルのスキーマは、すべて同一でなければなりません。
- ルーティング先トピックの名前。

以下の例のコネクタ設定では、トピックルーティング SMT の複数のオプションを設定します。

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

topic.regex

変更イベントレコードを特定のトピックにルーティングする必要があるかどうかを決定するために、変換がそれぞれのレコードに適用する正規表現を指定します。

この例では、正規表現 **(.*)customers_shard(.*)** は、名前に **customers_shard** 文字列が含まれるテーブルに対する変更のレコードがマッチします。この場合、以下の名前のテーブルのレコードが再ルーティングされます。

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

topic.replacement

ルーティング先トピックの名前を表す正規表現を指定します。変換により、マッチする各レコードがこの式で識別されるトピックにルーティングされます。この例では、上記3つのシャーディングされたテーブルのレコードが **myserver.mydb.customers_all_shards** トピックにルーティングされます。

schema.name.adjustment.mode

コネクタで使用するメッセージ変換器との互換性のために、結果のトピック名から派生するメッセージキースキーマ名を調整する方法を指定します。値は **none** (デフォルト) または **avro** です。

設定のカスタマイズ

変換で処理する、または処理しないテーブルを指定する [SMT 述語ステートメント](#) を定義して、設定をカスタマイズできます。述語は、正規表現に一致するテーブルをルーティングするように SMT を設定し、正規表現に一致する特定のテーブルを SMT に再ルーティングさせたくない場合に役立ちます。

12.2.3. 同一トピックにルーティングされる Debezium レコード間でのキーの一意性確保

Debezium の変更イベントキーは、テーブルのプライマリーキーを設定するテーブル列を使用します。複数の物理テーブルのレコードを1つのトピックにルーティングするには、それらの全テーブルに渡ってイベントキーが一意でなければなりません。ただし、それぞれの物理テーブルは、そのテーブル内でのみ一意なプライマリーキーを持つことができます。たとえば、**myserver.mydb.customers_shard1** テーブルの行は、**myserver.mydb.customers_shard2** テーブルの行と同じキー値を持つ場合があります。

変更イベントレコードが同じトピックにルーティングされる全テーブルに渡ってそれぞれのイベントキーが必ず一意になるように、トピックルーティング変換は変更イベントキーにフィールドを挿入します。デフォルトでは、挿入されるフィールドの名前は **__dbz__physicalTableIdentifier** です。挿入されるフィールドの値は、デフォルトのルーティング先トピックの名前です。

必要に応じて、別のフィールドをキーに挿入するようにトピックルーティング変換を設定することができます。そのためには、**key.field.name** オプションを指定し、それを既存のプライマリーキーフィールド名と競合しないフィールド名に設定します。以下に例を示します。

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

この例では、ルーティングされるレコードのキー構造に **shard_id** フィールドが追加されます。

キーの新しいフィールドの値を調整する場合は、以下の両方のオプションを設定します。

key.field.regex

1つまたは複数の文字グループをキャプチャーするために、変換がデフォルトのルーティング先トピックの名前に適用する正規表現を指定します。

key.field.replacement

キャプチャーされるこれらのグループに関して、挿入されるキーフィールドの値を決定するための正規表現を指定します。

以下に例を示します。

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

この設定では、デフォルトのルーティング先トピックの名前を以下のように仮定します。

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

変換では、2番目にキャプチャーされたグループの値であるシャード番号が、キーの新しいフィールドの値として使用されます。この例では、挿入されるキーフィールドの値は **1**、**2**、または **3** です。

テーブルにグローバルに一意的なキーが含まれ、キー構造を変更する必要がない場合は、**key.enforce.uniqueness** プロパティを **false** に設定することができます。

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

12.2.4. トピックルーティング変換を一部適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。

以下の方法のいずれかを使用して、SMT を選んで適用するようにコネクタを設定できます。

- [変換用の SMT 述語を設定する](#)。
- SMT に [topic.regex](#) 設定オプションを使用する。

12.2.5. Debezium トピックルーティング変換設定用のオプション

以下の表で、トピックルーティングの SMT 設定オプションを紹介します。

表12.1 トピックルーティングの SMT 設定オプション

オプション	デフォルト	説明
topic.regex		変更イベントレコードを特定のトピックにルーティングする必要があるかどうかを決定するために、変換がそれぞれのレコードに適用する正規表現を指定します。
topic.replacement		ルーティング先トピックの名前を表す正規表現を指定します。変換により、マッチする各レコードがこの式で識別されるトピックにルーティングされます。この式により、 topic.regex に指定する正規表現によってキャプチャーされるグループを参照することができます。グループを参照するには、 \$1 、 \$2 などと指定します。

オプション	デフォルト	説明
<code>key.enforce.uniqueness</code>	<code>true</code>	<p>レコードの変更イベントキーにフィールドを追加するかどうかを定義します。キーフィールドを追加することで、変更イベントレコードが同じトピックにルーティングされる全テーブルに渡って、それぞれのイベントキーの一意性が確保されます。この設定は、同じキーを持つが異なるソーステーブルに由来するレコードの変更イベントの競合を防ぐのに役立ちます。</p> <p>変換でキーフィールドを追加する必要がない場合は、false を指定します。たとえば、パーティションで分割された PostgreSQL テーブルのキーは必ず一意であるため、レコードをパーティションで分割された PostgreSQL テーブルから1つのトピックにルーティングする場合は、key.enforce.uniqueness=false を設定することができます。</p>
<code>key.field.name</code>	<code>__dbz__physicalTableIdentifier</code>	<p>変更イベントキーに追加されるフィールドの名前。このフィールドの値により、元のテーブル名が識別されます。SMT がこのフィールドを追加するには key.enforce.uniqueness が true (デフォルト) である必要があります。</p>
<code>key.field.regex</code>		<p>1つまたは複数の文字グループをキャプチャーするために、変換がデフォルトのルーティング先トピックの名前に適用する正規表現を指定します。SMT がこの正規表現を適用するには、key.enforce.uniqueness をデフォルトの true に設定する必要があります。</p>
<code>key.field.replacement</code>		<p>key.field.regex で指定する正規表現によりキャプチャーされるグループに関して、挿入されるキーフィールドの値を決定するための正規表現を指定します。SMT がこの正規表現を適用するには、key.enforce.uniqueness をデフォルトの true に設定する必要があります。</p>
<code>schema.name.adjustment.mode</code>	<code>none</code>	<p>結果のトピック名から派生したメッセージスキーマ名を、コネクタが使用するメッセージコンバーターとの互換性に合わせて調整する方法を指定します。none は調整 (デフォルト) を適用しません。avro は Avro タイプ名で使用できない文字をアンダースコアに置き換えます。</p>

オプション	デフォルト	説明
<code>logical.table.cache.size</code>	16	LRUCache で最大エンタリーを保持するために使用されるサイズ。キャッシュは、論理テーブルのキーと値の古い/新しいスキーマを保持し、派生キーとトピックの正規表現の結果もキャッシュして、ソースレコードの変換を改善します。

12.3. イベントの内容に応じた変更イベントレコードのトピックへのルーティング

デフォルトでは、Debezium はテーブルから読み取るすべての変更イベントを1つの静的なトピックにストリーミングします。ただし、イベントの内容に応じて、選択したイベントを別のトピックに再ルーティングする必要がある状況が考えられます。メッセージをその内容に基づいてルーティングするプロセスは、[コンテンツベースのルーティング](#) メッセージングパターンで説明されています。このパターンを Debezium に適用するには、コンテンツベースのルーティング [単一メッセージ変換 \(SMT\)](#) を使用して、イベントごとに評価される式を記述します。イベントがどのように評価されるかに応じて、SMT はイベントメッセージを元の宛先トピックにルーティングするか、あるいは式で指定したトピックに再ルーティングします。



重要

Debezium コンテンツベースのルーティング SMT はテクノロジープレビュー機能です。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の機能は、最新の技術をいち早く提供して、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。

カスタム SMT を作成してルーティングロジックをエンコードするのに Java を使用することは可能ですが、カスタムコーディングされた SMT の使用にはデメリットがあります。以下に例を示します。

- 変換を事前にコンパイルし、それを Kafka Connect にデプロイする必要がある。
- 変更が生じるたびにコードの再コンパイルおよび再デプロイが必要になり、運用の柔軟性が失われる。

コンテンツベースのルーティング SMT は、[JSR 223](#) (Scripting for the Java™ Platform) と統合するスクリプト言語をサポートしています。

Debezium には、JSR 223 API の実装は同梱されていません。Debezium で式言語を使用するには、その言語の JSR 223 スクリプトエンジン実装をダウンロードする必要があります。Debezium をデプロイする方法によって、必要な成果物を Maven Central から自動的にダウンロードするか、成果物を手動でダウンロードし、言語実装で使用する他の JAR ファイルと共に Debezium コネクターのプラグインディレクトリーに追加することが可能です。

12.3.1. Debezium コンテンツベースのルーティング SMT の設定

セキュリティ上の理由から、コンテンツベースのルーティング SMT は Debezium コネクタアーカイブには含まれていません。代わりに、別のアーティファクト `debezium-scripting-1.4.2.Final.tar.gz` で提供されます。

Dockerfile からカスタム Kafka Connect コンテナイメージを構築して Debezium コネクタを導入する場合、フィルター SMT を使用するには、Kafka Connect 環境に SMT アーティファクトを明示的に追加する必要があります。AMQ Streams を使用してコネクタをデプロイすると、Kafka Connect カスタムリソースで指定した設定パラメーターに基づいて、必要なアーティファクトを自動的にダウンロードすることができます。重要: ルーティング SMT が Kafka Connect インスタンスに追加されると、インスタンスにコネクタを追加できる任意のユーザーはスクリプト式を実行することができます。許可されたユーザーだけがスクリプト式を実行できるようにするには、ルーティング SMT を追加する前に、Kafka Connect インスタンスおよびその設定インターフェイスをセキュアにする必要があります。

以下の手順は、Dockerfile から Kafka Connect コンテナイメージを構築する場合に適用されます。AMQ Streams を使用して Kafka Connect イメージを作成する場合は、お使いのコネクタのデプロイメントピックに記載されている説明に従ってください。

手順

1. ブラウザーから [Red Hat Integration のダウンロードサイト](#) を開き、Debezium スクリプト SMT アーカイブ (**debezium-scripting-2.1.4.Final.tar.gz**) をダウンロードします。
2. アーカイブのコンテンツを Kafka Connect 環境の Debezium プラグインのディレクトリーに展開します。
3. JSR-223 スクリプトエンジンの実装を取得し、そのコンテンツを Kafka Connect 環境の Debezium プラグインのディレクトリーに追加します。
4. Kafka Connect プロセスを再起動し、新しい JAR ファイルを取得します。

Groovy 言語には、クラスパスで以下のライブラリーが必要です。

- **groovy**
- **groovy-json** (任意)
- **groovy-jsr223**

JavaScript 言語には、クラスパスで以下のライブラリーが必要です。

- **graalvm.js**
- **graalvm.js.scriptengine**

12.3.2. 例: Debezium コンテンツベースルーティングの基本設定

イベントの内容に基づいて変更イベントレコードをルーティングするように Debezium コネクタを設定するには、コネクタの Kafka Connect 設定で **ContentBasedRouter** SMT を設定します。

コンテンツベースのルーティング SMT 設定では、絞り込みの条件を定義する正規表現を指定する必要があります。設定で、ルーティングの条件を定義する正規表現を作成します。式は、イベントレコードを評価するためのパターンを定義します。また、パターンにマッチするイベントをルーティングする宛先トピックの名前も指定します。指定するパターンで、テーブルの挿入、更新、または削除操作などのイベントタイプを指定する場合があります。特定の列または行の値を照合するパターンを定義することもできます。

たとえば、すべての更新 (**u**) レコードを **updates** トピックに再ルーティングするには、コネクタ設定に以下の設定を追加します。

...

```

transforms=route
transforms.route.type=io.debezium.transforms.ContentBasedRouter
transforms.route.language=jsr223.groovy
transforms.route.topic.expression=value.op == 'u' ? 'updates' : null
...

```

上記の例では、**Groovy** 式言語の使用を指定しています。

パターンにマッチしないレコードは、デフォルトのトピックにルーティングされます。

設定のカスタマイズ

前の例は、**op** フィールドを含む DML イベントのみを処理するように設計された単純な SMT 設定を示しています。コネクタが発行する他の種類のメッセージ (ハートビートメッセージ、廃棄メッセージ、またはトランザクションまたはスキーマの変更に関するメタデータメッセージ) には、このフィールドは含まれません。処理の失敗を回避するために、特定のイベントのみに [変換を選択して適用する SMT 述語ステートメント](#) を定義できます。

12.3.3. Debezium コンテンツベースルーティングの式で使用される変数

Debezium は、特定の変数を SMT の評価コンテキストにバインドします。ルーティング先を制御するための条件を指定する式を作成する場合、SMT はこれらの変数の値を検索して解釈し、式の条件を評価することができます。

以下の表に、Debezium がコンテンツベースのルーティング SMT の評価コンテキストにバインドする変数のリストを示します。

表12.2 コンテンツベースルーティングの式で使用される変数

名前	説明	タイプ
鍵 (key)	メッセージのキー。	org.apache.kafka.connect.data.Struct
value	メッセージの値。	org.apache.kafka.connect.data.Struct
keySchema	Schema of the message key.	org.apache.kafka.connect.data.Schema
valueSchema	メッセージの値のスキーマ。	org.apache.kafka.connect.data.Schema
topic	ルーティング先トピックの名前。	文字列

名前	説明	タイプ
ヘッダー	<p>メッセージヘッダーの Java マッピング。キーフィールドはヘッダー名です。headers 変数は、以下のプロパティを公開します。</p> <ul style="list-style-type: none"> ● value (タイプ: Object) ● schema (タイプ: org.apache.kafka.connect.data.Schema) 	<pre>java.util.Map<String, io.debezium .transforms.scripting .RecordHeader></pre>

式は、その変数に対して任意のメソッドを呼び出すことができます。式は、SMT がメッセージをどのように処理するかを定義するブール値に解決する必要があります。式のルーティング条件が **true** と評価されると、メッセージは維持されます。ルーティング条件が **false** と評価されると、メッセージは削除されます。

式がそれ以外の効果を及ぼすことは許されません。つまり、式が渡す変数を変更することは許されません。

12.3.4. コンテンツベースのルーティング変換を一部適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。以下の方法のいずれかを使用して、SMT を選んで適用するようにコネクタを設定できます。

- [変換用の SMT 述語を設定する](#)。
- SMT に [topic.regex](#) 設定オプションを使用する。

12.3.5. 他のスクリプト言語によるコンテンツベースのルーティング条件の設定

コンテンツベースのルーティング条件を記述する方法は、使用するスクリプト言語によって異なります。たとえば、[基本設定の例](#) に示すように、式言語として **Groovy** を使用する場合、以下の式はすべての更新 (**u**) レコードを **updates** トピックにルーティングし、他のレコードをデフォルトのトピックにルーティングします。

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

他の言語では、同じ条件を表すのに異なる方法が使用されます。

ヒント

Debezium MongoDB コネクタは、**after** および **patch** フィールドを構造体ではなくシリアル化された JSON ドキュメントとして出力します。

MongoDB コネクタで ContentBasedRouting SMT を使うには、まず JSON の配列フィールドを個別のドキュメントに展開する必要があります。

式の中で JSON パーサーを使用すると、配列の各項目について個別の出力文書を生成することができます。例えば、表現言語として Groovy を使用している場合、**groovy-json** アーティファクトをクラスパスに追加し、**(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'** のような表現を追加しています。

JavaScript

式言語に JavaScript を使用する場合、以下の例に示すように、**Struct#get()** メソッドを呼び出してコンテンツベースのルーティング条件を指定することができます。

```
value.get('op') == 'u' ? 'updates' : null
```

JavaScript with Graal.js

JavaScript with Graal.js を使用してコンテンツベースのルーティング条件を作成する場合、Groovy で使用する方法と類似の方法を使用します。以下に例を示します。

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

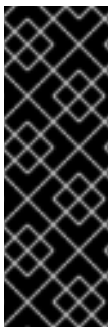
12.3.6. コンテンツベースのルーティング変換設定用のオプション

プロパティ	デフォルト	説明
topic.regex		イベントのルーティング先トピックの名前を評価するオプションの正規表現で、条件ロジックを適用するかどうかを決定します。ルーティング先トピックの名前が topic.regex の値とマッチする場合、変換はイベントをトピックに渡す前に条件ロジックを適用します。トピックの名前が topic.regex の値とマッチしない場合は、SMT は変更せずにイベントをトピックに渡します。
language		式を記述する言語。 jsr223 . で始まる必要があります。例えば、 jsr223.groovy や jsr223.graal.js 。 Debezium では、 JSR 223 API (Scripting for the Java™ Platform) によるブートストラップだけがサポートされます。
topic.expression		すべてのメッセージに対して評価される式。 String 値に評価する必要があり、 null 値以外の場合はメッセージを新しいトピックに再ルーティングし、 null 値の場合はメッセージをデフォルトのトピックにルーティングします。

null.handling.mode	保持	<p>トランスフォーメーションが null (tombstone) メッセージをどのように扱うかを指定します。以下のオプションのいずれかを指定することができます。</p> <p>保持 (デフォルト) メッセージを通過させます。</p> <p>drop メッセージを完全に削除します。</p> <p>evaluate メッセージに条件ロジックを適用します。</p>
---------------------------	----	---

12.4. DEBEZIUM 変更イベントレコードの絞り込み

デフォルトでは、Debezium は受信するすべてのデータ変更イベントを Kafka ブローカーに配信します。ただし、プロデューサーから出力されるイベントのサブセットだけが必要となるケースがほとんどです。該当するレコードだけを処理できるように、Debezium では **フィルター 単一メッセージ変換 (SMT)** を利用することができます。



重要

Debezium フィルター SMT はテクノロジープレビュー機能です。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の機能は、最新の技術をいち早く提供して、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。

カスタム SMT を作成してフィルターロジックをエンコードするのに Java を使用することは可能ですが、カスタムコーディングされた SMT の使用にはデメリットがあります。以下に例を示します。

- 変換を事前にコンパイルし、それを Kafka Connect にデプロイする必要がある。
- 変更が生じるたびにコードの再コンパイルおよび再デプロイが必要になり、運用の柔軟性が失われる。

フィルター SMT は、[JSR 223](#) (Scripting for the Java™ Platform) と統合するスクリプト言語をサポートしています。

Debezium には、JSR 223 API の実装は同梱されていません。Debezium で式言語を使用するには、その言語の JSR 223 スクリプトエンジン実装をダウンロードする必要があります。Debezium をデプロイする方法によって、必要な成果物を Maven Central から自動的にダウンロードするか、成果物を手動でダウンロードし、言語実装で使用する他の JAR ファイルと共に Debezium コネクターのプラグインディレクトリに追加することが可能です。

12.4.1. Debezium フィルター SMT の設定

セキュリティ上の理由から、フィルター SMT は Debezium コネクターアーカイブには含まれていません。代わりに、別のアーティファクト **debezium-scripting-1.4.2.Final.tar.gz** で提供されます。

Dockerfile からカスタム Kafka Connect コンテナイメージを構築して Debezium コネクターをデプロ

イする場合、フィルター SMT を使用するには、明示的に SMT アーカイブをダウンロードし、コネクタプラグインと一緒にファイルをデプロイする必要があります。AMQ Streams を使用してコネクタをデプロイすると、Kafka Connect カスタムリソースで指定した設定パラメーターに基づいて、必要なアーティファクトを自動的にダウンロードすることができます。重要: フィルター SMT が Kafka Connect インスタンスに追加されると、インスタンスにコネクタを追加できる任意のユーザーはスクリプト式を実行することができます。許可されたユーザーだけがスクリプト式を実行できるようにするには、フィルター SMT を追加する前に、Kafka Connect インスタンスおよびその設定インターフェイスをセキュアにする必要があります。

以下の手順は、Dockerfile から Kafka Connect コンテナイメージを構築する場合に適用されます。AMQ Streams を使用して Kafka Connect イメージを作成する場合は、お使いのコネクタのデプロイメントピックに記載されている説明に従ってください。

手順

1. ブラウザーから [Red Hat Integration のダウンロードサイト](#) を開き、Debezium スクリプト SMT アーカイブ (**debezium-scripting-2.1.4.Final.tar.gz**) をダウンロードします。
2. アーカイブのコンテンツを Kafka Connect 環境の Debezium プラグインのディレクトリーに展開します。
3. JSR-223 スクリプトエンジンの実装を取得し、そのコンテンツを Kafka Connect 環境の Debezium プラグインのディレクトリーに追加します。
4. Kafka Connect プロセスを再起動し、新しい JAR ファイルを取得します。

Groovy 言語には、クラスパスで以下のライブラリーが必要です。

- **groovy**
- **groovy-json** (任意)
- **groovy-jsr223**

JavaScript 言語には、クラスパスで以下のライブラリーが必要です。

- **graalvm.js**
- **graalvm.js.scriptengine**

12.4.2. 例: Debezium フィルター SMT の基本設定

Debezium コネクタの Kafka Connect 設定でフィルター変換を設定します。設定で、ビジネスルールに基づくフィルター条件を定義して、対象のイベントを指定します。フィルター SMT がイベントストリームを処理すると、設定されたフィルター条件に対して各イベントを評価します。フィルター条件の基準を満たすイベントのみがブローカーに渡されます。

変更イベントレコードを絞り込むように Debezium コネクタを設定するには、Debezium コネクタの Kafka Connect 設定で **Filter** SMT を設定します。フィルター SMT の設定には、フィルター条件を定義する正規表現を指定する必要があります。

たとえば、コネクタ設定に以下の設定を追加します。

```
...
transforms=filter
transforms.filter.type=io.debezium.transforms.Filter
```



```
transforms.filter.language=jsr223.groovy
transforms.filter.condition=value.op == 'u' && value.before.id == 2
...
```

上記の例では、**Groovy** 式言語の使用を指定しています。正規表現 **value.op == 'u' && value.before.id == 2** は、更新 (**u**) レコードで **id** 値が **2** のメッセージを除き、すべてのメッセージを削除します。

設定のカスタマイズ

前の例は、**op** フィールドを含む DML イベントのみを処理するように設計された単純な SMT 設定を示しています。コネクタが発行する可能性のある他の種類のメッセージ (ハートビートメッセージ、廃棄メッセージ、またはスキーマの変更とトランザクションに関するメタデータメッセージ) には、このフィールドは含まれません。処理の失敗を回避するために、特定のイベントのみに [変換を選択して適用する SMT 述語ステートメント](#) を定義できます。

12.4.3. フィルターの式で使用される変数

Debezium は、特定の変数をフィルター SMT の評価コンテキストにバインドします。フィルター条件を指定する式を作成する場合、Debezium が評価コンテキストにバインドする変数を使用することができます。変数をバインドすることで、Debezium は SMT が式の条件を評価する際に変数の値を検索して解釈できるようにします。

以下の表に、Debezium がフィルター SMT の評価コンテキストにバインドする変数のリストを示します。

表12.3 フィルターの式で使用される変数

名前	説明	タイプ
鍵 (key)	メッセージのキー。	<code>org.apache.kafka.connect.data.Struct</code>
value	メッセージの値。	<code>org.apache.kafka.connect.data.Struct</code>
keySchema	Schema of the message key.	<code>org.apache.kafka.connect.data.Schema</code>
valueSchema	メッセージの値のスキーマ。	<code>org.apache.kafka.connect.data.Schema</code>
topic	ルーティング先トピックの名前。	文字列
ヘッダー	メッセージヘッダーの Java マッピング。キーフィールドはヘッダー名です。 headers 変数は、以下のプロパティを公開します。 <ul style="list-style-type: none"> ● value (タイプ: <code>Object</code>) ● schema (タイプ: <code>org.apache.kafka.connect.data.Schema</code>) 	<code>java.util.Map<String, io.debezium.transforms.scripting.RecordHeader></code>

式は、その変数に対して任意のメソッドを呼び出すことができます。式は、SMT がメッセージをどのように処理するかを定義するブール値に解決する必要があります。式のフィルター条件が **true** と評価されると、メッセージは維持されます。フィルター条件が **false** と評価されると、メッセージは削除されます。

式がそれ以外の効果を及ぼすことは許されません。つまり、式が渡す変数を変更することは許されません。

12.4.4. フィルター変換を一部適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。以下の方法のいずれかを使用して、SMT を選んで適用するようにコネクタを設定できます。

- [変換用の SMT 述語を設定する](#)。
- SMT に `topic.regex` 設定オプションを使用する。

12.4.5. 他のスクリプト言語によるフィルター条件の設定

フィルター条件を記述する方法は、使用するスクリプト言語によって異なります。

たとえば、[基本設定の例](#) に示すように、式言語として **Groovy** を使用する場合、以下の式は **id** 値が **2** に設定された更新レコードを除くすべてのメッセージを削除します。

```
value.op == 'u' && value.before.id == 2
```

他の言語では、同じ条件を表すのに異なる方法が使用されます。

ヒント

Debezium MongoDB コネクタは、**after** および **patch** フィールドを構造体ではなくシリアライズされた JSON ドキュメントとして出力します。

MongoDB コネクタでフィルター SMT を使うには、まず JSON の配列フィールドを個別のドキュメントに展開する必要があります。

式の中で JSON パーサーを使用すると、配列の各項目について個別の出力文書を生成することができます。例えば、表現言語として Groovy を使用している場合、**groovy-json** アーティファクトをクラスパスに追加し、**(new groovy.json.JsonSlurper()).parseText(value.after).last_name == 'Kretchmar'** のような表現を追加しています。

JavaScript

式言語に JavaScript を使用する場合、以下の例に示すように、**Struct#get()** メソッドを呼び出してフィルター条件を指定することができます。

```
value.get('op') == 'u' && value.get('before').get('id') == 2
```

JavaScript with Graal.js

JavaScript with Graal.js を使用してフィルター条件を定義する場合、Groovy で使用する方法と類似の方法を使用します。以下に例を示します。

■

```
value.op == 'u' && value.before.id == 2
```

12.4.6. フィルター変換設定用のオプション

以下の表に、フィルター SMT で使用することができる設定オプションのリストを示します。

表12.4 フィルター SMT の設定オプション

プロパティ	デフォルト	説明
topic.regex		イベントのルーティング先トピックの名前を評価するオプションの正規表現で、フィルターロジックを適用するかどうかを決定します。ルーティング先トピックの名前が topic.regex の値とマッチする場合、変換はイベントをトピックに渡す前にフィルターロジックを適用します。トピックの名前が topic.regex の値とマッチしない場合は、SMT は変更せずにイベントをトピックに渡します。
language		式を記述する言語。 jsr223 で始まる必要があります。例えば、 jsr223.groovy や jsr223.graal.js 。 Debezium では、 JSR 223 API (Scripting for the Java™ Platform) によるブートストラップだけがサポートされます。
condition		すべてのメッセージに対して評価される式。 Boolean 値に評価されなければならない。結果が true の場合はメッセージを保持し、 false の場合はメッセージを削除します。
null.handling.mode	保持	トランスフォーメーションが null (tombstone) メッセージをどのように扱うかを指定します。以下のオプションのいずれかを指定することができます。 保持 (デフォルト) メッセージを通過させます。 drop メッセージを完全に削除します。 evaluate メッセージにフィルター条件を適用します。

12.5. DEBEZIUM の変更イベントからステート AFTER ソースレコードを抽出する

Debezium コネクタは、データ変更メッセージを出力し、ソースデータベースからキャプチャーする各操作を表します。コネクタが Apache Kafka に送信するメッセージの構造は複雑で、元のデータベースイベントの詳細を表します。

この複雑なメッセージ形式は、システムで発生する変更に関する情報を正確に詳述しますが、この形式は一部のダウストリームコンシューマーには適していない場合があります。シンクコネクタ、または Kafka エコシステムの他の部分では、フィールド名と値が単純化されたフラットな構造で表示されるようにフォーマットされたメッセージが必要になる場合があります。

Debezium コネクタが生成するイベントレコードの形式を簡素化するために、Debezium イベントフラット化単一メッセージ変換 (SMT) を使用できます。コネクタが生成するデフォルトの形式よりも単純な形式の Kafka レコードを必要とするコンシューマーをサポートするように変換を設定します。特定のユースケースに合わせて、SMT を Debezium コネクタ、または Debezium コネクタが生成するメッセージを消費するシンクコネクタに適用できます。Apache Kafka が Debezium 変更イベントメッセージを元の形式で保持できるようにするには、シンクコネクタの SMT を設定します。

イベントフラット化変換は [Kafka Connect SMT](#) です。



注記

この章では、Debezium SQL ベースのデータベースコネクタのイベントフラット化シングルメッセージ変換 (SMT) について説明します。Debezium MongoDB コネクタの同等の SMT の詳細は、[MongoDB 新規ドキュメントの状態の抽出](#) を参照してください。

詳細は以下のセクションを参照してください。

- [「Debezium 変更イベントの構造について」](#)
- [「Debezium イベントフラット化変換の動作」](#)
- [「Debezium イベントフラット化変換の設定」](#)
- [「Kafka レコードに Debezium メタデータを追加する例」](#)
- [「Debezium イベントフラット化変換設定用のオプション」](#)

12.5.1. Debezium 変更イベントの構造について

Debezium は、複雑な構造を持つデータ変更イベントを生成します。それぞれイベントは、以下の 3 つの部分で設定されます。

- 以下の項目が含まれるメタデータ (ただし、これらに限定されません)
 - データを変更した操作のタイプ。
 - データベースの名前や、変更が発生したテーブルなどのソース情報。
 - 変更が加えられたタイミングを識別するタイムスタンプ。
 - (任意の項目) トランザクション情報
- 変更前の行データ
- 変更後の行データ

以下の例は、**UPDATE** 変更イベントのメッセージ構造の一部を示しています。

```
{
  "op": "u",
  "source": {
```

```

...
},
"ts_ms": "...",
"before": {
  "field1": "oldvalue1",
  "field2": "oldvalue2"
},
"after": {
  "field1": "newvalue1",
  "field2": "newvalue2"
}
}

```

コネクタの変更イベント構造の詳細は、コネクタのドキュメントを参照してください。

イベントフラット化 SMT が前の例のメッセージを処理した後、メッセージ形式が単純化され、次にあるメッセージが生成されます。

```

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

```

12.5.2. Debezium イベントフラット化変換の動作

イベントフラットニング SMT は、Kafka レコードの Debezium 変更イベントから **after** フィールドを抽出します。SMT は元の変更イベントを **after** フィールドのみで置き換え、シンプルな Kafka レコードを作成します。

Debezium コネクタまたは Debezium コネクタから出力されるメッセージを使用するシンクコネクタに、イベントフラット化 SMT を設定することができます。シンクコネクタにイベントフラット化を設定するメリットは、Apache Kafka に保存されるレコードに Debezium の変更イベント全体が含まれることです。SMT を元のコネクタまたはシンクコネクタに適用するかどうかの判断は、特定のユースケースによります。

以下の操作のいずれかを実行するように変換を設定することができます。

- 変更イベントからのメタデータを簡素化した Kafka レコードに追加する。デフォルト動作では、SMT はメタデータを追加しません。
- **DELETE** 操作の変更イベントを含む Kafka レコードをストリームに保持します。デフォルトの動作は、SMT が **DELETE** 操作変更イベントの Kafka レコードをドロップするというもので、ほとんどのコンシューマーがまだ処理できないためです。

データベースの **DELETE** 操作により、Debezium は 2 つの Kafka レコードを生成します。

- **"op": "d"**、**before** 行のデータ、その他のフィールドが含まれるレコード。
- 削除された行と同じキーを持ち、値が **null** である墓石のレコード。このレコードは Apache Kafka のマーカーです。これは、[ログコンパクション](#)によりこのキーを持つすべてのレコードが削除されることを意味します。

before 行のデータを含むレコードをドロップする代わりに、イベントフラットニング SMT が以下のいずれかを行うように設定することができます。

- ストリーム内のレコードを保持し、**"value": "null"** フィールドのみを持つように編集します。

- レコードをストリームに維持し、追加した "**__deleted**": "true" エントリーと共に **before** フィールドに含まれていたキー/値のペアが含まれる **value** フィールドを持つようにそのレコードを編集する。

同様に、tombstone レコードを破棄する代わりに、イベントフラット化 SMT を設定して tombstone レコードをストリームに維持することができます。

12.5.3. Debezium イベントフラット化変換の設定

コネクタの設定に SMT 設定の詳細を追加して、Kafka Connect ソースコネクタまたはシンクコネクタに Debezium イベントフラット化 SMT を設定します。たとえば、変換のデフォルトの動作を取得するには、次の例のように、オプションを指定せずにコネクタ設定に追加します。

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

他の Kafka Connect のコネクタ設定と同様に、**transforms=** にコンマで区切られた複数の SMT エイリアスを設定し、Kafka Connect に SMT を適用させたい順番に設定することができます。

次の **.properties** の例では、いくつかのイベントフラット化 SMT オプションを設定しています。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.transforms.ExtractNewRecordState
transforms.unwrap.drop.tombstones=false
transforms.unwrap.delete.handling.mode=rewrite
transforms.unwrap.add.fields=table,lsn
```

drop.tombstones=false

イベントストリームに **DELETE** 操作の墓石の記録を残します。

delete.handling.mode=rewrite

DELETE 操作では、変更イベントにあった **value** フィールドをフラット化することで、Kafka レコードを編集します。**value** フィールドには、**before** フィールドにあったキーと値のペアが直接入ります。SMT では、例えば **__deleted** を追加して、それを **true** に設定します。

```
"value": {
  "pk": 2,
  "cola": null,
  "__deleted": "true"
}
```

add.fields=table,lsn

table および **lsn** フィールドの変更イベントメタデータを簡素化した Kafka レコードに追加します。

設定のカスタマイズ

コネクタは、多くの種類のイベントメッセージ (ハートビートメッセージ、廃棄メッセージ、またはトランザクションまたはスキーマの変更に関するメタデータメッセージ) を発行する場合があります。イベントのサブセットに変換を適用するには、特定のイベントだけを対象にして [変換を選択して適用する SMT 述語ステートメント](#) を定義できます。

12.5.4. Kafka レコードに Debezium メタデータを追加する例

イベントフラット化 SMT を設定して、元の変更イベントメタデータを簡素化した Kafka レコードに追加できます。たとえば、簡素化したレコードのヘッダーまたは値に、次のいずれかの項目を含めることができます。

- 変更を加えた操作のタイプ
- データベースまたは変更が加えられたテーブルの名前
- Postgres LSN フィールド等のコネクタ固有のフィールド

簡略化された Kafka レコードのヘッダーにメタデータを追加するには、**add.header** オプションを指定します。簡略化された Kafka レコードの値にメタデータを追加するには、**add.fields** オプションを指定します。これらのオプションには、それぞれ変更イベントフィールド名のコンマ区切りリストを設定します。スペースは指定しないでください。フィールド名が重複している場合、それらのフィールドの1つのメタデータを追加するには、フィールドと共に構造体を指定します。以下に例を示します。

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

この設定では、簡素化した Kafka レコードには以下のような内容が含まれます。

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

また、簡略化された Kafka のレコードには、**__db** ヘッダーが付いています。

簡素化した Kafka レコードでは、SMT はメタデータフィールド名の前にダブルアンダースコアを追加します。また、構造体を指定すると、SMT は構造体名とフィールド名の間アンダースコアを挿入します。

DELETE 操作のシンプルな Kafka レコードにメタデータを追加するには、**delete.handling.mode=rewrite** も設定する必要があります。

12.5.5. イベントフラット化変換を選択的に適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。

SMT を選択的に適用する方法の詳細は、[変換用の SMT 述語の設定](#) を参照してください。

12.5.6. Debezium イベントフラット化変換設定用のオプション

次の表で、イベントフラット化 SMT を設定する際に指定することのできるオプションを説明します。

表12.5 イベントフラット化 SMT 設定オプションの説明

オプション	デフォルト	説明
<code>drop.tombstones</code>	<code>true</code>	<p>Debezium は、DELETE 操作ごとに廃棄レコードを生成します。デフォルト動作では、イベントフラット化 SMT はストリームからトゥームストーンレコードを削除します。廃棄レコードをストリームに残すには、drop.tombstones=false を指定します。</p>
<code>delete.handling.mode</code>	<code>drop</code>	<p>Debezium は、DELETE 操作ごとに変更イベントレコードを生成します。デフォルト動作では、イベントフラット化 SMT はストリームからこれらのレコードを削除します。DELETE 操作の Kafka レコードをストリームに残すには、delete.handling.mode を <code>none</code> または <code>rewrite</code> に設定します。</p> <p>ストリームに変更イベントの記録を残す場合は、<code>none</code> を指定します。レコードには <code>"value": "null"</code> のみが含まれています。</p> <p><code>rewrite</code> を指定して変更イベントレコードをストリームに残し、レコードを編集して、<code>before</code> フィールドにあったキーと値のペアを含む <code>値</code> フィールドを作成し、さらに <code>__deleted: true</code> を <code>value</code> に追加します。これは、レコードが削除されていることを示す別の方法です。</p> <p><code>rewrite</code> を指定すると、DELETE 操作の更新された簡素化したレコードだけで、削除されたレコードを追跡できます。Debezium コネクターが作成するトゥームストーンレコードをドロップするデフォルトの動作を受け入れることを検討できます。</p>

オプション	デフォルト	説明
route.by.field		<p>行データを使用してレコードをルーティングするトピックを決定するには、このオプションを after フィールド属性に設定します。SMT は、指定した after フィールド属性の値にマッチする名前のトピックにレコードをルーティングします。DELETE 操作の場合は、このオプションを before フィールド属性に設定します。</p> <p>たとえば、設定が route.by.field=destination の場合、名前が after.destination の値のトピックにレコードがルーティングされます。Debezium コネクタのデフォルト動作では、名前がデータベースおよび変更が加えられたテーブルの名前で設定されるトピックに、それぞれの変更イベントレコードが送信されます。</p> <p>シンクコネクタにイベントフラット化 SMT を設定する場合、このオプションを設定すると、ルーティング先トピックの名前が簡素化した変更イベントレコードで更新されるデータベーステーブルの名前に優先する場合に役立ちます。トピック名が実際のユースケースに適しない場合は、route.by.field を設定してイベントを再ルーティングすることができます。</p>
add.fields.prefix	__ (double-underscore)	このオプションの文字列を設定して、フィールドに接頭辞を設定します。

オプション	デフォルト	説明
add.fields	デフォルト値なし	<p>このオプションをメタデータフィールドのコンマ区切りリスト (スペースなし) に設定し、簡素化した Kafka レコードの値に追加します。フィールド名が重複している場合、それらのフィールドの1つのメタデータを追加するには、フィールドと共に構造体を指定します (例: source.ts_ms)。</p> <p>オプションとして、<field name>:<new field name> でフィールド名を上書きできます。たとえば、新しいフィールド名は version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP のようになります。new field name は、大文字と小文字が区別されることに注意してください。</p> <p>SMT が簡素化したレコードの値にメタデータフィールドを追加する場合、それぞれのメタデータフィールド名の前にダブルアンダースコアが追加されます。構造体の指定に関して、SMT は構造体名とフィールド名の間にもアンダースコアを挿入します。</p> <p>変更イベントレコードにないフィールドを指定した場合でも、SMT はレコードの値にそのフィールドを追加します。</p>
add.headers.prefix	__ (double-underscore)	このオプションの文字列を設定して、ヘッダーに接頭辞を設定します。

オプション	デフォルト	説明
<code>add.headers</code>	デフォルト値なし	<p>このオプションをメタデータフィールドのコンマ区切りリスト (スペースなし) に設定し、簡素化した Kafka レコードのヘッダーに追加します。フィールド名が重複している場合、それらのフィールドの1つのメタデータを追加するには、フィールドと共に構造体を指定します (例: <code>source.ts_ms</code>)。</p> <p>オプションとして、<code><field name>:<new field name></code> でフィールド名を上書きできます。たとえば、新しいフィールド名は <code>version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP</code> のようになります。<code>new field name</code> は、大文字と小文字が区別されることに注意してください。</p> <p>SMT が簡素化したレコードのヘッダーにメタデータフィールドを追加する場合、それぞれのメタデータフィールド名の前にダブルアンダースコアが追加されます。構造体の指定に関して、SMT は構造体名とフィールド名の間にもアンダースコアを挿入します。</p> <p>変更イベントレコードにないフィールドを指定した場合、SMT はヘッダーにそのフィールドを追加しません。</p>

12.6. DEBEZIUM MONGODB 変更イベントから AFTER 状態のソースドキュメント抽出

Debezium MongoDB コネクタは、MongoDB コレクションで発生する各操作を表すデータ変更メッセージを出力します。これらのイベントメッセージの複雑な構造は、元のデータベースイベントの詳細を忠実に表しています。ただし、ダウンストリームのコンシューマーの中には、メッセージを元の形式で処理できない場合があります。たとえば、データコレクション内のネストされたドキュメントを表すために、コネクタは、ネストされたフィールドを含む形式でイベントメッセージを出力します。シンクコネクタ、または元のメッセージの階層形式を処理できない他のコンシューマーをサポートするには、Debezium MongoDB イベントフラット化 (ExtractNewDocumentState) の単一メッセージ変換 (SMT) を使用できます。SMT は元のメッセージの構造を簡素化し、他の方法でメッセージを変更してデータを処理しやすくすることができます。

イベントフラット化変換は [Kafka Connect SMT](#) です。



注記

この章では、Debezium MongoDB ベースのデータベースコネクタのイベントフラット化シングルメッセージ変換 (SMT) について説明します。リレーショナルデータベースで使用する同等の SMT の詳細は、[新記録の状態抽出 SMT のドキュメント](#) を参照してください。

詳細は以下のセクションを参照してください。

- [「Debezium MongoDB 変更イベントの構造の説明」](#)
- [「Debezium MongoDB イベントフラット化変換の動作」](#)
- [「Debezium MongoDB イベントフラット化変換の設定」](#)
- [「MongoDB イベントメッセージで配列をエンコードするためのオプション」](#)
- [「MongoDB イベントメッセージでネストされた構造のフラット化」](#)
- [「Debezium MongoDB コネクターが **\\$unset** 操作によって削除されたフィールドの名前を報告する方法」](#)
- [「元のデータベース操作のタイプの判別」](#)
- [「MongoDB イベントフラット化 SMT を使用した Debezium メタデータの Kafka レコード追加」](#)
- [「MongoDB 抽出の新しいドキュメント状態変換を選択して適用するオプション」](#)
- [「MongoDB の Debezium イベントフラット化変換の設定オプション」](#)
- [既知の制限](#)

12.6.1. Debezium MongoDB 変更イベントの構造の説明

Debezium MongoDB コネクターは、構造が複雑な変更イベントを生成します。各イベントメッセージには、以下の部分が含まれます。

ソースメタデータ

以下のフィールドが含まれますが、これに限定されません。

- コレクション内のデータを変更した操作のタイプ (作成/挿入、更新、または削除)。
- 変更が発生したデータベースおよびコレクションの名前。
- 変更が加えられたタイミングを識別するタイムスタンプ。
- (任意の項目) トランザクション情報

ドキュメントデータ

before データ

このフィールドは、Debezium コネクターの **capture.mode** が以下のいずれかの値に設定されている場合に MongoDB 6.0 以降を実行する環境に存在します。

- **change_streams_with_pre_image.**
- **change_streams_update_full_with_pre_image.**
詳細は、[MongoDB プレイメージサポート](#) を参照してください。

After データ

現在の操作後にドキュメントに存在する値を表す JSON 文字列。イベントメッセージに **after** フィールドが存在するかどうかは、イベントの種類とコネクター設定によって異なります。

MongoDB **insert** 操作の **create** イベントには、**capture.mode** 設定に関係なく、常に **after** フィールドが含まれます。**update** イベントの場合、**after** フィールドは **capture.mode** が以下のいずれかの値に設定されている場合にのみ存在します。

- **change_streams_update_full**
- **change_streams_update_full_with_pre_image.**



注記

変更イベントメッセージの **after** 値は、必ずしもイベント直後のドキュメントの状態を表すとは限りません。値は動的に計算されず、コネクタが変更イベントをキャプチャーした後、コレクションをクエリーしてドキュメントの現在の値を取得します。

たとえば、複数の操作 **a**、**b**、および **c** がドキュメントを立て続けに変更する状況を想像してみてください。コネクタが変更 **a** を処理するとき、全ドキュメントのコレクションをクエリーします。その間に **b** と **c** の変更が発生します。コネクタが変更 **a** に関する全ドキュメントのクエリーに対する応答を受信すると、後に続く **b** または **c** の変更をもとにしたドキュメントのバージョンを受信する可能性があります。詳細は、**capture.mode** プロパティのドキュメントを参照してください。

以下のフラグメントは、MongoDB の **insert** 操作後にコネクタが発行する **create** 変更イベントの基本構造を示しています。

```
{
  "op": "c",
  "after": {"field1": "newvalue1", "field2": "newvalue1"},
  "source": { ... }
}
```

前の例の **after** フィールドの複雑な形式では、ソースデータベースで発生した変更に関する詳細情報を提供します。ただし、一部のコンシューマーは、ネストされた値が含まれるメッセージを処理できません。元のメッセージの複雑なネストされたフィールドを、より単純で普遍的、かつ互換性のある構造に変換するには、MongoDB のイベントフラット化 SMT を使用します。SMT は、以下の例のようにメッセージ内のネストされたフィールドの構造をフラット化します。

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

Debezium MongoDB コネクタによって生成されたメッセージのデフォルト構造の詳細は、[コネクタのドキュメント](#) を参照してください。

12.6.2. Debezium MongoDB イベントフラット化変換の動作

MongoDB のイベントフラット化 SMT は、Debezium MongoDB コネクタによって発行された **create** または **update** 変更イベントメッセージから **after** フィールドを抽出します。SMT は、元の変更イベントメッセージを処理した後、**after** フィールドの内容のみを含む、単純化バージョンを生成します。

ユースケースに応じて、ExtractNewDocumentState SMT を Debezium MongoDB コネクタ、または

Debezium コネクタが生成するメッセージを消費するシンクコネクタに適用できます。SMT を Debezium MongoDB コネクタに適用すると、SMT はコネクタが Apache Kafka に送信する前に出力するメッセージを変更します。Kafka が完全な Debezium 変更イベントメッセージを元の形式で確実に保持するには、SMT をシンクコネクタに適用します。

イベントフラット化 SMT を使用して MongoDB コネクタから送信されたメッセージを処理すると、SMT は元のメッセージのレコードの構造を、通常のシンクコネクタで使用できるように、適切に型指定された Kafka Connect レコードに変換します。たとえば、SMT は元のメッセージの **after** 情報を表す JSON 文字列を、任意のコンシューマーが処理できるスキーマ構造に変換します。

オプションとして、MongoDB のイベントフラット化 SMT を設定して、処理中に他の方法でメッセージを変更できます。詳細は [設定トピック](#) を参照してください。

12.6.3. Debezium MongoDB イベントフラット化変換の設定

Debezium MongoDB コネクタによって発行されたメッセージを消費するシンクコネクタ用に、MongoDB のイベントフラット化 (ExtractNewDocumentState) SMT を設定します。

詳細は以下のセクションを参照してください。

- [「例: Debezium MongoDB イベントフラット化変換の基本設定」](#)
- [「MongoDB イベントメッセージで配列をエンコードするためのオプション」](#)
- [「MongoDB イベントメッセージでネストされた構造のフラット化」](#)
- [「Debezium MongoDB コネクタが **\\$unset** 操作によって削除されたフィールドの名前を報告する方法」](#)
- [「元のデータベース操作のタイプの判別」](#)
- [「MongoDB イベントフラット化 SMT を使用した Debezium メタデータの Kafka レコード追加」](#)
- [「MongoDB 抽出の新しいドキュメント状態変換を選択して適用するオプション」](#)
- [「MongoDB の Debezium イベントフラット化変換の設定オプション」](#)

12.6.3.1. 例: Debezium MongoDB イベントフラット化変換の基本設定

SMT のデフォルト動作を取得するには、以下の例のように、オプションを指定せずにシンクコネクタの設定に SMT を追加します。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
```

Kafka Connect コネクタ設定と同様に、**transforms=** を複数のコンマ区切りの SMT エイリアスに設定できます。Kafka Connect は、リスト順で指定した変換を適用します。

MongoDB イベントフラット化 SMT を使用するコネクタに複数のオプションを設定できます。以下の例は、コネクタの **drop.tombstones**、**delete.handling.mode**、および **add.headers** オプションを指定する設定です。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
transforms.unwrap.drop.tombstones=false
```

```
transforms.unwrap.delete.handling.mode=drop
transforms.unwrap.add.headers=op
```

前の例の設定オプションの詳細は、[設定トピック](#) を参照してください。

設定のカスタマイズ

コネクタは、多くの種類のイベントメッセージ (ハートビートメッセージ、廃棄メッセージ、またはトランザクションの変更に関するメタデータメッセージ) を発行する場合があります。イベントのサブセットに変換を適用するには、特定のイベントだけを対象にして [変換を選択して適用する SMT 述語ステートメント](#) を定義できます。

12.6.4. MongoDB イベントメッセージで配列をエンコードするためのオプション

デフォルトでは、イベントフラット化 SMT は MongoDB 配列を Apache Kafka Connect または Apache Avro スキーマと互換性のある配列に変換します。MongoDB 配列には複数のタイプの要素を含めることができますが、Kafka 配列のすべての要素が同じタイプである必要があります。

SMT が環境のニーズを満たす方法で配列をエンコードするようにするには、[array.encoding](#) 設定オプションを指定できます。以下の例は、配列エンコーディングを指定するための設定を示しています。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
transforms.unwrap.array.encoding=<array|document>
```

設定に応じて、SMT は、次のエンコード方法のいずれかを使用して、ソースメッセージ内の配列の各インスタンスを処理します。

配列エンコーディング

[array.encoding](#) が **array** (デフォルト) に設定されている場合、SMT エンコードは **array** データ型を使用して元のメッセージの配列をエンコードします。正しい処理を確保するには、配列インスタンスのすべての要素が同じタイプである必要があります。このオプションで制限が課されることとなりますが、ダウンストリームクライアントが配列を簡単に処理できるようにします。

ドキュメントのエンコーディング

[array.encoding](#) が **document** に設定されている場合に、[BSON シリアル化](#) と同様の方法で、ソースの各配列を **structs** の中の **struct** に変換します。メイン **struct** には、**_0**、**_1**、**_2** などのフィールドが含まれます。各フィールド名は、元の配列内の要素のインデックスを表します。SMT は、これらの各インデックスフィールドに、ソース配列内の同等の要素に対して取得した値を設定します。Avro エンコーディングでは数字で始まるフィールド名が禁止されているため、インデックス名にはアンダースコアが接頭辞として付けられます。

次の例は、Debezium MongoDB コネクタが、異種データ型で構成される配列を含むデータベースドキュメントを表す方法を示しています。

例12.1 例: 複数のデータ型を含む配列のドキュメントエンコーディング

```
{
  "_id": 1,
  "a1": [
    {
      "a": 1,
      "b": "none"
    },
    {
```

```

    "a": "c",
    "d": "something"
  }
]
}

```

array.encoding が **document** に設定されている場合、SMT は前述のドキュメントを以下の形式に変換します。

```

{
  "_id": 1,
  "a1": {
    "_0": {
      "a": 1,
      "b": "none"
    },
    "_1": {
      "a": "c",
      "d": "something"
    }
  }
}

```

document エンコーディングオプションを使用すると、SMT は異種要素で設定される任意の配列を処理できます。ただし、このオプションを使用する前に、シンクコネクタと他のダウンストリームコンシューマーが複数のデータ型を含む配列を処理できることを常に確認してください。

12.6.5. MongoDB イベントメッセージでネストされた構造のフラット化

データベース操作に埋め込みドキュメントが含まれる場合、Debezium MongoDB コネクタは、元のドキュメントの階層構造を反映する Kafka イベントレコードを発行します。つまり、イベントメッセージは、ネストされたフィールド構造のセットとして、ネスト化ドキュメントを表します。ダウンストリームコネクタがネスト化された構造が含まれるメッセージを処理できない環境では、イベントフラット化 SMT を設定して、メッセージ内の階層構造をフラット化できます。フラットなメッセージ構造は、テーブルのようなストレージに適しています。

ネストされた構造をフラット化するように SMT を設定するには、**flatten.struct** 設定オプションを **true** に設定します。変換されたメッセージでは、フィールド名がドキュメントソースと一致するよう構築されます。SMT は、親ドキュメントフィールドの名前とネストされたドキュメントフィールドの名前を連結して、フラット化された各フィールドの名前を変更します。名前のコンポーネントを区切るには、**flatten.struct.delimiter** オプションで定義される区切り文字を使用します。**struct.delimiter** のデフォルト値はアンダースコア文字 (`_`) です。

次の例は、SMT がネストされた構造をフラット化するかどうかを指定する設定です。

```

transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
transforms.unwrap.flatten.struct=<true|false>
transforms.unwrap.flatten.struct.delimiter=<string>

```

以下の例は、MongoDB コネクタが出力するイベントメッセージを示しています。メッセージには、ネストされた 2 つのドキュメント **b** と **c** のフィールドを含むドキュメント **a** のフィールドが含まれます。

-


```
{
  "_id": 1,
  "a": {
    "b": 1,
    "c": "none"
  },
  "d": 100
}
```

以下の例のメッセージは、MongoDB の SMT が前のメッセージでネスト化された構造をフラット化した後に表示した出力です。

```
{
  "_id": 1,
  "a_b": 1,
  "a_c": "none",
  "d": 100
}
```

結果のメッセージでは、元のメッセージでネストされた **b** フィールドと **c** フィールドがフラット化され、名前が変更されます。名前が変更されたフィールドは、親ドキュメント **a** の名前を、ネストされたドキュメントの名前 **a_b** および **a_c** と連結することによって形成されます。新しいフィールド名のコンポーネントは、**struct.delimiter** 設定プロパティの設定で定義されているようにアンダースコアで区切られます。

12.6.6. Debezium MongoDB コネクタが \$unset 操作によって削除されたフィールドの名前を報告する方法

MongoDB では、**\$unset** Operator と **\$rename** Operator の両方がドキュメントからフィールドを削除します。MongoDB コレクションはスキーマレスであるため、更新でドキュメントからフィールドが削除されると、更新済みドキュメントで抜けているフィールドの名前を推測できません。Debezium は、削除されたフィールドの名前をリストする **removedFields** 要素を含む更新メッセージを発行して、削除済みフィールドに関する情報を必要とする可能性があるシンクコネクタまたはその他のコンシューマーをサポートします。

以下の例は、フィールド **a** を削除する操作の更新メッセージの一部を示しています。

```
"payload": {
  "op": "u",
  "ts_ms": "...",
  "before": "{ ... }",
  "after": "{ ... }",
  "updateDescription": {
    "removedFields": ["a"],
    "updatedFields": null,
    "truncatedArrays": null
  }
}
```

上記の例では、**before** および **after** はドキュメントが更新される前と後のソースドキュメントの状態を表します。これらのフィールドは、コネクタの **capture.mode** が次のリストで説明されているように設定されている場合にのみ、コネクタが発行するイベントメッセージに存在します。

before フィールド

変更前のドキュメントの状態を提供します。このフィールドは、**capture.mode** が以下のいずれかの値に設定されている場合にのみ表示されます。

- **change_streams_with_pre_image**
- **change_streams_update_full_with_pre_image.**

After フィールド

変更後のドキュメントの完全な状態を提供します。このフィールドは、**capture.mode** が以下のいずれかの値に設定されている場合にのみ表示されます。

- **change_streams_update_full**
- **change_streams_update_full_with_pre_image.**

完全なドキュメントをキャプチャーするようにコネクタが設定されていると仮定すると、**ExtractNewDocumentState** SMT が **\$unset** イベントの **update** メッセージを受信すると、SMT は、次の例に示すように、削除されたフィールドに **null** 値があることを表し、メッセージを再エンコードします。

```
{
  "id": 1,
  "a": null
}
```

完全なドキュメントをキャプチャーするように設定されていないコネクタの場合、SMT が **\$unset** 操作の更新イベントを受信すると、以下の出力メッセージが生成されます。

```
{
  "a": null
}
```

12.6.7. 元のデータベース操作のタイプの判別

SMT がイベントメッセージをフラット化した後、生成されるメッセージでは、イベントを生成した操作が **create**、**update**、または初期スナップショットの **read** であるかを判断できなくなります。通常、**delete** 操作に関する情報を公開するか、削除に伴うイベントを書き換えるようにコネクタを設定して、削除操作を識別できます。イベントメッセージで廃棄および書き換えに関する情報を公開するようにコネクタを設定する方法は、**drop.tombstones** および **delete.handling.mode** プロパティを参照してください。

イベントメッセージでデータベース操作のタイプを報告させるには、SMT は次の要素のいずれかに **op** フィールドを追加します。

- イベントメッセージのボディ。
- メッセージヘッダー

たとえば、元の操作のタイプを示すヘッダープロパティを追加するには、次の例のように、変換を追加してから、**add.headers** プロパティをコネクタ設定に追加します。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
transforms.unwrap.add.headers=op
```

上記の設定に基づいて、SMT は **op** ヘッダーをメッセージに追加し、文字列の値を割り当てて操作のタイプを特定することでイベントタイプを報告します。割り当てられた文字列の値は、元の [MongoDB 変更イベントメッセージ](#) の **op** フィールドの値に基づいています。

12.6.8. MongoDB イベントフラット化 SMT を使用した Debezium メタデータの Kafka レコード追加

MongoDB のイベントフラット化 SMT は、元の変更イベントメッセージから簡素化されたメッセージにメタデータフィールドを追加できます。追加されたメタデータフィールドには、2つのアンダースコア ("__") が接頭辞として付けられます。イベントレコードにメタデータを追加すると、変更イベントが発生したコレクションの名前などのコンテンツを含めたり、レプリカセット名などのコネクタ固有のフィールドを含めたりすることができます。現在、SMT は、**source**、**transaction**、および **updateDescription** の変更イベントのサブ構造フィールドのみを追加できます。

MongoDB 変更イベントの構造に関する詳細は、[MongoDB コネクターのドキュメント](#) を参照してください。

たとえば、次の設定を指定して、変更イベントのレプリカセット名 (**rs**) とコレクション名を、最終的なフラット化されたイベントレコードに追加できます。

```
transforms=unwrap,...
transforms.unwrap.type=io.debezium.connector.mongodb.transforms.ExtractNewDocumentState
transforms.unwrap.add.fields=rs,collection
```

上記の設定により、フラット化されたレコードに次のコンテンツが追加されます。

```
{ "__rs" : "rs0", "__collection" : "my-collection", ... }
```

SMT でメタデータフィールドを **delete** イベントに追加する場合には、**delete.handling.mode** オプションの値を **rewrite** に設定します。

12.6.9. MongoDB 抽出の新しいドキュメント状態変換を選択して適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。

SMT を選択的に適用する方法の詳細は、[変換用の SMT 述語の設定](#) を参照してください。

12.6.10. MongoDB の Debezium イベントフラット化変換の設定オプション

以下の表は、MongoDB イベントフラット化 SMT の設定オプションを示しています。

プロパティ	デフォルト	説明
-------	-------	----

プロパティ	デフォルト	説明
array.encoding	array	<p>SMT が元のイベントメッセージから読み取る配列をエンコードする際に使用する形式を指定します。以下のオプションのいずれかを設定します。</p> <p>array</p> <p>SMT は array データ型を使用して、MongoDB 配列を Apache Kafka Connect または Apache Avro スキーマと互換性のある形式にエンコードします。このオプションを設定する場合は、各配列インスタンスの要素が同じタイプであることを確認します。MongoDB では複数のデータ型を含めることができますが、ダウンストリームクライアントによっては配列を処理できません。</p> <p>document</p> <p>SMT は、BSON シリアル化 と同様の方法で、各 MongoDB 配列を structs の struct に変換します。メインの struct には、_0、_1、_2 などの名前前のフィールドが含まれています。Avro の命名標準に準拠するために、SMT は各インデックスフィールドの数値名の前にアンダースコアを付けます。各数値フィールド名は、元の配列にある要素のインデックスを表します。SMT は、これらの各インデックスフィールドに、指定された配列要素のソースドキュメントから取得した値を設定します。</p> <p>array.coding オプションの詳細は、MongoDB イベントメッセージで配列をエンコードするためのオプション を参照してください。</p>
flatten.struct	false	<p>SMT は、メッセージ内のネストされたプロパティの名前を設定可能な区切り文字で連結して、単純なフィールド名を形成することで、元のイベントメッセージの構造 (structs) をフラット化します。</p>
flatten.struct.delimiter	—	<p>flatten.struct が true に設定されている場合、出力レコードにフィールド名を生成するために、変換が入力レコードから連結するフィールド名の間に入挿入する区切り文字を指定します。</p>

プロパティ	デフォルト	説明
drop.tombstones	true	Debezium は、 DELETE 操作ごとに tombstone レコードを生成します。デフォルト動作では、イベントフラット化 SMT はストリームからトゥームストーンレコードを削除します。tombstone レコードをストリームに残すには、 drop.tombstones=false を指定します。
delete.handling.mode	drop	<p>SMT が delete 操作用に生成する変更イベントレコードを処理する方法を指定します。以下のオプションのいずれかを設定します。</p> <p>drop</p> <p>SMT は、イベントストリームから delete 操作のレコードを削除します。</p> <p>none</p> <p>SMT は、イベントストリームからの元の変更イベントレコードを保持します。レコードには "value": "null" のみが含まれています。</p> <p>rewrite</p> <p>SMT は、ストリームからの変更イベントレコードの変更バージョンを保持します。レコードが削除されたことを示す別の方法を提供するために、変更されたレコードには、元のレコードからのキーと値のペアを含む value フィールドなど、value に __deleted: true が追加されます。</p> <p>rewrite オプションを設定すると、削除されたレコードを追跡するには、DELETE 操作の更新された単純化されたレコードで十分であると感じる場合があります。このような場合には、SMT で tombstone レコードを削除する が推奨されます。</p>
add.headers.prefix	__ (double-underscore)	このオプションの文字列を設定して、ヘッダーに接頭辞を設定します。

プロパティ	デフォルト	説明
add.headers	デフォルトなし	<p>SMT で簡略化されたメッセージのヘッダーに追加するメタデータフィールドのコンマ区切りのリストを、スペースを入れずに指定します。元のメッセージに重複したフィールド名が含まれている場合、struct の名前とフィールドの名前を一緒に指定することで、変更する特定のフィールドを識別できます (例: source.ts_ms)。</p> <p>必要に応じて、以下の形式のエントリをリストに追加し、フィールドの元の名前を上書きして新しい名前を割り当てることができます。</p> <p><field_name>:<new_field_name>.</p> <p>以下に例を示します。</p> <pre>version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP</pre> <p>指定する新しい名前の値では、大文字と小文字が区別されます。</p> <p>SMT が簡略化したメッセージのヘッダーにメタデータフィールドを追加するとき、SMT は各メタデータフィールド名の前に 2 つのアンダースコアを付けます。構造体の指定に関して、SMT は構造体名とフィールド名の間にもアンダースコアを挿入します。</p> <p>変更イベントの元のメッセージにないフィールドを指定した場合、SMT ではヘッダーにそのフィールドが追加されません。</p>
add.fields.prefix	__(double-underscore)	フィールド名の前に付けるオプションの文字列を指定します。

プロパティ	デフォルト	説明
<code>add.fields</code>	デフォルトなし	<p>このオプションを、簡略化された Kafka メッセージの value 要素に追加するメタデータフィールドのコンマ区切りリスト (スペースなし) に設定します。元のメッセージに重複したフィールド名が含まれている場合、struct の名前とフィールドの名前を一緒に指定することで、変更する特定のフィールドを識別できます (例: <code>source.ts_ms</code>)。必要に応じて、以下の形式のエントリーをリストに追加し、フィールドの元の名前を上書きして新しい名前を割り当てることができます。</p> <p><code><field_name>:<new_field_name></code>.</p> <p>以下に例を示します。</p> <pre>version:VERSION, connector:CONNECTOR, source.ts_ms:EVENT_TIMESTAMP</pre> <p>指定する新しい名前値では、大文字と小文字が区別されます。</p> <p>SMT が簡略化したメッセージの value 要素にメタデータフィールドを追加するとき、SMT は各メタデータフィールド名の前に2つのアンダースコアを付けます。構造体の指定に関して、SMT は構造体名とフィールド名の間にもアンダースコアを挿入します。</p> <p>元の変更イベントメッセージに存在しないフィールドを指定した場合でも、SMT は指定されたフィールドを変更されたメッセージの value 要素に追加します。</p>
<code>sanitize.field.names</code>	<code>false</code>	<p>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 をご覧ください。</p>

既知の制限

- MongoDB はスキーマのないデータベースであるため、Debezium を使用してスキーマベースのデータリレーショナルデータベースに変更をストリーミングするときに一貫した列定義を確保するには、名前が同じコレクション内のフィールドに同じ型のデータを格納する必要があります。
- SMT を設定して、シンクコネクタと互換性のある形式でメッセージを生成します。シンクコネクタがフラットなメッセージ構造を必要としているにもかかわらず、ソース MongoDB ドキュメント内の配列を Structs の struct としてエンコードするメッセージを受信した場合、シンクコネクタはメッセージを処理できません。

12.7. 送信トレイパターンを使用する DEBEZIUM コネクタの設定

送信トレイパターンを使用することで、複数の (マイクロ) サービス間で安全かつ確実にデータを交換することができます。送信トレイパターンの実装により、サービスの内部状態 (通常はそのデータベースに永続化される) と同じデータを必要とするサービスで使用されるイベントの状態との間に不整合が生じるのを防ぐことができます。

Debezium アプリケーションに送信トレイパターンを実装するには、Debezium コネクタを以下のよう

- 送信トレイテーブルの変更をキャプチャーする
- Debezium 送信トレイイベントルーター単一メッセージ変換 (SMT) を適用する

送信トレイ SMT を適用するように設定された Debezium コネクタは、送信トレイテーブルで生じた変更だけをキャプチャーする必要があります。詳細は、[変換を選択的に適用するオプション](#) を参照してください。

コネクタが複数の送信トレイテーブルの変更をキャプチャーすることができるのは、それぞれの送信トレイテーブルが同じ構造を持つ場合に限りです。

送信トレイパターンが有用な理由およびその動作については、[Reliable Microservices Data Exchange With the Outbox Pattern](#) を参照してください。



注記

送信トレイイベントルーター SMT は MongoDB コネクタと互換性がありません。

MongoDB ユーザーは、[MongoDB 送信ボックスイベントルーター SMT](#) を実行できません。

詳細は以下のセクションを参照してください。

- [「Debezium 送信トレイメッセージの例」](#)
- [「Debezium 送信トレイイベントルーター SMT が要求する送信トレイテーブルの構造」](#)
- [「Debezium 送信トレイイベントルーター SMT の基本設定」](#)
- [「送信トレイイベントルーター変換を選択的に適用するオプション」](#)
- [「Debezium 送信トレイメッセージでのペイロードフォーマットとしての Avro の使用」](#)
- [「Debezium 送信トレイメッセージへの追加フィールドの出力」](#)
- [「JSON としてエスケープされた JSON 文字列の拡張」](#)
- [「送信トレイイベントルーター変換設定用のオプション」](#)

12.7.1. Debezium 送信トレイメッセージの例

Debezium 送信トレイイベントルーター SMT の設定方法を理解するには、以下の Debezium 送信トレイメッセージの例を確認してください。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
```



```
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  {"id": 1, "lineItems": [{"id": 1, "item": "Debezium in Action", "status": "ENTERED",
    "quantity": 2, "totalPrice": 39.98}, {"id": 2, "item": "Debezium for Dummies", "status":
    "ENTERED", "quantity": 1, "totalPrice": 29.99}], "orderDate": "2019-01-31T12:13:01",
    "customerId": 123}
}
```

送信トレイイベントルーター SMT を適用するように設定された Debezium コネクタは、以下のような Debezium のオリジナルメッセージを変換して上記のメッセージを生成します。

```
# Kafka Message key: "406c07f3-26f0-4eea-a50c-109940064b8f"
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484
{
  "before": null,
  "after": {
    "id": "406c07f3-26f0-4eea-a50c-109940064b8f",
    "aggregateid": "1",
    "aggregatetype": "Order",
    "payload": "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\":
    \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\",
    \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
    \"customerId\": 123}",
    "timestamp": 1556890294344,
    "type": "OrderCreated"
  },
  "source": {
    "version": "2.1.4.Final",
    "connector": "postgresql",
    "name": "dbserver1-bare",
    "db": "orderdb",
    "ts_usec": 1556890294448870,
    "txId": 584,
    "lsn": 24064704,
    "schema": "inventory",
    "table": "outboxevent",
    "snapshot": false,
    "last_snapshot_record": null,
    "xmin": null
  },
  "op": "c",
  "ts_ms": 1556890294484
}
```

この Debezium 送信トレイメッセージの例は、[デフォルトの送信トレイイベントルーター設定](#)に基づいています。ここでは、送信トレイテーブル構造および集約に基づくイベントルーティングを想定しています。動作をカスタマイズするために、送信トレイイベントルーター SMT にはさまざまな [設定オプション](#) が用意されています。

12.7.2. Debezium 送信トレイイベントルーター SMT が要求する送信トレイテーブルの構造

デフォルトの送信トレイイベントルーター SMT 設定を適用するには、送信トレイテーブルに以下の列がなければなりません。

Column	Type	Modifiers
id	uuid	not null
aggregatetype	character varying(255)	not null
aggregateid	character varying(255)	not null
type	character varying(255)	not null
payload	jsonb	

表12.6 要求される送信トレイテーブル列の説明

列	結果
id	<p>イベントの一意の ID が含まれます。送信トレイメッセージでは、この値はヘッダーです。たとえば、重複するメッセージを削除するために、この ID を使用することができます。</p> <p>イベントの一意の ID を別の outbox テーブルの列から取得するには、コネクタ設定で table.field.event.id SMT オプションを設定します。</p>
aggregatetype	<p>コネクタが送信トレイメッセージを出力するトピックの名前に SMT が追加する値が含まれます。デフォルトの動作では、この値は route.topic.replacement SMT オプションのデフォルトの #{routedByValue} 変数を置き換えます。</p> <p>たとえば、デフォルトの設定では、route.by.field SMT オプションは aggregatetype に、route.topic.replacement SMT オプションは outbox.event.#{routedByValue} に設定されます。アプリケーションが送信トレイテーブルに 2 つのレコードを追加するとします。最初のレコードでは、aggregatetype 列の値は customers です。2 つ目のレコードでは、aggregatetype 列の値は orders です。コネクタは、最初のレコードを outbox.event.customers トピックに出力します。コネクタは、2 番目のレコードを outbox.event.orders トピックに出力します。</p> <p>別の送信トレイテーブル列からこの値を取得するには、コネクタ設定で route.by.field SMT オプションを設定します。</p>
aggregateid	<p>ペイロードの ID を提供するイベントのキーが含まれます。SMT は、この値を出力される送信トレイメッセージのキーとして使用します。これは、Kafka パーティションで正しい順序を維持するのに重要です。</p> <p>イベントキーを別の送信トレイテーブルの列から取得するには、コネクタ設定で table.field.event.key SMT オプションを設定します。</p>

列	結果
payload	<p>送信トレイ変更イベントの表現。デフォルトの構造は JSON です。デフォルトでは、Kafka のメッセージ値は payload 値のみで設定されます。ただし、送信トレイイベントで追加のフィールドが含まれるように設定されている場合、Kafka メッセージ値にはペイロードと追加フィールドの両方のエンベロープエンブレットが含まれ、各フィールドは個別に表されます。詳細は、追加フィールドでのメッセージ出力 を参照してください。</p> <p>別の送信トレイテーブル列からイベントペイロードを取得するには、コネクタの構成で table.field.event.payload SMT オプションを設定します。</p>
追加のカスタム列	<p>送信トレイテーブルの列は、ペイロードセクション内またはメッセージヘッダーのいずれかとして 送信トレイイベントに追加 できます。</p> <p>一例として、イベントを分類または整理するのに役立つ、ユーザー定義の値を伝える列 eventType があります。</p>

12.7.3. Debezium 送信トレイイベントルーター SMT の基本設定

送信トレイパターンをサポートするように Debezium コネクタを設定するには、**outbox.EventRouter** SMT を設定します。SMT のデフォルトの動作を取得するには、次の例のように、オプションを指定せずにコネクタ設定に追加します。

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

設定のカスタマイズ

コネクタは、多くの種類のイベントメッセージ (ハートビートメッセージ、廃棄メッセージ、またはトランザクションまたはスキーマの変更に関するメタデータメッセージ) を発行する場合があります。outbox テーブルで発生したイベントのみに変換を適用するには、これらのイベントのみに [変換を選択的に適用する SMT 述語ステートメント](#) を定義します。

12.7.4. 送信トレイイベントルーター変換を選択的に適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。以下の方法のいずれかを使用して、SMT を選んで適用するようにコネクタを設定できます。

- [変換用の SMT 述語を設定する](#)。
- SMT の **route.topic.regex** 設定オプションを使用する。

12.7.5. Debezium 送信トレイメッセージでのペイロードフォーマットとしての Avro の使用

送信トレイイベントルーター SMT は、任意のペイロードフォーマットをサポートします。送信トレイテーブルの **payload** 列の値は、透過的に渡されます。JSON を使用する代わりに、Avro を使用するこ

ともできます。これは、メッセージフォーマットの管理や、送信トレイイベントスキーマの後方互換性を維持した進化の確保に役立ちます。

送信トレイメッセージペイロード用にソースアプリケーションがどのように Avro フォーマットのコンテンツを生成するかは、本ドキュメントの範囲外です。1つの可能性として、**Kafka Avro Serializer** クラスを利用して **Generic Record** インスタンスをシリアル化することができます。Kafka メッセージの値が正確な Avro バイナリーデータとなるようにするには、以下の設定をコネクタに適用します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.BinaryDataConverter
```

デフォルトでは、**payload** 列の値 (Avro データ) が唯一のメッセージ値となります。**BinaryDataConverter** を値のコンバーターとして設定すると、**payload** 列の値がそのまま Kafka メッセージの値に反映されます。

ハートビート、トランザクションメタデータ、またはスキーマ変更イベントを出力するように Debezium コネクタを設定することができます (サポートはコネクタによって異なります)。これらのイベントは **BinaryDataConverter** でシリアル化できないため、コンバーターがこれらのイベントのシリアル化方法を認識できるように追加の設定を指定する必要があります。例として、以下の設定では、スキーマがない状態で Apache Kafka **JsonConverter** を使用することを示しています。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.BinaryDataConverter
value.converter.delegate.converter.type=org.apache.kafka.connect.json.JsonConverter
value.converter.delegate.converter.type.schemas.enable=false
```

委譲 **Converter** 実装は **delegate.converter.type** オプションで指定します。コンバーターで追加の設定オプションが必要な場合は (例: 上記の **schemas.enable=false** を使用したスキーマの無効化)、それらを指定することもできます。



注記

コンバーター **io.debezium.converters.ByteBufferConverter** は Debezium バージョン 1.9 以降で非推奨となり、2.0 で削除されました。さらに、Kafka Connect を使用する場合、Debezium 2.x にアップグレードする前にコネクタの設定を更新する必要があります。

12.7.6. Debezium 送信トレイメッセージへの追加フィールドの出力

送信トレイテーブルに含まれる列の値を、出力される送信トレイメッセージに追加することができます。例えば、**aggregatetype** 列に **purchase-order** という値を持ち、**event Type** という列に **order-created** および **order-shipped** という値を持つ outbox テーブルを考えてみましょう。**column:placement:alias** の構文でフィールドを追加することができます。

placement に許可されている値は、**header**、**envelope**、**partition** です。

eventType 列の値を送信トレイメッセージのヘッダーに出力するには、以下のような SMT を設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:header:type
```

結果は、**型** がキーとして Kafka メッセージのヘッダー、および **eventType** 列の値はその値になります。

eventType 列の値を送信トレイメッセージのエンベロープに出力するには、以下のような SMT を設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=eventType:envelope:type
```

送信メッセージをどのパーティションで生成するかを制御するには、SMT を次のように設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.table.fields.additional.placement=partitionColumn:partition
```

なお、**partition** 配置については、エイリアスを追加しても効果はありません。

12.7.7. JSON としてエスケープされた JSON 文字列の拡張

Debezium 送信トレイメッセージに String として表現された **payload** が含まれていることに気付いたかもしれません。そのため、この文字列が実際の JSON の場合、以下のように Kafka メッセージでエスケープされて表示されます。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": 1, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\",
  \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\":
  \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\",
  \"customerId\": 123}"
}
```

送信トレイイベントルーターを使用すると、JSON ドキュメント自体から推測されるコンパニオンスキーマを使用して、このメッセージコンテンツを実際 JSON に展開できます。これにより、Kafka メッセージは以下ようになります。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=4d47e190-0402-4048-bc2c-89dd54343cdc"
# Kafka Message Timestamp: 1556890294484
{
  "id": 1, "lineItems": [{"id": 1, "item": "Debezium in Action", "status": "ENTERED", "quantity": 2,
  "totalPrice": 39.98}, {"id": 2, "item": "Debezium for Dummies", "status": "ENTERED", "quantity": 1,
  "totalPrice": 29.99}], "orderDate": "2019-01-31T12:13:01", "customerId": 123
}
```

この変換を有効にするには、**table.expand.json.payload** を true に設定し、以下のように **JsonConverter** を使用する必要があります。

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

```
transforms.outbox.table.expand.json.payload=true
value.converter=org.apache.kafka.connect.json.JsonConverter
```

12.7.8. 送信トレイイベントルーター変換設定用のオプション

次の表で、送信トレイイベントルーター SMT に指定することのできるオプションを説明します。表の **グループ** 列は、Kafka の設定オプションクラスを示しています。

表12.7 送信トレイイベントルーター SMT 設定オプションの説明

オプション	デフォルト	グループ	説明
table.op.invalid.behavior	warn	テーブル	<p>送信トレイテーブルに UPDATE 操作がある場合の SMT の動作を決定します。設定可能な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● warn: SMT はログに警告を記録し、次の送信トレイテーブルレコードに進みます。 ● error: SMT はログにエラーを記録し、次の送信トレイテーブルレコードに進みます。 ● fatal: SMT はログにエラーを記録し、コネクタは処理を停止します。 <p>送信トレイテーブルのすべての変更は、INSERT 操作であると想定されます。つまり、送信トレイテーブルはキューとして機能し、送信トレイテーブルのレコードに対する更新は許可されません。SMT は、送信トレイテーブルの DELETE 操作を自動的に除外します。</p>
table.field.event.id	id	テーブル	一意のイベント ID が含まれる送信トレイテーブル列を指定します。この ID は、出力されるイベントのヘッダーの id キーに保存されます。
table.field.event.key	aggregateid	テーブル	イベントキーが含まれる送信トレイテーブル列を指定します。この列に値が含まれる場合、SMT はその値を出力される送信トレイメッセージのキーとして使用します。これは、Kafka パーティションで正しい順序を維持するのに重要です。

オプション	デフォルト	グループ	説明
<code>table.field.event.timestamp</code>		テーブル	デフォルトでは、出力される送信トレイメッセージのタイムスタンプは、Debezium イベントのタイムスタンプです。送信トレイメッセージで別のタイムスタンプを使用するには、このオプションを出力される送信トレイメッセージに使用するタイムスタンプが含まれる送信トレイテーブル列に設定します。
<code>table.field.event.payload</code>	<code>payload</code>	テーブル	イベントペイロードが含まれる送信トレイテーブル列を指定します。
<code>table.expand.json.payload</code>	<code>false</code>	テーブル	String ペイロードの JSON 拡張を実行するかどうかを指定します。コンテンツが見つからなかった場合や、解析エラーの場合は、コンテンツはそのままとして保持されます。 詳細は、 escaped json の拡張 セクションを参照してください。
<code>table.json.payload.null.behavior</code>	<code>ignore</code>	テーブル	JSON 拡張プロパティ <code>table.expand.json.payload</code> を有効にすると、送信ボックステーブルに <code>null</code> 値を含む json ペイロードの動作が決定されます。設定可能な値は以下のとおりです。 <ul style="list-style-type: none"> ● <code>ignore</code>: null 値を無視します。 ● <code>optional_bytes</code> - null 値を保持し、null を接続のオプションのバイトとして扱います。

オプション	デフォルト	グループ	説明
table.fields.additional.placement		テーブル、エンベロープ	<p>送信トレイメッセージのヘッダーまたはエンベロープに追加する1つまたは複数の送信トレイテーブル列を指定します。ペアのコンマ区切りリストを指定します。それぞれのペアで、列の名前および値をヘッダーとエンベロープのどちらに含めるかを指定します。ペア内の値はコロンで区切ります。以下に例を示します。</p> <p>id:header,my-field:envelope</p> <p>列のエイリアスを指定するには、3番目の値としてエイリアスが含まれるトリオを指定します。以下に例を示します。</p> <p>id:header,my-field:envelope:my-alias</p> <p>2番目の値は配置で、常に header または envelope でなければなりません。</p> <p>設定例は、Debezium 送信トレイメッセージへの追加フィールドの出力 に記載されています。</p>
table.field.event.schema.version		テーブル、スキーマ	<p>このオプションを設定すると、Kafka Connect スキーマ Javadoc で説明されているように、その値がスキーマバージョンとして使用されます。</p>
route.by.field	aggregatetype	ルーター	<p>送信トレイテーブルの列の名前を指定します。デフォルトの動作では、この列の値が、コネクタが送信トレイメッセージを出力するトピックの名前の一部になります。例を 要求される送信トレイテーブルの説明 に示します。</p>
route.topic.regex	(? < routedByValue >.*)	ルーター	<p>送信トレイ SMT が <code>RegexRouter</code> で送信トレイテーブルレコードに適用する正規表現を指定します。この正規表現は、route.topic.replacement SMT オプションの設定の一部です。</p> <p>デフォルトの動作では、SMT は route.topic.replacement SMT オプションの設定でデフォルトの `\${routedByValue} 変数を route.by.field 送信トレイ SMT オプションの設定に置き換えることです。</p>

オプション	デフォルト	グループ	説明
route.topic.replacement	<code>outbox.event .\${routedByValue}</code>	ルーター	<p>コネクターが送信トレイメッセージを出力するトピックの名前を指定します。デフォルトのトピック名では、outbox.event.の後に送信トレイテーブルレコードの aggregatetype 列の値が続きます。たとえば、aggregatetype の値が customers の場合には、トピック名は outbox.event.customers になります。</p> <p>トピック名を変更するには、次の操作を行います。</p> <ul style="list-style-type: none"> ● route.by.field オプションを別の列に設定します。 ● route.topic.regex オプションを別の正規表現に設定する。
route.tombstone.on.empty.payload	<code>false</code>	ルーター	空または null のペイロードによってコネクターがトゥームストーンイベントを出力するかどうかを示します。

12.8. 送信トレイパターンを使用する DEBEZIUM MONGODB コネクターの設定



注記

この SMT は、DebeziumMongoDB コネクターでのみ使用されます。リレーショナルデータベースに送信ボックスイベントルーター SMT を使用する方法は、[送信ボックスイベントルーター](#) を参照してください。

送信トレイパターンを使用することで、複数の (マイクロ) サービス間で安全かつ確実にデータを交換することができます。送信トレイパターンの実装により、サービスの内部状態 (通常はそのデータベースに永続化される) と同じデータを必要とするサービスで使用されるイベントの状態との間に不整合が生じるのを防ぐことができます。

Debezium アプリケーションに送信トレイパターンを実装するには、Debezium コネクターを以下のよう設定します。

- 送信トレイコレクションの変更をキャプチャーする
- Debezium MongoDB 送信トレイイベントルーター単一メッセージ変換 (SMT) を適用する

MongoDB 送信トレイ SMT を適用するように設定された Debezium コネクターは、送信トレイコレクションで生じた変更だけをキャプチャーする必要があります。詳細は、[変換を選択的に適用するオプション](#) を参照してください。

コネクタが複数の送信トレイコレクションの変更をキャプチャーすることができるのは、それぞれの送信トレイコレクションが同じ構造を持つ場合に限りです。



注記

この SMT を使用するには、実際のビジネスコレクションの操作と送信トレイコレクションへの挿入を、ビジネスコレクションおよび送信トレイコレクション間にデータの不整合が発生するのを回避するために、MongoDB 4.0 以降でサポートされているマルチドキュメントトランザクションの一部として実行する必要があります。将来の更新では、既存のデータを更新し、マルチドキュメントトランザクションなしで ACID トランザクションに送信トレイイベントを挿入できるようにするために、送信トレイイベントを、独立した送信トレイコレクションとしてではなく、既存コレクションのサブドキュメントとして保存するための追加の設定をサポートする予定です。

送信トレイパターンの詳細については、[Reliable Microservices Data Exchange With the Outbox Pattern](#) を参照してください。

詳細は以下のセクションを参照してください。

- [「Debezium MongoDB 送信トレイメッセージの例」](#)
- [「Debezium mongodb 送信トレイイベントルーター SMT が要求する送信トレイコレクションの構造」](#)
- [「基本的な Debezium MongoDB 送信トレイイベントルーター SMT 設定」](#)
- [「Debezium MongoDB 送信トレイメッセージでペイロードフォーマットとして Avro を使用」](#)
- [「Debezium MongoDB 送信トレイメッセージへの追加フィールドの出力」](#)
- [「送信トレイイベントルーター変換設定用のオプション」](#)

12.8.1. Debezium MongoDB 送信トレイメッセージの例

Debezium MongoDB 送信トレイイベントルーター SMT の設定方法を理解するには、以下の Debezium 送信トレイメッセージの例を検討してください。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "b2730779e1f596e275826f08"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": \"{\\$oid\\\": \\\"da8d6de63b7745ff8f4457db\\\"}\", \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in Action\", \"status\": \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium for Dummies\", \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-31T12:13:01\", \"customerId\": 123}"
}
```

MongoDB 送信トレイイベントルーター SMT を適用するように設定された Debezium コネクタは、次の例で示すとおり、raw Debezium 変更イベントメッセージを変換して上記のメッセージを生成します。

```
# Kafka Message key: { "id": "{\\$oid\\\": \"596e275826f08b2730779e1f\"}" }
# Kafka Message Headers: ""
# Kafka Message Timestamp: 1556890294484
```

```

{
  "patch": null,
  "after": {"_id": {"$oid": "\596e275826f08b2730779e1f"}, "aggregateid": {"$oid":
  "\b2730779e1f596e275826f08"}, "aggreatetype": "\Order", "type": "\OrderCreated", "payload":
  {"_id": {"$oid": "\da8d6de63b7745ff8f4457db"}, "lineltems": [{"id": 1, "item": "\Debezium in
  Action", "status": "\ENTERED", "quantity": 2, "totalPrice": 39.98}, {"id": 2, "item": "\Debezium
  for Dummies", "status": "\ENTERED", "quantity": 1, "totalPrice": 29.99}], "orderDate": "\2019-01-
  31T12:13:01", "customerid": 123}},
  "source": {
    "version": "2.1.4.Final",
    "connector": "mongodb",
    "name": "fulfillment",
    "ts_ms": 1558965508000,
    "snapshot": false,
    "db": "inventory",
    "rs": "rs0",
    "collection": "customers",
    "ord": 31,
    "h": 1546547425148721999
  },
  "op": "c",
  "ts_ms": 1556890294484
}

```

この Debezium 送信トレイメッセージの例は、[デフォルトの送信トレイイベントルーター設定](#)に基づいています。ここでは、送信トレイコレクション構造および集約に基づくイベントルーティングを想定しています。動作をカスタマイズするために、送信トレイイベントルーター SMT にはさまざまな [設定オプション](#) が用意されています。

12.8.2. Debezium mongodb 送信トレイイベントルーター SMT が要求する送信トレイコレクションの構造

デフォルトの MongoDB 送信トレイイベントルーター SMT 設定を適用するには、送信トレイコレクションに以下のフィールドがなければなりません。

```

{
  "_id": "objectId",
  "aggreatetype": "string",
  "aggregateid": "objectId",
  "type": "string",
  "payload": "object"
}

```

表12.8 要求される送信トレイコレクションフィールドの説明

フィールド	結果
id	<p>イベントの一意の ID が含まれます。送信トレイメッセージでは、この値はヘッダーです。たとえば、重複するメッセージを削除するために、この ID を使用することができます。</p> <p>イベントの一意の ID を別の送信トレイコレクションフィールドから取得するには、コネクタ設定で collection.field.event.id SMT オプションを設定します。</p>

フィールド	結果
aggregatetype	<p>コネクタが送信トレイメッセージを出力するトピックの名前に SMT が追加する値が含まれます。デフォルトの動作では、この値は route.topic.replacement SMT オプションのデフォルトの #{routedByValue} 変数を置き換えます。</p> <p>たとえば、デフォルトの設定では、route.by.field SMT オプションは aggregatetype に、route.topic.replacement SMT オプションは outbox.event.#{routedByValue} に設定されます。アプリケーションが 2 つのドキュメントを送信トレイコレクションに追加するとします。最初のドキュメントでは、aggregatetype フィールドの値は customers です。2 番目のドキュメントでは、aggregatetype フィールドの値は orders です。コネクタは、最初のドキュメントを outbox.event.customers トピックに送信します。コネクタは、2 番目のドキュメントを outbox.event.orders トピックに送信します。</p> <p>別の送信トレイコレクションフィールドからこの値を取得するには、コネクタ設定で route.by.field SMT オプションを設定します。</p>
aggregateid	<p>ペイロードの ID を提供するイベントのキーが含まれます。SMT は、この値を出力される送信トレイメッセージのキーとして使用します。これは、Kafka パーティションで正しい順序を維持するのに重要です。</p> <p>別の送信トレイコレクションフィールドからイベントキーを取得するには、コネクタ設定で collection.field.event.key SMT オプションを設定します。</p>
payload	<p>送信トレイ変更イベントの表現。デフォルトの構造は JSON です。デフォルトでは、Kafka のメッセージ値は payload 値のみで設定されます。ただし、送信トレイイベントで追加のフィールドが含まれるように設定されている場合、Kafka メッセージ値にはペイロードと追加フィールドの両方のエンベロープエンブレットが含まれ、各フィールドは個別に表されます。詳細は、追加フィールドでのメッセージ出力 を参照してください。</p> <p>別の送信トレイコレクションフィールドからイベントペイロードを取得するには、コネクタ設定で collection.field.event.payload SMT オプションを設定します。</p>
追加のカスタムフィールド	<p>送信トレイコレクションの追加フィールドは、ペイロードセクション内に、またはメッセージヘッダーとして、送信トレイイベントに追加 できます。</p> <p>一例として、イベントを分類または整理するのに役立つ、ユーザー定義の値を伝えるフィールド eventType などがあります。</p>

12.8.3. 基本的な Debezium MongoDB 送信トレイイベントルーター SMT 設定

送信トレイパターンをサポートするように Debezium MongoDB コネクタを設定するには、**outbox.MongoEventRouter** SMT を設定します。SMT のデフォルトの動作を取得するには、次の例のように、オプションを指定せずにコネクタ設定に追加します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
```

設定のカスタマイズ

コネクタは、多くの種類のイベントメッセージ (ハートビートメッセージ、廃棄メッセージ、またはトランザクションの変更に関するメタデータメッセージ) を発行する場合があります。送信トレイコレクションで発生したイベントだけに変換を適用するには、これらのイベントのみを対象に [変換を選択して適用する SMT 述語ステートメント](#) を定義します。

12.8.4. MongoDB 送信トレイイベントルーター変換を選択的に適用するオプション

データベースの変更が発生したときに Debezium コネクタが出力する変更イベントメッセージの他に、コネクタはハートビートメッセージなど、他のタイプのメッセージとスキーマ変更およびトランザクションに関するメタデータメッセージも出力します。これらの他のメッセージの構造は、SMT が処理するように設計された変更イベントメッセージの構造とは異なるため、目的のデータ変更メッセージのみを処理するようにコネクタを SMT を選んで適用することが推奨されます。以下の方法のいずれかを使用して、SMT を選んで適用するようにコネクタを設定できます。

- [変換用の SMT 述語を設定する](#)。
- SMT の `route.topic.regex` 設定オプションを使用する。

12.8.5. Debezium MongoDB 送信トレイメッセージでペイロードフォーマットとして Avro を使用

MongoDB 送信ボックスイベントルーター SMT は、任意のペイロード形式をサポートします。送信トレイコレクションの `payload` フィールド値は透過的に渡されます。JSON を使用する代わりに、Avro を使用することもできます。これは、メッセージフォーマットの管理や、送信トレイイベントスキーマの後方互換性を維持した進化の確保に役立ちます。

送信トレイメッセージペイロード用にソースアプリケーションがどのように Avro フォーマットのコンテンツを生成するかは、本ドキュメントの範囲外です。1つの可能性として、**Kafka Avro Serializer** クラスを利用して **Generic Record** インスタンスをシリアライズすることができます。Kafka メッセージの値が正確な Avro バイナリーデータとなるようにするには、以下の設定をコネクタに適用します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
value.converter=io.debezium.converters.ByteArrayConverter
```

デフォルトでは、`payload` フィールドの値 (Avro データ) が唯一のメッセージ値となります。値変換器として **ByteArrayConverter** を設定すると、`payload` フィールドの値がそのまま Kafka メッセージの値に伝搬されます。

これは、他の SMT に推奨される **BinaryDataConverter** とは異なることに注意してください。これは、MongoDB が内部でバイト配列を保存する際に行っているアプローチが異なるためです。

ハートビート、トランザクションメタデータ、またはスキーマ変更イベントを出力するように Debezium コネクタを設定することができます (サポートはコネクタによって異なります)。これらのイベントは **ByteArrayConverter** でシリアライズできないため、コンバーターがこれらのイベントのシリアライズ方法を認識できるように追加の設定を指定する必要があります。例として、以下の設定では、スキーマがない状態で Apache Kafka **JsonConverter** を使用することを示しています。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
```

```
value.converter=io.debezium.converters.ByteArrayConverter
value.converter.delegate.converter.type=org.apache.kafka.connect.json.JsonConverter
value.converter.delegate.converter.type.schemas.enable=false
```

委譲 **Converter** 実装は **delegate.converter.type** オプションで指定します。コンバーターで追加の設定オプションが必要な場合は (例: 上記の **schemas.enable=false** を使用したスキーマの無効化)、それらを指定することもできます。

12.8.6. Debezium MongoDB 送信トレイメッセージへの追加フィールドの出力

送信トレイコレクションに含まれるフィールドの値を、出力される送信トレイメッセージに追加することができます。例えば、**aggregatetype** フィールドに **purchase-order** という値を持ち、**event Type** というフィールドに **order-created** および **order-shipped** という値を持つ送信トレイコレクションを考えてみましょう。**field:placement:alias** の構文でフィールドを追加できる。

placement に許可されている値は、**header**、**envelope**、**partition** です。

eventType フィールドの値を送信トレイメッセージのヘッダーに出力するには、以下のような SMT を設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.collection.fields.additional.placement=eventType:header:type
```

結果は、**type** がキーとして Kafka メッセージのヘッダー、および **eventType** 列の値はその値になります。

eventType フィールドの値を送信トレイメッセージのエンベロープに出力するには、以下のような SMT を設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.collection.fields.additional.placement=eventType:envelope:type
```

送信メッセージをどのパーティションで生成するかを制御するには、SMT を次のように設定します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
transforms.outbox.collection.fields.additional.placement=partitionField:partition
```

なお、**partition** 配置については、エイリアスを追加しても効果はありません。

12.8.7. JSON としてエスケープされた JSON 文字列の拡張

デフォルトでは、Debezium 送信トレイメッセージの **payload** は文字列として表されます。文字列の元のソースが JSON 形式の場合、次の例に示すように、結果の Kafka メッセージはエスケープシーケンスを使用して文字列を表します。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "{\"id\": {\"$oid\": \"da8d6de63b7745ff8f4457db\"}}, \"lineItems\": [{\"id\": 1, \"item\": \"Debezium in
```

```
Action\", \"status\": \"ENTERED\", \"quantity\": 2, \"totalPrice\": 39.98}, {\"id\": 2, \"item\": \"Debezium
for Dummies\", \"status\": \"ENTERED\", \"quantity\": 1, \"totalPrice\": 29.99}], \"orderDate\": \"2019-01-
31T12:13:01\", \"customerId\": 123}
}
```

メッセージの内容を展開し、エスケープされた JSON を元のエスケープされていない JSON 形式に変換するように、送信トレイイベントルーターを設定できます。変換された文字列では、コンパニオンスキーマは元の JSON ドキュメントから推定されます。次の例は、結果の Kafka メッセージで展開された JSON を示しています。

```
# Kafka Topic: outbox.event.order
# Kafka Message key: "1"
# Kafka Message Headers: "id=596e275826f08b2730779e1f"
# Kafka Message Timestamp: 1556890294484
{
  "id": "da8d6de63b7745ff8f4457db", "lineItems": [{"id": 1, "item": "Debezium in Action", "status":
"ENTERED", "quantity": 2, "totalPrice": 39.98}, {"id": 2, "item": "Debezium for Dummies", "status":
"ENTERED", "quantity": 1, "totalPrice": 29.99}], "orderDate": "2019-01-31T12:13:01", "customerId":
123
}
```

変換時の文字列変換を有効にするには、**collection.expand.json.payload** の値を **true** に設定し、次の例に示すように **StringConverter** を使用します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
transforms.outbox.collection.expand.json.payload=true
value.converter=org.apache.kafka.connect.storage.StringConverter
```

12.8.8. 送信トレイイベントルーター変換設定用のオプション

次の表で、送信トレイイベントルーター SMT に指定することのできるオプションを説明します。表の **グループ** 列は、Kafka の設定オプションクラスを示しています。

表12.9 送信トレイイベントルーター SMT 設定オプションの説明

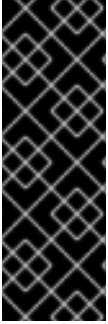
オプション	デフォルト	グループ	説明
		プ	

オプション	デフォルト	グループ	説明
<code>collection.op.invalid.behavior</code>	<code>warn</code>	コレクション	<p>送信トレイコレクションも更新操作がある場合の SMT の動作を決定します。設定可能な値は以下のとおりです。</p> <ul style="list-style-type: none"> ● warn: SMT はログに警告を記録し、次の送信トレイコレクションドキュメントに進みます。 ● error: SMT はログにエラーを記録し、次の送信トレイコレクションドキュメントに進みます。 ● fatal: SMT はログにエラーを記録し、コネクタは処理を停止します。 <p>送信トレイコレクションのすべての変更は、挿入または削除操作であると想定されます。つまり、送信トレイコレクションはキューとして機能します。送信トレイコレクション内のドキュメントの更新は許可されていません。SMT は、送信トレイコレクションの削除操作を自動的に除外します (進行中の送信トレイイベントが削除されるため)。</p>
<code>collection.field.event.id</code>	<code>_id</code>	コレクション	一意のイベント ID が含まれる送信トレイコレクションフィールドを指定します。この ID は、出力されるイベントのヘッダーの <code>id</code> キーに保存されます。
<code>collection.field.event.key</code>	<code>aggregateid</code>	コレクション	イベントキーが含まれる送信トレイコレクションフィールドを指定します。このフィールドに値が含まれる場合、SMT はその値を出力される送信トレイメッセージのキーとして使用します。これは、Kafka パーティションで正しい順序を維持するのに重要です。
<code>collection.field.event.timestamp</code>		コレクション	デフォルトでは、出力される送信トレイメッセージのタイムスタンプは、Debezium イベントのタイムスタンプです。送信トレイメッセージで別のタイムスタンプを使用するには、このオプションを出力される送信トレイメッセージに使用するタイムスタンプが含まれる送信トレイコレクションフィールドに設定します。

オプション	デフォルト	グループ	説明
<code>collection.field.event.payload</code>	<code>payload</code>	コレクション	イベントペイロードが含まれる送信トレイコレクションフィールドを指定します。
<code>collection.expand.json.payload</code>	<code>false</code>	コレクション	String ペイロードの JSON 拡張を実行するかどうかを指定します。コンテンツが見つからなかった場合や、解析エラーの場合は、コンテンツはそのままとして保持されます。 詳細は、 escaped json の拡張 セクションを参照してください。
<code>collection.fields.additional.placement</code>		コレクション、エンベロープ	送信トレイメッセージのヘッダーまたはエンベロープに追加する1つまたは複数の送信トレイコレクションフィールドを指定します。ペアのコンマ区切りリストを指定します。それぞれのペアで、フィールドの名前および値をヘッダーとエンベロープのどちらに含めるかを指定します。ペア内の値はコロンで区切ります。以下に例を示します。 id:header,my-field:envelope フィールドのエイリアスを指定するには、3番目の値としてエイリアスが含まれるトリオを指定します。以下にフィールドを示します。 id:header,my-field:envelope:my-alias 2番目の値は配置で、常に header または envelope でなければなりません。 設定例は、 Debezium 送信トレイメッセージへの追加フィールドの出力 に記載されています。
<code>collection.field.event.schema.version</code>		コレクション、スキーマ	このオプションを設定すると、 Kafka Connect スキーマ Javadoc で説明されているように、その値がスキーマバージョンとして使用されます。

オプション	デフォルト	グループ	説明
route.by.field	<code>aggregatetype</code>	ルーター	送信トレイコレクションのフィールドの名前を指定します。デフォルトでは、このフィールドで指定された値が、コネクターが送信トレイメッセージを出力するトピックの名前の一部になります。例を 要求される送信トレイコレクションの説明 に示します。
route.topic.regex	<code>(? <routedByValue >.*)</code>	ルーター	送信トレイ SMT が <code>RegexRouter</code> で送信トレイコレクションドキュメントに適用する正規表現を指定します。この正規表現は、 route.topic.replacement SMT オプションの設定の一部です。 + デフォルトの動作では、SMT は route.topic.replacement SMT オプションの設定のデフォルト <code>\${routedByValue}</code> 変数を route.by.field 送信トレイ SMT オプションの設定に置き換えることです。
route.topic.replacement	<code>outbox.event .\${routedByValue}</code>	ルーター	コネクターが送信トレイメッセージを出力するトピックの名前を指定します。デフォルトのトピック名では、 <code>outbox.event.</code> の後に送信トレイコレクションドキュメントの <code>aggregatetype</code> フィールドの値が続きます。たとえば、 <code>aggregatetype</code> の値が <code>顧客</code> の場合には、トピック名は <code>outbox.event.customers</code> になります。 + トピック名を変更するには、次の操作を行います。 <ul style="list-style-type: none"> ● route.by.field オプションを別のフィールドに設定します。 ● route.topic.regex オプションを別の正規表現に設定します。
route.tombstone.on.empty.payload	<code>false</code>	ルーター	空または <code>null</code> のペイロードによってコネクターがトゥームストーンイベントを出力するかどうかを示します。

第13章 DEBEZIUM カスタムデータ型コンバーターの開発



重要

カスタム開発コンバーターの使用は、テクノロジープレビュー機能のみです。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行い、フィードバックを提供していただくことを目的としています。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、[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 変更イベントレコードの各フィールドは、ソーステーブルまたはデータコレクションのフィールドまたは列を表します。コネクターが変更イベントレコードを Kafka に送信すると、ソースの各フィールドのデータ型が Kafka Connect スキーマ型に変換されます。列の値も同様に、変換先フィールドのスキーマタイプに一致するように変換されます。コネクターごとに、コネクターが各データ型を変換する方法をデフォルトのマッピングで指定します。これらのデフォルトマッピングは、各コネクターのデータ型のドキュメントで説明されています。

通常、デフォルトのマッピングで十分ですが、一部のアプリケーションでは別のマッピングを適用する場合があります。たとえば、デフォルトのマッピングが UNIX エポック以降のミリ秒の形式を使用して列をエクスポートする場合は、カスタムマッピングが必要になることがあります。ダウンストリームアプリケーションは、フォーマットされた文字列としてしか、列の値を消費できません。カスタムコンバーターを開発およびデプロイして、データ型マッピングをカスタマイズします。特定のタイプのすべての列に作用するようにカスタムコンバーターを設定するか、特定のテーブル列のみに適用されるように範囲を狭めてください。コンバーター関数は、指定された基準に一致する列のデータ型変換リクエストをインターセプトし、指定の変換を実行します。コンバーターは、指定された基準に一致しない列を無視します。

カスタムコンバーターは、Debezium サービスプロバイダーインターフェイス (SPI) を実装する Java クラスです。コネクター設定で **converters** プロパティを設定して、カスタムコンバーターを有効化および設定します。**converters** プロパティは、コネクターで使用できるコンバータを指定し、変換動作をさらに変更するサブプロパティを含めることができます。

コネクターを開始すると、コネクター設定で有効になっているコンバーターがインスタンス化され、レジストリーに追加されます。レジストリーは、各コンバーターと、そのコンバーターの列またはフィールドを関連付けて処理します。Debezium が新しい変更イベントを処理するときにはいつでも、設定されたコンバーターを呼び出して、登録されている列またはフィールドを変換します。

13.1. DEBEZIUM カスタムデータ型コンバーターの作成

以下の例は、インターフェイス **io.debezium.spi.converter.CustomConverter** を実装する Java クラスのコンバーター実装を示しています。

```
public interface CustomConverter<S, F extends ConvertedField> {

    @FunctionalInterface
    interface Converter { 1
        Object convert(Object input);
    }

    public interface ConverterRegistration<S> { 2
        void register(S fieldSchema, Converter converter); 3
    }
}
```

```

void configure(Properties props);

void converterFor(F field, ConverterRegistration<S> registration); ❷
}

```

- ❶ データをある型から別の型に変換する関数。
- ❷ コンバーターを登録するためのコールバック。
- ❸ 現在のフィールドの指定のスキーマとコンバーターを登録します。同じフィールドに対して複数回呼び出すことはできません。
- ❹ 特定のフィールドで使用するカスタマイズされた値とスキーマコンバーターを登録します。

カスタムコンバーターメソッド

CustomConverter インターフェイスの実装には、以下のメソッドが含まれている必要があります。

configure()

コネクタ設定に指定されたプロパティをコンバーターインスタンスに渡します。**configure** メソッドは、コネクタが初期化されると実行されます。複数のコネクタでコンバーターを使用し、コネクタのプロパティ設定に基づいてその動作を変更できます。

configure メソッドは以下の引数を受け入れます。

props

コンバーターインスタンスに渡すプロパティが含まれています。各プロパティは、特定のタイプの列値を変換する形式を指定します。

converterFor()

コンバーターを登録して、データソースの特定の列またはフィールドを処理します。Debezium は、**converterFor()** メソッドを呼び出して、コンバーターに変換の **registration** を呼び出すように促します。**converterFor** メソッドは、各列に対して1回実行されます。

メソッドは以下の引数を受け入れます。

field

処理されるフィールドまたは列に関するメタデータを渡すオブジェクト。列のメタデータには、列またはフィールドの名前、テーブルまたはコレクションの名前、データ型、サイズなどを含めることができます。

登録

対象のスキーマ定義と列データを変換するコードを提供する

io.debezium.spi.converter.CustomConverter.ConverterRegistration 型のオブジェクト。ソース列が、コンバーターが処理する型と一致する場合に、コンバーターは **registration** パラメーターを呼び出します。**register** メソッドを呼び出して、スキーマ内の各列のコンバーターを定義します。スキーマは、Kafka Connect **SchemaBuilder** API を使用して表されます。

13.1.1. Debezium カスタムコンバーターの例

以下の例は、以下の操作を実行する単純なコンバーターを実装します。

- コネクタ設定で指定された **schema.name** プロパティの値に基づいてコンバーターを設定する **configure** メソッドを実行します。コンバーターの設定は、各インスタンスに固有です。

- データ型が **isbn** に設定されているソース列の値を処理するコンバーターを登録する **converterFor** メソッドを実行します。
 - **schema.name** プロパティに指定された値に基づいてターゲット **STRING** スキーマを識別します。
 - ソース列の ISBN データを **String** 値に変換します。

例13.1 簡単なカスタムコンバーター

```
public static class IsbnConverter implements CustomConverter<SchemaBuilder,
RelationalColumn> {

    private SchemaBuilder isbnSchema;

    @Override
    public void configure(Properties props) {
        isbnSchema = SchemaBuilder.string().name(props.getProperty("schema.name"));
    }

    @Override
    public void converterFor(RelationalColumn column,
        ConverterRegistration<SchemaBuilder> registration) {

        if ("isbn".equals(column.typeName())) {
            registration.register(isbnSchema, x -> x.toString());
        }
    }
}
```

13.1.2. Debezium および Kafka Connect API モジュールの依存関係

カスタムコンバーター Java プロジェクトには、Debezium API および Kafka Connect API ライブラリーモジュールに対するコンパイル依存関係があります。以下の例のように、これらのコンパイル依存関係は、プロジェクトの **pom.xml** に含める必要があります。

```
<dependency>
  <groupId>io.debezium</groupId>
  <artifactId>debezium-api</artifactId>
  <version>${version.debezium}</version> ❶
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>connect-api</artifactId>
  <version>${version.kafka}</version> ❷
</dependency>
```

❶ **\${version.debezium}** は Debezium コネクターのバージョンを表します。

❷ **\${version.kafka}** は、環境内の Apache Kafka のバージョンを表します。

13.2. DEBEZIUM コネクターでのカスタムコンバーターの使用

カスタムコンバーターは、ソーステーブルの特定の列または列の型に作用して、ソースのデータ型を Kafka Connect スキーマ型に変換する方法を指定します。コネクタでカスタムコンバーターを使用するには、コンバーター JAR ファイルをコネクタファイルとともにデプロイしてから、コンバーターを使用するようにコネクタを設定します。

13.2.1. カスタムコンバーターのデプロイ

前提条件

- カスタムコンバーター Java プログラムがある。

手順

- Debezium コネクタでカスタムコンバーターを使用するには、Java プロジェクトを JAR ファイルにエクスポートし、そのファイルを使用する各 Debezium コネクタの JAR ファイルを含むディレクトリにコピーします。

たとえば、一般的なデプロイメントでは、Debezium コネクタファイルは Kafka Connect ディレクトリーのサブディレクトリー (`/kafka/connect`) に保存され、各コネクタ JAR は独自のサブディレクトリー (`/kafka/connect/debezium-connector-db2`、`/kafka/connect/debezium-connector-mysql` など) に格納されます。コネクタでコンバーターを使用するには、コンバーターの JAR ファイルをコネクタのサブディレクトリーに追加します。



注記

複数のコネクタでコンバーターを使用するには、各コネクタのサブディレクトリーにコンバーター JAR ファイルのコピーを配置する必要があります。

13.2.2. カスタムコンバーターを使用するためのコネクタの設定

コネクタがカスタムコンバーターを使用できるようにするには、コンバーターの名前とクラスを指定するプロパティをコネクタ設定に追加します。コンバーターが特定のデータ型の形式をカスタマイズするためにさらに情報を必要とする場合は、他の設定オプションを定義して、その情報を指定することもできます。

手順

- 以下の必須プロパティをコネクタ設定に追加して、コネクタインスタンスのコンバーターを有効にします。

```
converters: <converterSymbolicName> 1
<converterSymbolicName>.type: <fullyQualifiedConverterClassName> 2
```

1 **converters** プロパティは必須であり、コネクタで使用するコンバーターインスタンスのシンボリック名のコンマ区切りリストを列挙します。このプロパティに一覧表示される値は、コンバーターに指定する他のプロパティの値の名前の接頭辞として機能します。

2 **<converterSymbolicName>.type** プロパティは必須であり、コンバーターを実装するクラスの名前を指定します。たとえば、前述の [カスタムコンバーターのサンプル](#) では、以下のプロパティをコネクタ設定に追加します。

```
converters: isbn  
isbn.type: io.debezium.test.IsbnConverter
```

- 他のプロパティをカスタムコンバーターに関連付けるには、プロパティ名の前にコンバーターの記号名を付け、その後にドット (.) を付けます。シンボリック名は、**converters** プロパティの値として指定するラベルです。たとえば、前述の **isbn** コンバーターのプロパティを追加して、コンバーターコードの **configure** メソッドに渡す **schema.name** を指定するには、以下のプロパティを追加します。

```
isbn.schema.name: io.debezium.postgresql.type.Isbn
```

改訂日時: 2023-05-04