Red Hat Integration 2022.Q1

Debezium ユーザーガイド

Debezium 1.7 での使用

Last Updated: 2022-10-25

Red Hat Integration 2022.Q1 Debezium ユーザーガイド

Debezium 1.7 での使用

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

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Debezium_User_Guide.ent file |.

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、Debezium で提供されるコネクターを使用する方法を説明します。

目次

前書き **8**
多様性を受け入れるオープンソースの強化 8

第1章 DEBEZIUM の概要 **9**
1.1. DEBEZIUM の機能 9
1.2. DEBEZIUM アーキテクチャーの説明 10

第2章 必要となるカスタムリソースのアップグレード **11**

第3章 DB2 の DEBEZIUM コネクター **12**
3.1. DEBEZIUM DB2 コネクターの概要 12
3.2. DEBEZIUM DB2 コネクターの仕組み 13
3.2.1. Debezium Db2 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 14
3.2.1.1. アドホックスナップショット 14
3.2.1.2. 増分スナップショット 16
3.2.2. Debezium Db2 コネクターによる変更データテーブルの読み取り方法 20
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 20
3.2.4. Debezium Db2 コネクターのスキーマ変更トピック 21
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 25
3.3. DEBEZIUM DB2 コネクターのデータ変更イベントの説明 26
3.3.1. Debezium db2 変更イベントのキー 28
3.3.2. Debezium Db2 変更イベントの値 30
3.4. DEBEZIUM DB2 コネクターによるデータ型のマッピング方法 39
3.5. DEBEZIUM コネクターを実行するための DB2 の設定 44
3.5.1. 変更データキャプチャーの Db2 テーブルの設定 44
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 46
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 47
3.6. DEBEZIUM DB2 コネクターのデプロイ 48
3.6.1. Db2 JDBC ドライバーの取得 48
3.6.2. AMQ Streams を使用した Db2 コネクターデプロイメント 48
3.6.3. AMQ Streams を使用した Debezium Db2 コネクターのデプロイ 49
3.6.4. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium Db2 コネクターのデプロイ 53
3.6.5. Debezium Db2 コネクターが実行していることの確認 57
3.6.6. Debezium Db2 コネクター設定プロパティーの説明 61
3.7. DEBEZIUM DB2 コネクターのパフォーマンスの監視 77
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 77
3.7.2. Debezium Db2 コネクターレコードストリーミングの監視 79
3.7.3. Debezium Db2 コネクターのスキーマ履歴の監視 81
3.8. DEBEZIUM DB2 コネクターの管理 82
3.9. DEBEZIUM コネクター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 83
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 83
3.9.2. Debezium Db2 コネクターでのオンラインスキーマ更新の実行 84

第4章 MONGODB の DEBEZIUM コネクター **86**
4.1. DEBEZIUM MONGODB コネクターの概要 86
4.2. DEBEZIUM MONGODB コネクターの仕組み 87
4.2.1. Debezium コネクターでサポートされる MongoDB トポロジー 88
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 89
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 89
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 90

4.2.5. 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 90
4.2.6. 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 91
4.2.7. 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 91
4.3. DEBEZIUM MONGODB コネクターのデータ変更イベントの説明 93
4.3.1. Debezium MongoDB 変更イベントのキー 94
4.3.2. Debezium MongoDB 変更イベントの値 96
4.4. DEBEZIUM コネクターと連携する MONGODB の設定 105
4.5. DEBEZIUM MONGODB コネクターのデプロイメント 106
4.5.1. AMQ Streams を使用した MongoDB コネクターデプロイメント 106
4.5.2. AMQ Streams を使用した Debezium MongoDB コネクターのデプロイ 106
4.5.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium MongoDB コネクターのデプロイ 111
4.5.4. Debezium MongoDB コネクターが実行していることの確認 114
4.5.5. Debezium Db2 コネクター設定プロパティーの説明 118
4.6. DEBEZIUM MONGODB コネクターのパフォーマンスの監視 126
4.6.1. MongoDB スナップショット作成中の Debezium の監視 126
4.6.2. Debezium MongoDB コネクターレコードストリーミングの監視 128
4.7. DEBEZIUM MONGODB コネクターによる障害および問題の処理方法 130

第5章 MYSQL の DEBEZIUM コネクター 134
5.1. DEBEZIUM MYSQL コネクターの仕組み 134
5.1.1. Debezium コネクターでサポートされる MySQL トポロジー 135
5.1.2. Debezium MySQL コネクターによるデータベーススキーマの変更の処理方法 136
5.1.3. Debezium MySQL コネクターによるデータベーススキーマの変更の公開方法 136
5.1.4. 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 141
5.1.4.1. アドホックスナップショット 143
5.1.4.2. 増分スナップショット 144
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 148
5.2. DEBEZIUM MYSQL コネクターのデータ変更イベントの説明 150
5.2.1. Debezium MySQL 変更イベントのキー 152
5.2.2. Debezium MySQL 変更イベントの値 153
5.3. DEBEZIUM MYSQL コネクターによるデータ型のマッピング方法 164
5.4. DEBEZIUM コネクターを実行するための MYSQL の設定 170
5.4.1. Debezium コネクターの MySQL ユーザーの作成 170
5.4.2. Debezium の MySQL binlog の有効化 172
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 172
5.4.4. Debezium の MySQL セッションタイムアウトの設定 174
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 174
5.5. DEBEZIUM MYSQL コネクターのデプロイメント 175
5.5.1. AMQ Streams を使用した MySQL コネクターデプロイメント 175
5.5.2. AMQ Streams を使用した Debezium MySQL コネクターのデプロイ 176
5.5.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium MySQL コネクターのデプロイ 180
5.5.4. Debezium MySQL コネクターが実行していることの確認 184
5.5.5. Debezium MySQL コネクター設定プロパティーの説明 188
5.6. DEBEZIUM MYSQL コネクターのパフォーマンスの監視 209
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 209
5.6.2. Debezium MySQL コネクターレコードストリーミングの監視 212
5.6.3. Debezium MySQL コネクターのスキーマ履歴の監視 214
5.7. DEBEZIUM MYSQL コネクターによる障害および問題の処理方法 215

第6章 ORACLE 向けの DEBEZIUM コネクター (テクノロジープレビュー) 218
6.1. DEBEZIUM ORACLE コネクターの仕組み 218

6.1.1. Debezium Oracle 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 219
6.1.1.1. アドホックスナップショット 219
6.1.1.2. 増分スナップショット 221
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 225
6.1.3. Debezium Oracle コネクターによるデータベーススキーマの変更の公開方法 225
6.1.4. トランザクション境界を表す Debezium Oracle コネクターによって生成されたイベント 229
6.1.4.1. 変更データイベントのエンリッチメント 230
6.1.5. Oracle SCN 値間のギャップ 231
6.2. DEBEZIUM ORACLE コネクターのデータ変更イベントの説明 232
6.2.1. Debezium Oracle コネクター変更イベントのキー 233
6.2.2. Debezium Oracle 変更イベントの値 234
6.3. DEBEZIUM ORACLE コネクターによるデータ型のマッピング方法 240
6.4. DEBEZIUM と連携させるための ORACLE の設定 246
6.4.1. Debezium で使用する Oracle データベースの準備 246
6.4.2. ログサイジングのやり直し 247
6.4.3. Debezium Oracle コネクターの Oracle ユーザーの作成 247
6.5. DEBEZIUM ORACLE コネクターのデプロイメント 248
6.5.1. AMQ Streams を使用した Debezium Oracle コネクターのデプロイメント 248
6.5.2. AMQ Streams を使用した Debezium Oracle コネクターのデプロイ 249
6.5.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium Oracle コネクターのデプロイ 253
6.5.4. Oracle JDBC ドライバーの取得 257
6.5.5. コンテナーデータベースとノンコンテナーデータベースの設定 258
6.5.6. Debezium Oracle コネクターが実行していることの確認 258
6.6. DEBEZIUM ORACLE コネクター設定プロパティーの説明 262
6.7. DEBEZIUM ORACLE コネクターのパフォーマンスの監視 282
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 282
6.7.2. Debezium Oracle コネクターのストリーミングメトリクス 285
6.7.3. Debezium Oracle コネクターのスキーマ履歴メトリクス 291
6.8. DEBEZIUM ORACLE コネクターによる障害および問題の処理方法 292

第7章 POSTGRESQL の DEBEZIUM コネクター 294
7.1. DEBEZIUM POSTGRESQL コネクターの概要 294
7.2. DEBEZIUM POSTGRESQL コネクターの仕組み 295
7.2.1. PostgreSQL コネクターのセキュリティー 296
7.2.2. Debezium PostgreSQL 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 296
7.2.2.1. アドホックスナップショット 297
7.2.2.2. 増分スナップショット 298
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 302
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 303
7.2.5. Debezium PostgreSQL 変更イベントレコードのメタデータ 304
7.2.6. 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 305
7.3. DEBEZIUM POSTGRESQL コネクターのデータ変更イベントの説明 307
7.3.1. Debezium PostgreSQL の変更イベントのキー 309
7.3.2. Debezium PostgreSQL 変更イベントの値 310
7.4. DEBEZIUM POSTGRESQL コネクターによるデータ型のマッピング方法 322
7.5. DEBEZIUM コネクターを実行するための POSTGRESQL の設定 334
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 334
7.5.2. Debezium コネクターの PostgreSQL パーミッションの設定 335
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 335
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 336
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 337
7.6. DEBEZIUM POSTGRESQL コネクターのデプロイメント 338

7.6.1. AMQ Streams を使用した PostgreSQL コネクターデプロイメント 338
7.6.2. AMQ Streams を使用した Debezium PostgreSQL コネクターのデプロイ 339
7.6.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium PostgreSQL コネクターのデプロイ 343
7.6.4. Debezium PostgreSQL コネクターが実行していることの確認 346
7.6.5. Debezium PostgreSQL コネクター設定プロパティーの説明 350
7.7. DEBEZIUM POSTGRESQL コネクターのパフォーマンスの監視 373
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 373
7.7.2. Debezium PostgreSQL コネクターレコードストリーミングの監視 376
7.8. DEBEZIUM POSTGRESQL コネクターによる障害および問題の処理方法 378

第8章 SQL SERVER の DEBEZIUM コネクター 381
8.1. DEBEZIUM SQL SERVER コネクターの概要 381
8.2. DEBEZIUM SQL SERVER コネクターの仕組み 382
8.2.1. Debezium SQL Sever 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 382
8.2.1.1. アドホックスナップショット 383
8.2.1.2. 増分スナップショット 384
8.2.2. Debezium SQL Server コネクターによる変更データテーブルの読み取り方法 388
8.2.3. 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 388
8.2.4. Debezium SQL Server コネクターによるスキーマ変更トピックの使用方法 389
8.2.5. Debezium SQL Server コネクターのデータ変更イベントの説明 393
8.2.5.1. Debezium SQL Server 変更イベントのキー 395
8.2.5.2. Debezium SQL Server 変更イベントの値 396
8.2.6. 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 405
8.2.6.1. 変更データイベントのエンリッチメント 407
8.2.7. Debezium SQL Server コネクターによるデータ型のマッピング方法 407
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 412
8.3.1. SQL Server データベースでの CDC の有効化 413
8.3.2. SQL Server テーブルでの CDC の有効化 414
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 415
8.3.4. Azure 上の SQL Server 416
8.3.5. SQL Server キャプチャー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 416
8.3.6. SQL Server のキャプチャージョブエージェントの設定パラメーター 416
8.4. DEBEZIUM SQL SERVER コネクターのデプロイ 417
8.4.1. AMQ Streams を使用した SQL Server コネクターデプロイメント 417
8.4.2. AMQ Streams を使用した Debezium SQL Server コネクターのデプロイ 418
8.4.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium SQL Server コネクターのデプロイ 422
8.4.4. Debezium SQL Server コネクター設定プロパティーの説明 430
8.5. スキーマ変更後のキャプチャーテーブルの更新 445
8.5.1. スキーマの変更後のオフライン更新の実行 446
8.5.2. スキーマの変更後のオンライン更新の実行 447
8.6. DEBEZIUM SQL SERVER コネクターのパフォーマンスの監視 448
8.6.1. Debezium SQL Server コネクターのスナップショットメトリクス 449
8.6.2. Debezium SQL Server コネクターのストリーミングメトリクス 451
8.6.3. Debezium SQL Server コネクターのスキーマ履歴メトリクス 453

第9章 DEBEZIUM の監視 455
9.1. DEBEZIUM コネクターを監視するためのメトリクス 455
9.2. ローカルインストールでの JMX の有効化 455
9.2.1. Zookeeper JMX 環境変数 455
9.2.2. Kafka JMX 環境変数 456
9.2.3. Kafka Connect JMX 環境変数 456

9.3. OPENSHIFT 上での DEBEZIUM の監視 456

第10章 DEBEZIUM のログ機能 457
10.1. DEBEZIUM ログの概念 457
10.2. デフォルトの DEBEZIUM ログ設定 457
10.3. DEBEZIUM ログの設定 458
10.3.1. ロガーを設定して Debezium のログレベルを変更する 458
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 460
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 460
10.4. OPENSHIFT での DEBEZIUM ログ 461

第11章 アプリケーション用 DEBEZIUM コネクターの設定 462
11.1. KAFKA CONNECT 自動トピック作成のカスタマイズ 462
11.1.1. Kafka ブローカーの自動トピック作成の無効化 463
11.1.2. Kafka Connect の自動トピック作成の設定 463
11.1.3. 自動的に作成されたトピックの設定 464
11.1.3.1. トピック作成グループ 464
11.1.3.2. トピック作成グループの設定プロパティー 464
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 465
11.1.3.4. Debezium カスタムトピック作成グループ設定の指定 466
11.1.3.5. Debezium カスタムトピック作成グループの登録 467
11.2. AVRO シリアライゼーションを使用する DEBEZIUM コネクターの設定 468
11.2.1. Service Registry の概要 469
11.2.2. Avro シリアライゼーションを使用する Debezium コネクターのデプロイの概要 470
11.2.3. Debezium コンテナーで Avro を使用するコネクターのデプロイ 470
11.2.4. Avro の名前の要件について 474
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 474
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 475
11.3.2. Debezium CloudEvents コンバーターの設定例 477
11.3.3. Debezium CloudEvents コンバーター設定オプション 478
11.4. DEBEZIUM コネクターへのシグナル送信 478
11.4.1. Debezium シグナリングの有効化 479
11.4.1.1. デベージアムシグナリングデータ収集の必須構造 480
11.4.1.2. Debezium シグナルのデータコレクションの作成 480
11.4.2. デベージアムシグナルアクションの種類 481
11.4.2.1. ロギング信号 481
11.4.2.2. アドホックなスナップショット信号 481
11.4.2.3. 増分スナップショット 482

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 484
12.1. SMT 述語を使用した変換の選択的適用 484
12.1.1. SMT 述語について 484
12.1.2. SMT 述語の定義 486
12.1.3. 廃棄 (tombstone) イベントの無視 487
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 488
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 488
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 489
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 490
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 491
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 491
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 492
12.3.1. Debezium コンテンツベースのルーティング SMT の設定 493
12.3.2. 以下に例を示します。Debezium コンテンツベースルーティングの基本設定 494
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 494

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 495
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 495
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 496
12.4. DEBEZIUM 変更イベントレコードの絞り込み 497
12.4.1. Debezium フィルター SMT の設定 498
12.4.2. 以下に例を示します。Debezium ベーシックフィルターの SMT 設定 498
12.4.3. フィルターの式で使用される変数 499
12.4.4. フィルター変換を一部適用するオプション 500
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 500
12.4.6. フィルター変換設定用のオプション 501
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 501
12.5.1. Debezium 変更イベントの構造について 502
12.5.2. Debezium イベントフラット化変換の動作 503
12.5.3. Debezium イベントフラット化変換の設定 503
12.5.4. Kafka レコードに Debezium メタデータを追加する例 504
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 505
12.5.6. Debezium イベントフラット化変換設定用のオプション 505
12.6. 送信トレイパターンを使用する DEBEZIUM コネクターの設定 508
12.6.1. Debezium 送信トレイメッセージの例 509
12.6.2. 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 510
12.6.3. Debezium 送信トレイイベントルーター SMT の基本設定 511
12.6.4. 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 511
12.6.5. 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 512
12.6.6. Debezium 送信トレイメッセージへの追加フィールドの出力 512
12.6.7. JSON としてエスケープされた JSON 文字列の拡張 513
12.6.8. 送信トレイイベントルーター変換設定用のオプション 513
12.7. 送信トレイパターンを使用する DEBEZIUM MONGODB コネクターの設定 516
12.7.1. Debezium MongoDB 送信トレイメッセージの例 517
12.7.2. 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 518
12.7.3. 基本的な Debezium MongoDB 送信トレイイベントルーター SMT 設定 520
12.7.4. 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 520
12.7.5. 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 520
12.7.6. Debezium MongoDB 送信トレイメッセージへの追加フィールドの出力 521
12.7.7. JSON としてエスケープされた JSON 文字列の拡張 521
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 522

前書き

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 コネクター*
- 4章*MongoDB の Debezium コネクター*
- 5章*MySQL の Debezium コネクター*
- 6章*Oracle 向けの Debezium コネクター (テクノロジープレビュー)*
- 7章*PostgreSQL の Debezium コネクター*
- 8章*SQL Server の Debezium コネクター*
- 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 コネクターの設定*
- 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*

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 プロパティー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、フィルターリング、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。

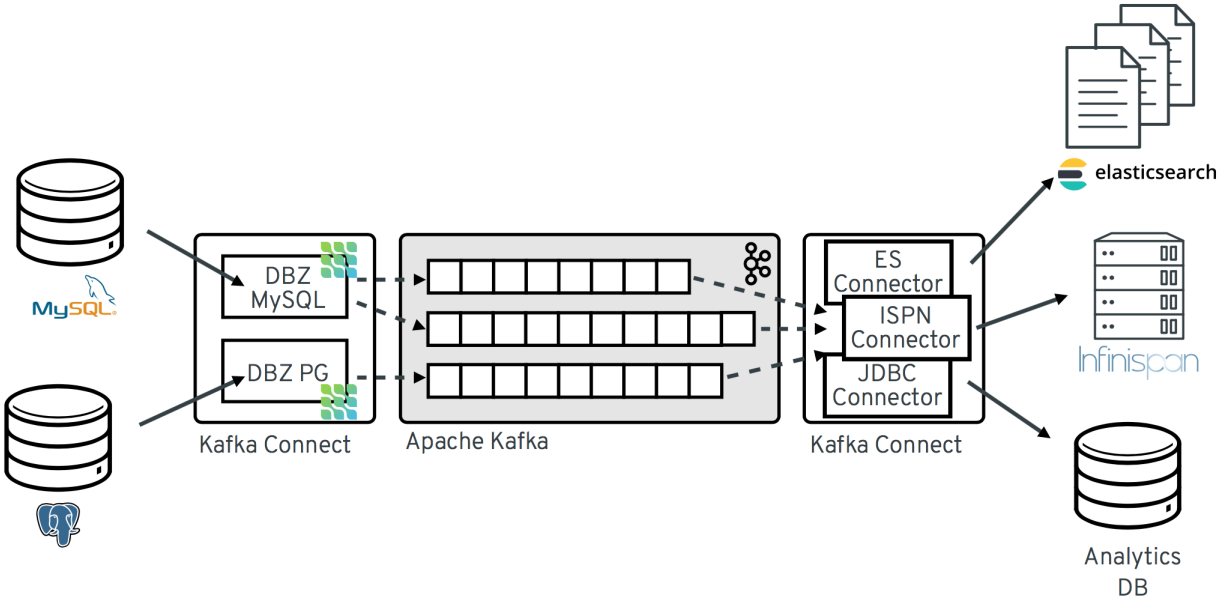
各コネクターのドキュメントには、コネクター機能と設定オプションの詳細が記載されています。

1.2. DEBEZIUM アーキテクチャーの説明

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ー
- 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ー

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ーを示しています。



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

- MySQL コネクターは、**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 コネクター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ーは、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ーによっては、Debezium の new record state extraction の変換を設定する必要が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ーに **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 コネクタープラグインです。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 コネクターの設定に使用する 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 Streams2.0 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 Streams2.0 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 コネクター

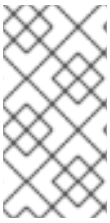
Debezium の Db2 コネクターは、Db2 データベースのテーブルで行レベルの変更をキャプチャーできます。このコネクターと互換性のある Db2 データベースのバージョンについては、Debezium でサポートされる設定ページを参照してください。

このコネクターは、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 コネクターは、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ーが含まれます。また、削除に関する特別なエントリーもあります。Debezium Db2 コネクター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 コネクターが Db2 データベースに初めて接続すると、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ー設定プロパティーがあります。

スナップショットが完了すると、コネクター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

コネクターには、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 コネクター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 コネクターの概要」
- 「Debezium Db2 コネクターの仕組み」
- 「Debezium Db2 コネクターのデータ変更イベントの説明」
- 「Debezium Db2 コネクターによるデータ型のマッピング方法」
- 「Debezium コネクターを実行するための Db2 の設定」
- 「Debezium Db2 コネクターのデプロイ」
- 「Debezium Db2 コネクターのパフォーマンスの監視」
- 「Debezium Db2 コネクターの管理」
- 「Debezium コネクター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 コネクターの概要

Debezium Db2 コネクターは、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 コネクターは 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ーを実行します。

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 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、コネクター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。コネクターは、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、コネクター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 コネクターが最初に特定の 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。コネクターは、スナップショットの完成後に、スナップショットが作成された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、コネクター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 コネクターはフォールトトラレントです。コネクター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ー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。コネクターが何らかの理由で停止した場合 (通信障害、ネットワークの問題、クラッシュなど)、コネクター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、コネクターの停止時にスナップショットが完了しなかった場合、コネクター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 コネクターの仕組み

Debezium Db2 コネクターを最適に設定および実行するには、コネクター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 コネクターによるデータベーススナップショットの実行方法」

- 「Debezium Db2 コネクター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 コネクター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 コネクターによって生成されたイベント」

3.2.1. Debezium Db2 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

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 コネクター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 コネクター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ければなら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** などのコネクター設定プロパティーを使用すると、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す。

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** コネクター設定プロパティーによって決定されます。

3. サーバーのトランザクションログで、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。コネクターは、内部データベース履歴トピックでこの情報を永続化します。

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、コネクター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、コネクターは以下を行います。

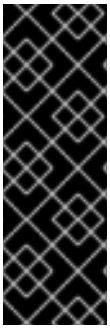
    a. スナップショットの開始前に、テーブルが作成されたことを確認します。そうでない場合は、スナップショットはそのテーブルをスキップします。スナップショットが完了し、コネクターが変更イベントの出力を開始した後、コネクターはスナップショットの実行中に作成されたテーブルの変更イベントを生成します。

    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. コネクターオフセットにスナップショットの正常な完了を記録します。

3.2.1.1. 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の使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。通常の状況では、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、コネクター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。

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**	スナップショットを作成するテーブルの完全修飾名が含まれる配列。 名前の形式は **signal.data.collection** 設定オプションと同じです。

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

重要

増分スナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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 1 KB 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、ストリーム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ーに提供します。スナップショットウィンドウ中に、バッファー内の **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、バッファーされた **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、バッファーに含まれるのは、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。

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。SQL **INSERT** クエリーとしてテーブルにシグナルを送信します。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、テーブルを一覧表示する **data-collections** アレイを指定します (例: **{"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。

前提条件

- シグナルが有効になっている。
  - シグナルデータコレクションがソースのデータベースに存在し、コネクターはこれをキャプチャーするように設定されています。
  - 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>_"}');
```

以下に例を示します。

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

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

以下の表では、これら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

値	説明
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。
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。

値	説明
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。
data-collections	スナップショットに含めるテーブル名の配列を指定するシグナルの **data** フィールドの必須コンポーネント。 配列は、**signal.data.collection** 設定プロパティーにコネクターのシグナルテーブルの名前を指定するときに使用する形式で、完全修飾名別にテーブルを一覧表示します。
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。 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。

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。

以下に例を示します。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。
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。

警告

Db2 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。

3.2.2. Debezium Db2 コネクター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 コネクター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。コネクター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す。
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、コネクター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. コネクターが Kafka Connect に渡した最大 LSN を保存します。

再起動後、コネクター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。コネクターが稼働し、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、コネクターは変更を検出して、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。

databaseName.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。

databaseName

database.server.name コネクター設定プロパティーで指定したコネクターの論理名です。

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、4 つのテーブルが含まれる **mydatabase** データベースを使用する Db2 インストールについて考えてみましょう。**MYSCHEMA** スキーマにある **PRODUCTS**、**PRODUCTS_ON_HAND**、**CUSTOMERS**、および **ORDERS**コネクター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**

コネクターは同様の命名規則を適用して、内部データベース履歴トピック (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 コネクターのスキーマ変更トピック

Debezium Db2 コネクターを設定すると、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。

コネクターは、スキーマ変更イベントをすべて **<serverName>** という名前の Kafka トピックに書き込みます。**<serverName>** は **database.server.name** 設定プロパティーに指定されたコネクターの名前に置き換えます。コネクター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ー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、内部データベース履歴トピックにも格納します。内部データベース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。

以下に例を示します。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": "1.7.2.Final",
      "connector": "db2",
      "name": "db2",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "testdb",
      "schema": "DB2INST1",
      "table": "CUSTOMERS",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "databaseName": "TESTDB", (1)
    "schemaName": "DB2INST1",
    "ddl": null, (2)
    "tableChanges": [ (3)
      {
        "type": "CREATE", (4)
        "id": "\"DB2INST1\".\"CUSTOMERS\"", (5)
```

```
"table": { 6
  "defaultCharsetName": null,
  "primaryKeyColumnNames": [ 7
    "ID"
  ],
  "columns": [ 8
    {
      "name": "ID",
      "jdbcType": 4,
      "nativeType": null,
      "typeName": "int identity",
      "typeExpression": "int identity",
      "charsetName": null,
      "length": 10,
      "scale": 0,
      "position": 1,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "FIRST_NAME",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
      "typeExpression": "varchar",
      "charsetName": null,
      "length": 255,
      "scale": null,
      "position": 2,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "LAST_NAME",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
      "typeExpression": "varchar",
      "charsetName": null,
      "length": 255,
      "scale": null,
      "position": 3,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "EMAIL",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
      "typeExpression": "varchar",
      "charsetName": null,
      "length": 255,
```

```
            "scale": null,
            "position": 4,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          }
        ]
      }
    }
  ]
 }
}
```

表3.3 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	**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。
2	**ddl**	Db2 コネクターの場合は常に **null** です。その他のコネクターでは、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 コネクターでは使用できません。
3	**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。
4	**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
5	**id**	作成、変更、または破棄されたテーブルの完全な識別子。
6	**table**	適用された変更後のテーブルメタデータを表します。
7	**primaryKeyColumnNames**	テーブルのプライマリーキーを設定する列のリスト。
8	**列**	変更されたテーブルの各列のメタデータ。

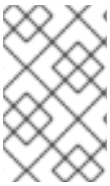
コネクターがスキーマ変更トピックに送信するメッセージでは、メッセージキーはスキーマの変更が含まれるデータベースの名前です。以下の例では、**payload** フィールドにキーが含まれます。

```
{
  "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。

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",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "00000025:00000d08:0025",
  "event_count": 2,
  "data_collections": [
```

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

コネクターはトランザクションイベントを **<database.server.name>.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、コネクターは変更イベント **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 コネクターのデータ変更イベントの説明

Debezium Db2 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": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

表3.4 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ー 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ー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

Debezium Db2 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。

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、コネクターは同じ 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。

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のようになります。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": "mydatabase.MYSCHEMA.CUSTOMERS.Key" 4
   },
   "payload": { 5
      "ID": 1004
   }
}
```

表3.5 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	**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.MYSCHEMA.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-name.table-name.**Key** です。この例では、以下のようになります。 • **mydatabase** はこのイベントを生成したコネクターの名前です。 • **MYSCHEMA** 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**	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。

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

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

```
        "name": "io.debezium.connector.db2.Source", 3
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "mydatabase.MYSCHEMA.CUSTOMERS.Envelope" 4
  },
  "payload": { 5
    "before": null, 6
    "after": { 7
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "source": { 8
      "version": "1.7.2.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
    },
    "op": "c", 9
    "ts_ms": 1559729471739 10
  }
}
```

表3.6 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**	**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。 **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。コネクターは、**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。 **before** および **after** フィールドのスキーマ名は**logicalName.schemaName.tableName.Value** の形式を取るので、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 スキーマには独自の進化と履歴があります。
3	**name**	**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**	**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。
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。
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**	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、トレーサビリティーが提供されます。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かを認識できます。ソースメタデータには以下が含まれています。 ● 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**	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。 ● **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**	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。 **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。

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": { 1
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "john.doe@example.org"
    },
    "after": { 2
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "noreply@example.org"
    },
    "source": { 3
      "version": "1.7.2.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
    },
    "op": "u", 4
    "ts_ms": 1559729998706 5
  }
}
```

表3.7 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**	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かを認識できます。ソースメタデータには以下が含まれています。 ● 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**	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。
5	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。

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": {  1
      "ID": 1005,
      "FIRST_NAME": "john",
      "LAST_NAME": "doe",
      "EMAIL": "noreply@example.org"
    },
    "after": null,  2
    "source": {  3
      "version": "1.7.2.Final",
      "connector": "db2",
      "name": "myconnector",
      "ts_ms": 1559730445243,
      "snapshot": false,
      "db": "mydatabase",
      "schema": "MYSCHEMA",
      "table": "CUSTOMERS",
      "change_lsn": "00000027:00000db0:0005",
      "commit_lsn": "00000027:00000db0:0007"
    },
    "op": "d",  4
    "ts_ms": 1559730450205  5
  }
}
```

表3.8 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**	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。 ● 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**	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**	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。 **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。

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 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。

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3.12「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。

- **リテラル型** は、Kafka Connect スキーマ型を使用して、値を表す方法を記述します。**INT8**、**INT16**、**INT32**、**INT64**、**FLOAT32**、**FLOAT64**、**BOOLEAN**、**STRING**、**BYTES**、**ARRAY**、**MAP**、および **STRUCT**。
- **セマンティック型** 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3.9 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
DECFLOAT	**BYTES**	**org.apache.kafka.connect.data.Decimal**

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
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ー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 設定プロパティーがデフォルトの **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3.10 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。
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=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、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3.11 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。
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。
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。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。

タイムスタンプ型

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ません。

表3.12 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。
SMALLMONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。
MONEY	**BYTES**	**org.apache.kafka.connect.data.Decimal** **scale** スキーマパラメーターには、小数点を移動した桁数を表す整数が含まれます。**connect.decimal.precision** スキーマパラメーターには、指定の 10 進数値の精度を表す整数が含まれます。

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

- **db2instl** 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
```

5. 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。**db2instl** 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
```

6. 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
```

7. 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
```

8. ASN 制御テーブルを作成します。

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

9. 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。

10. 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 &
```

11. 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。**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。

```
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

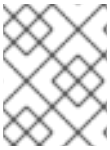
エージェントの実行が開始されます。エージェントは新しい変更イベントレコードをトランザクションログから読み取り、イベントレコードをキャプチャーテーブルに複製します。変更がソーステーブルにコミットされてから、対応する変更テーブルに変更が反映される間、常に短いレイテンシーが間隔で発生します。この遅延間隔は、ソーステーブルで変更が発生したときから、Debezium がその変更を Apache Kafka にストリーミングできるようになるまでの差を表します。

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ー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 コンテナーイメージをビルドします。

Debezium Db2 コネクターでは、Db2 データベースに接続するために Db2JDBC ドライバーが必要です。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ーの説明」

3.6.1. Db2 JDBC ドライバーの取得

ライセンス要件により、Db2 JDBC ドライバーファイルは Debezium Db2 コネクターアーカイブに含まれていません。使用するデプロイメント方法に関係なく、デプロイメントを完了するためにドライバーファイルをダウンロードする必要があります。

以下の手順では、ドライバーを取得してお使いの環境で使用する方法を説明します。

手順

1. ブラウザーから、IBM サポートサイトに ナビゲートし、ご使用のバージョンの Db2 に一致する JDBC ドライバーをダウンロードします。
   - Dockerfile を使用してコネクターを構築する場合は、ダウンロードしたファイルを Debezium Db2 コネクターのファイルがあるディレクトリー、たとえば **<kafka_home>/libs** ディレクトリーにコピーします。
   - AMQ ストリームを使用してコネクターを Kafka Connect イメージに追加する 場合:
     a. このドライバーを Maven リポジトリーまたは OpenShift クラスターで利用できる別の HTTP サーバーにデプロイします。
     b. アーティファクト URL を **KafkaConnect** カスタムリソースに追加します。

KafkaConnector リソースを適用してコネクターをデプロイすると、コネクターは指定されたドライバーを使用するよう設定されます。

3.6.2. AMQ Streams を使用した Db2 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 コンテナーイメージをビルドすることです。

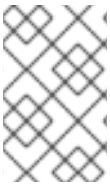
デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- 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**コンテナーイメージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ー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 にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。

- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

  - 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 として保存します。

  - 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。

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、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

例3.1 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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-db2
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-db2/1.7.2.Final-redhat-<build_number>/debezium-connector-db2-1.7.2.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.0-redhat-<build-number>.zip
          - type: zip
```

```
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表3.13 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、クラスターオペレーターが **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ー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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメント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。

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

```
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ー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、**db2-inventory-connector.yaml** として保存します。

例3.2 Debezium コネクターの 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)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: db2.debezium-db2.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_db2 (10)
    database.include.list: public.inventory (11)
```

表3.14 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 がデータベースに接続するユーザーアカウントの名前。

項目	説明
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター 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 コンテナーイメージをビルドして Debezium Db2 コネクターのデプロイ

Debezium Db2 コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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

- Db2 が実行中で、Db2 を設定して Debezium コネクターと連携する 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。
- Db2 に必要な JDBC ドライバーを 取得している。
- Debezium コネクターを実行するコンテナーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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 コンテナーを作成します。
   a. Debezium Db2 コネクターアーカイブをダウンロードします。
   b. Debezium Db2 コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

   c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-db2.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) 任意のファイル名を指定できます。

(2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

   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。

   d. 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ー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ーイメージを構築します。

   e. カスタムイメージを quay.io などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** および **image** プロパティーを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。

```
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-db2 2
```

1 **KafkaConnector** リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。

g. 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 Db2 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、コネクターに使用できる設定プロパティーを使用して、**.yaml** ファイルに Debezium Db2 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コラムで 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ー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 コネクターを設定します。このホストには、**mydatabase** という名前のデータベース、名前が **inventory**というテーブルがあり、**fulfillment** がサーバーの論理名です。

Db2 inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
   name: inventory-connector 1
```

```
    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)
      database.server.name: fullfillment (10)
      database.include.list: public.inventory (11)
```

表3.15 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	namespace を形成する Db2 インスタンス/クラスターの論理名で、コネクターが書き込む Kafka トピックの名前、Kafka Connect スキーマ名、および Arvo コネクター が使用される場合に対応する Avro スキーマの namespace のすべてに使用されます。
11	Debezium が変更をキャプチャーする必要があるすべてのテーブルのリスト。

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 **mydatabase** データベースに対して実行を開始します。

Debezium Db2 コネクターに設定できる設定プロパティーの完全リストは、Db2 コネクタープロパティー 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。

3.6.5. Debezium Db2 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 コネクターは 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-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_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

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---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

   - 端末画面で、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.connector.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":"1.7.2.Final-redhat-00001","connector":"db2","name":"inventory_connector_db2","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"db2-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。

3.6.6. Debezium Db2 コネクター設定プロパティーの説明

Debezium Db2 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ーが多数あります。多くのプロパティーにはデフォルト値があります。プロパティーに関する情報は、以下のように設定されています。

- 必要な設定プロパティー
- 高度な設定プロパティー
- 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設定プロパティー。
  - パススルーデータベースの履歴プロパティー
- 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ー。

必要な Debezium Db2 コネクター設定プロパティー

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

プロパティー	デフォルト	説明
name	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。このプロパティーはすべての 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。Db2 コネクターには、常に **io.debezium.connector.db2.Db2Connector** の値を使用します。
tasks.max	**1**	このコネクターのために作成する必要のあるタスクの最大数。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。

プロパティー	デフォルト	説明
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
database.server.name	デフォルトなし	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 を特定および提供する論理名。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。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターから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。
table.include.list	デフォルトなし	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)。include リストに含まれていないテーブルの変更はキャプチャーされません。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。また、**table.exclude.list** プロパティーを設定しないでください。
table.exclude.list	デフォルトなし	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。また、**table.include.list** プロパティーを設定しないでください。

プロパティー	デフォルト	説明
column.exclude.list	空の文字列	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。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。
column.mask.hash.hashAlgorithm.with.salt.salt	該当なし	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 schemaName.tableName.columnName です。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。 仮名は、指定された hashAlgorithm と salt 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、参照整合性は維持され、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 section に説明されています。 以下の例では、**CzQMA0cB5K** が無作為に選択された salt になります。 column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName 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。 使用される hashAlgorithm、選択された salt、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。

プロパティー	デフォルト	説明
time.precision.mode	**adaptive**	時間、日付、およびタイムスタンプは、異なる精度の種類で表すことができます。 **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。 **connect** は、Kafka Connect の **Time**、**Date**、および **Timestamp** の組み込み表現を使用して、常に時間とタイムスタンプ値を表し、データベース列の精度に関わらず、ミリ秒の精度を使用します。時間的価値 を参照します。
tombstones.on.delete	**true**	廃棄 (tombstone) イベントの後に**削除**イベントが続くかどうかを制御します。 **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。 **false** - delete イベントのみが出力されます。 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。
include.schema.changes	**true**	コネクターがデータベーススキーマの変更を、データベースサーバー ID と同じ名前の Kafka トピックに公開するかどうかを指定するブール値。各スキーマの変更は、データベース名が含まれるキーと、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履歴を内部で記録する方法には依存しません。
column.truncate.to._length_.chars	**該当なし**	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 **schemaName.tableName.columnName** です。変更イベントレコードでは、これらの列の値がプロパティー名の **長さ** によって指定される文字数よりも長い場合は切り捨てられます。単一の設定で、異なる長さを持つ複数のプロパティーを指定できます。長さは正の整数である必要があります (例:**column.truncate.to.20.chars**)。

プロパティー	デフォルト	説明
column.mask.with._length_.chars	該当なし	文字ベースの列の完全修飾名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。変更イベント値では、指定のテーブルコラムの値はアスタリスク (*) の長さ (数) に置き換えられます。単一の設定で、異なる長さを持つ複数のプロパティーを指定できます。長さは正の整数またはゼロでなければなりません。ゼロを指定すると、コネクターは値を空の文字列に置き換えます。
column.propagate.source.type	該当なし	列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。列の完全修飾名の形式は、databaseName.tableName.columnName または databaseName.schemaName.tableName.columnName です。 コネクターは指定された各列に対して、列の元の型と元の長さをパラメーターとして、出力された変更レコードの対応するフィールドスキーマに追加します。以下の追加されたスキーマパラメーターは、元の型名と可変幅型の元の長さを伝播します。 **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** このプロパティーは、シンクデータベースの対応するコラムのサイズを適切に調整する場合に便利です。

プロパティー	デフォルト	説明
datatype.propagate.source.type	該当なし	一部の列のデータベース固有のデータ型名と一致する正規表現のコンマ区切りリスト (任意)。完全修飾データ型名の形式は、databaseName.tableName.typeName または databaseName.schemaName.tableName.typeName です。 これらのデータタイプでは、コネクターは出力された変更レコードの対応するフィールドスキーマにパラメーターを追加します。追加されたパラメーターは、列の元の型と長さを指定します。 **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** これらのパラメーターは、それぞれ可変幅型の列の元の型名と長さを伝播します。このプロパティーは、シンクデータベースの対応する列のサイズを適切に調整するのに便利です。 Db2 固有のデータ型名の一覧は、Db2 data types を参照してください。

プロパティー	デフォルト	説明
message.key.columns	空の文字列	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。 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。 テーブルのカスタムメッセージキーを作成するには、テーブルとメッセージキーとして使用する列をリストします。各リストエントリーは以下の形式を取ります。 **<fully-qualified_tableName>:_<keyColumn>_,<keyColumn>** 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。 **<schemaName>.<tableName>** プロパティーは複数のテーブルのエントリーをリストできます。リスト内の異なるテーブルのエントリーは、セミコロンを使用して、区切ります。 以下の例では、テーブル **inventory.customers および** buy **orders**: **inventory.customers:pk1,pk2;(.*).purchaseorders:pk3,pk4** のメッセージキーを設定します。上記の例では、**pk1** と **pk2** 列はテーブル **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。

高度なコネクター設定プロパティー

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

プロパティー	デフォルト	説明
snapshot.mode	**Initial**	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。 **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、データの完全な表現で Kafka トピックに入力する場合に便利です。 **schema_only** - 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。

プロパティー	デフォルト	説明
snapshot.isolation.mode	repeatable_read	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。 **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。 **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。 **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。 **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。
event.processing.failure.handling.mode	fail	イベントの処理中にコネクター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。 **fail** - コネクター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。 **warn** - コネクター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。 **skip** - コネクター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。

プロパティー	デフォルト	説明
poll.interval.ms	1000	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 1000 ミリ秒 (1 秒) です。
max.queue.size	8192	ブロッキングキューの最大サイズの正の整数値。コネクターは、データベースログから読み取る変更イベントをブロッキングキューに配置してから Kafka に書き込みます。このキューは、たとえば Kafka へのレコードの書き込みが遅い場合や Kafka が利用できない場合などに、変更データテーブルを読み取るためのバックプレシャーを提供できます。キューに表示されるイベントは、コネクターによって定期的に記録されるオフセットには含まれません。**max.queue.size** の値は常に **max.batch.size** コネクター設定プロパティーの値よりも大きくする必要があります。
max.batch.size	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。
max.queue.size.in.bytes	0	ブロッキングキューの最大サイズ (バイト単位) の long 値。この機能はデフォルトで無効になっています。正の long 値が設定されると有効になります。

プロパティー	デフォルト	説明
heartbeat.interval.ms	**0**	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。 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、このプロパティーを、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。 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。
heartbeat.topics.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名の接頭辞を指定します。このトピック名の形式は **<heartbeat.topics.prefix>.<server.name>** です。
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、このプロパティーは、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	**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。このプロパティーは、バッチの行の最大数を指定します。

プロパティー	デフォルト	説明
snapshot.lock.timeout.ms	10000	スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する正の整数値。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。他の設定可能な値は以下のとおりです。 **0** - ロックを取得できないとすぐに失敗します。 **-1** - コネクターは永久に待機します。

プロパティー	デフォルト	説明
snapshot.select.statement.overrides	デフォルトなし	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、プロパティーを使用します。このプロパティー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。 プロパティーには、**<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、 **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** 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ーを追加して、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ーの名前 (**snapshot.select.statement.overrides.<schemaName>.<tableName>**) を指定します。例: **snapshot.select.statement.overrides.customers.orders**. 例: 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、以下のプロパティーを追加します。 > "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" 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。

プロパティー	デフォルト	説明
sanitize.field.names	コネクターが **key.converter** または **value.converter** プロパティーを 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.metadata	**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、Transaction metadata を参照してください。
skipped.operations	デフォルトなし	ストリーミング中にスキップされる操作タイプのコンマ区切りリスト。操作には、**c** (挿入/作成)、**u** (更新)、および **d** (削除) が含まれます。デフォルトでは、操作はスキップされません。
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。**<schemaName>.<tableName>** の形式を使用してコレクション名を指定します。 シグナル機能はテクノロジープレビュー機能です。
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。 増分スナップショットはテクノロジープレビュー機能です。

Debezium コネクターデータベース履歴設定プロパティー

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 **database.history.*** プロパティー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 **database.history** プロパティーについて説明しています。

表3.16 コネクターデータベース履歴設定プロパティー

プロパティー	デフォルト	説明
database.history.kafka.topic		コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。
database.history.kafka.bootstrap.servers		Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。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。
database.history.kafka.recovery.poll.interval.ms	**100**	永続化されたデータのポーリングが行われている間にコネクターが起動/回復を待つ最大時間 (ミリ秒単位) を指定する整数値。デフォルトは 100 ミリ秒です。
database.history.kafka.recovery.attempts	**4**	エラーでコネクターのリカバリーが失敗する前に、コネクターが永続化された履歴データの読み取りを試行する最大回数。データが受信されなかった場合に最大待機する時間は、**recovery.attempts** x **recovery.poll.interval.ms** です。
database.history.skip.unparseable.ddl	**false**	コネクターが不正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。
database.history.store.only.monitored.tables.ddl **今後のリリースで非推奨になり、削除される予定です。代わりに database.history.store.only.captured.tables.ddl を使用してください。**	**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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 安全なデフォルトは **false** です。

プロパティー	デフォルト	説明
database.history.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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 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ー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履歴プロパティー

Debezium は、Kafka プロデューサーを使用して、データベース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。**database.history.producer.*** および **database.history.consumer.*** 接頭辞で始まるパススルー設定プロパティー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ー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ーデータベース履歴プロパティーは、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。

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

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

Debezium は、プロパティーを Kafka クライアントに渡す前に、プロパティー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ー および Kafka コンシューマー設定プロパティーの詳細は、Kafka のドキュメントを参照してください。

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ー

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ーは、接頭辞 **database.*** で始まります。たとえば、コネクターは **database.foobar=false** などのプロパティーを JDBC URL に渡します。

データベース履歴クライアントのパススループロパティー の場合のように、Debezium はプロパティー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ときに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 モニター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。

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	型	説明
LastEvent	**string**	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	**long**	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	**long**	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	**long**	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **:非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	**string[]**	コネクターによって監視されるテーブルの一覧。

属性	型	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。
SnapshotAborted	boolean	スナップショットが中断されたかどうか。
SnapshotCompleted	boolean	スナップショットが完了したかどうか。
SnapshotDurationInSeconds	long	スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。
RowsScanned	Map<String, Long>	スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。

属性	型	説明
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

コネクターは、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	型	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。

重要

増分スナップショットの使用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 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。

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	型	説明

属性	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに 'CapturedTables' メトリクスを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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	型	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。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (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	型	説明
Status	string	データベース履歴の状態を示す **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。

属性	型	説明
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 コネクターの管理

Debezium Db2 コネクターをデプロイしたら、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3.17 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');**

タスク	コマンドおよび注記
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 コネクター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 コネクター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ー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もあります。Debezium Db2 コネクター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持つ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、新しいスキーマを持つ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 コネクターを再起動します。

3.9.2. Debezium Db2 コネクター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 コネクターを停止しません。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ーモードの場合は、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、コネクター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ーモードのテーブル 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 キャプチャー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 キャプチャーサービスを再初期化します。
8. 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。

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 キャプチャー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 コネクター

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

このコネクターと互換性のある 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 コネクター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 コネクターの概要」
- 「Debezium MongoDB コネクターの仕組み」
- 「Debezium MongoDB コネクターのデータ変更イベントの説明」
- 「Debezium コネクターと連携する MongoDB の設定」
- 「Debezium MongoDB コネクターのデプロイメント」
- 「Debezium MongoDB コネクターのパフォーマンスの監視」
- 「Debezium MongoDB コネクターによる障害および問題の処理方法」

4.1. DEBEZIUM MONGODB コネクターの概要

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 コネクターは、レプリカセットまたはシャードクラスターの変更をキャプチャーします。

MongoDB **レプリカセット** は、すべてが同じデータのコピーを持つサーバーのセットで設定され、レプリケーションによって、クライアントがレプリカセットの **プライマリー** のドキュメントに追加したすべての変更が、**セカンダリー**と呼ばれる別のレプリカセットのサーバーに適用されるようにします。MongoDB のレプリケーションでは、プライマリーが **oplog** (または操作ログ) に変更を記録した後、各セカンダリーがプライマリーの oplog を読み取って、すべての操作を順番に独自のドキュメントに適用します。新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。

MongoDB コネクターはこのレプリケーションメカニズムを使用しますが、実際にはレプリカセットのメンバーにはなりません。ただし、MongoDB のセカンダリーと同様に、コネクターはレプリカセットのプライマリーの oplog を常に読み取ります。また、コネクターが初めてレプリカセットを表示するとき、oplog を確認して最後に記録されたトランザクションを取得した後、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実行します。すべてのデータがコピーされると、コネクターは oplog から読み取った位置から変更をストリーミングします。MongoDB oplog における操作は べき等 であるため、操作の適用回数に関係なく、同じ最終状態になります。

MongoDB コネクターが変更を処理すると、イベントの発生元となる oplog の位置を定期的に記録します。MongoDB コネクターが停止したときに、最後に処理した oplog の位置を記録するため、再起動時にはその位置からストリーミングが開始されます。つまり、コネクターを停止、アップグレード、または維持でき、後で再起動できます。イベントを何も失うことなく、停止した場所を正確に特定します。

当然ながら、MongoDB の oplogs は通常は最大サイズに制限されているため、コネクターを長時間停止しないようにしてください。長時間停止すると、oplog の操作によってはコネクターによって読み取られる前にパージされる可能性があります。この場合、コネクターを再起動すると、不足している oplog 操作が検出され、スナップショットが実行されます。その後、変更のストリーミングが続行されます。

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、コネクターがレプリカセットのプライマリーと通信する際に問題が発生した場合は、再接続を試み (ネットワークまたはレプリカセットを圧倒しないように指数バックオフを使用)、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。これにより、コネクターはレプリカセットメンバーシップの変更を動的に調整でき、通信の失敗を自動的に処理できます。

その他のリソース

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

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ープロセス全体でタスクを調整、配布、および管理できるようにします。

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」

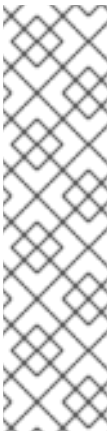
- 「Debezium MongoDB コネクターでレプリカセットおよびシャードクラスターに論理名を使用する方法」
- 「Debezium MongoDB コネクターでのスナップショットの実行方法」
- 「Debezium MongoDB コネクターでの変更イベントレコードのストリーミング方法」
- 「Debezium MongoDB 変更イベントレコードを受信する Kafka トピックのデフォルト名」
- 「イベントキーが Debezium MongoDB コネクターのトピックパーティション設定を制御する方法」
- 「トランザクション境界を表す Debezium MongoDB コネクターに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** プロパティーを使用して、1 つ以上のレプリカセットサーバーのアドレスを **シードアドレス** として提供します。コネクターはこれらのシードを使用してレプリカセットに接続した後、レプリカセットからメンバーの完全セットを取得し、どのメンバーがプライマリーであるかを認識します。コネクターは、プライマリーに接続するタスクを開始し、プライマリーの oplog から変更をキャプチャーします。レプリカセットが新しいプライマリーを選出すると、タスクは自動的に新しいプライマリーに切り替えます。



注記

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

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

MongoDB のシャードクラスター

MongoDB のシャードクラスター は以下で設定されます。

- レプリカセットとしてデプロイされる 1 つ以上の**シャード**。
- クラスターの**設定サーバー**として動作する個別のレプリカセット。
- クライアントが接続し、要求を適切なシャードにルーティングする 1 つ以上の **ルーター** (**mongos** とも呼ばれます)。
  シャードクラスターで MongoDB コネクターを使用するには、コネクターを**設定サーバー**レプリカセットのホストアドレスで設定します。コネクターがこのレプリカセットに接続する

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

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

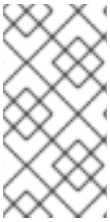
各 MongoDB コネクターに、ソース MongoDB システムを意味する一意の論理名を命名する必要があります。論理名は、アルファベットまたはアンダースコアで始まり、残りの文字を英数字またはアンダースコアとすることが推奨されます。

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** 設定プロパティーの値まで)、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。



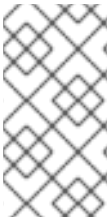
注記

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は、タスクの再割り当てと再設定をしないようにします。コネクターはスナップショットの進捗とともにメッセージをログに記録します。最大限の制御を行う場合は、各コネクターに対して Kafka Connect の個別のクラスターを実行します。

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、タスクはレプリカセットのプライマリーノードに接続し、その位置から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ーである **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

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

前述のように、コネクタータスクは常にレプリカセットのプライマリーノードを使用して oplog からの変更をストリーミングし、コネクター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ーを選出すると、コネクターは即座に変更のストリーミングを停止し、新しいプライマリーに接続して、同じ場所にある新しいプライマリーノードから変更のストリーミングを開始します。同様に、コネクターとレプリカセットメンバーとの通信で問題が発生した場合は、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、コネクターはレプリカセットメンバーシップの変更を動的に調整でき、通信障害を自動的に処理できます。

要約すると、MongoDB コネクター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ーが待機する可能性があります。

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

たとえば、**products**, **products_on_hand**, **customers**, and **orders** の 4 つのコレクションで設定される

inventory データベースを含む MongoDB レプリカセットについて考えてみましょう。コネクターが監視するこのデータベースの論理名が **fulfillment** である場合、コネクターは以下の 4 つの Kafka トピックでイベントを生成します。

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

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

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

4.2.6. 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。

4.2.7. 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
    }
  ]
}
```

transaction.topic オプションでオーバーライドされない限り、トランザクションイベントは **database.server.name.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。

```
{
  "patch": null,
  "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 コネクターのデータ変更イベントの説明

Debezium MongoDB コネクターは、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、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": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

表4.1 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。

項目	フィールド名	説明
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。
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、コネクターによって、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** のコネクター、**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のようになります。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4.2 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。 • **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。

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。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4.3 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

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

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": "string",
         "optional": true,
         "name": "io.debezium.data.Json", 2
         "version": 1,
         "field": "after"
       },
       {
         "type": "string",
         "optional": true,
         "name": "io.debezium.data.Json",
         "version": 1,
         "field": "patch"
       },
       {
         "type": "struct",
         "fields": [
           {
             "type": "string",
             "optional": false,
             "field": "version"
           },
           {
             "type": "string",
```

```
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": false,
            "field": "rs"
          },
          {
            "type": "string",
            "optional": false,
            "field": "collection"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "ord"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "h"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mongo.Source", (3)
        "field": "source"
      },
      {
        "type": "string",
        "optional": true,
        "field": "op"
      },
      {
        "type": "int64",
```

```
          "optional": true,
          "field": "ts_ms"
        }
      ],
      "optional": false,
      "name": "dbserver1.inventory.customers.Envelope" 4
      },
    "payload": { 5
      "after": "{\"_id\" : {\"$numberLong\" : \"1004\"},\"first_name\" : \"Anne\",\"last_name\" :
\"Kretchmar\",\"email\" : \"annek@noanswer.org\"}", 6
      "patch": null,
      "source": { 7
        "version": "1.7.2.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4.4 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。

項目	フィールド名	説明
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。
6	**after**	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。MongoDB の oplog エントリーには、作成イベントのみにドキュメントの完全な状態が含まれます。つまり、**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。
7	**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。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。 ● 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。 ● 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 操作の一意の識別子。これは MongoDB のバージョンに依存します。これは、oplog イベントの **h** フィールド、または oplog イベントの **lsid** および **txnNumber** フィールドを表す **stxnid** という名前のフィールドです。

項目	フィールド名	説明
8	**op**	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。 • **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)
9	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。

更新イベント

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。**更新** イベントには **after** の値はありません。その代わりに、以下の 2 つのフィールドがあります。

- **patch** は、べき等更新操作の JSON 表現が含まれる文字列フィールドです。
- **filter** は、更新の選択基準の JSON 表現が含まれる文字列フィールドです。**filter** 文字列には、シャード化コレクションの複数のシャードキーフィールドを含めることができます。

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)
    "patch": "{\"$set\":{\"first_name\":\"Anne Marie\"}}", (3)
    "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", (4)
    "source": { (5)
      "version": "1.7.2.Final",
      "connector": "mongodb",
      "name": "fulfillment",
      "ts_ms": 1558965508000,
      "snapshot": true,
      "db": "inventory",
      "rs": "rs0",
```

```
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

表4.5 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、**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。 **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	**patch**	ドキュメントへの実際の MongoDB のべき等変更の JSON 文字列表現が含まれます。この例では、更新で **first_name** フィールドを新しい値に変更されています。 **更新** イベント値には **after** フィールドが含まれません。
4	**filter**	更新するドキュメントの特定に使用された MongoDB 選択基準の JSON 文字列表現が含まれます。

項目	フィールド名	説明
5	**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。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。 ● 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 操作の一意の識別子。これは MongoDB のバージョンに依存します。これは、oplog イベントの **h** フィールド、または oplog イベントの **lsid** および **txnNumber** フィールドを表す **stxnid** という名前のフィールドです。

警告

Debezium 変更イベントでは、MongoDB は **patch** フィールドの内容を提供します。このフィールドの形式は、MongoDB データベースのバージョンによって異なります。したがって、新しい MongoDB データベースバージョンにアップグレードする場合は、形式が変更された可能性があるため注意してください。本書のサンプルは、MongoDB 3.4 から取得したため、ご使用のアプリケーションではイベントの形式が異なる場合があります。

注記

MongoDB の oplog では、**更新** イベントには変更されたドキュメントの**前** または **後** の状態は含まれません。そのため、Debezium コネクターがこの情報を提供することはできません。ただし、Debezium コネクターは **作成** および **読み取り** イベントでドキュメントの開始状態を提供します。ストリームのダウンストリームのコンシューマーは、ドキュメントごとに最新状態を維持し、新しいイベントの状態を保存された状態に比較することで、ドキュメント状態を再構築できます。Debezium コネクターはこの状態を維持できません。

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。**delete** 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** の値や **patch** の値は含まれません。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
      "filter": "{\"_id\" : {\"$numberLong\" : \"1004\"}}", 3
      "source": { 4
        "version": "1.7.2.Final",
        "connector": "mongodb",
        "name": "fulfillment",
        "ts_ms": 1558965508000,
        "snapshot": true,
        "db": "inventory",
        "rs": "rs0",
        "collection": "customers",
        "ord": 6,
        "h": 1546547425148721999
      }
    }
  }
```

表4.6 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。 **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	**filter**	削除するドキュメントの特定に使用された MongoDB 選択基準の JSON 文字列表現が含まれます。

項目	フィールド名	説明
4	**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。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。 ● 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 操作の一意の識別子。これは MongoDB のバージョンに依存します。これは、**oplog** イベントの **h** フィールド、または **oplog** イベントの **lsid** および **txnNumber** フィールドを表す **stxnid** という名前のフィールドです。

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

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

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 コンテナーイメージをビルドします。

関連情報

- 「Debezium Db2 コネクター設定プロパティーの説明」

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 コンテナー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**コンテナーイメージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ー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

にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。
- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

  - 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 として保存します。

  - 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。

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、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

   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

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   ```

```
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-mongodb
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mongodb/1.7.2.Final-redhat-<build_number>/debezium-connector-mongodb-1.7.2.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.0-redhat-<build-number>.zip
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表4.7 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、クラスターオペレーターが **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ー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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメントを参照してください。

項目	説明
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。

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

```
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ー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 (1)
spec:
  class: io.debezium.connector.mongodb.MongoDbConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-
```

```
bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: mongodb.debezium-mongodb.svc.cluster.local 5
    database.port: 3306 6
    database.user: debezium 7
    database.password: dbz 8
    database.dbname: mydatabase 9
    database.server.name: inventory_connector_mongodb 10
    database.include.list: public.inventory 11
```

表4.8 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 がデータベースに接続するユーザーアカウントの名前。
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター 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 コンテナーイメージをビルドして Debezium MongoDB コネクターのデプロイ

Debezium MongoDB コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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 コネクターと連携する 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ーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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 コンテナーを作成します。
   a. Debezium MongoDB コネクターアーカイブをダウンロードします。
   b. Debezium MongoDB コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

   c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-mongodb.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
```

```
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) 任意のファイル名を指定できます。

(2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

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。

d. 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ー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ーイメージを構築します。

e. カスタムイメージを **quay.io** などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** および **image** プロパティーを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。

```
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-mongodb (2)
```

(1) **KafkaConnector** リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。

g. 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 PostgreSQL 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、コネクターに使用できる設定プロパティーを使用して、**.yaml** ファイルに Debezium MongoDB コネクターを設定します。コネクター設定で、Debezium に指示を出して MongoDB レプリカセットまたはシャードクラスターのサブセットの変更イベントを生成する場合があります。任意で、不必要なコレクションを除外するプロパティーを設定できます。

以下の例では、MongoDB レプリカセット **rs0** を **192.168.99.100** のポート **27017** に接続し、**インベントリー** コレクションで発生した変更をキャプチャーする Debezium コネクターを設定します。**fullfillment** はレプリカセットの論理名です。

MongoDB inventory-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
  name: inventory-connector 1
  labels: strimzi.io/cluster: my-connect-cluster
 spec:
  class: io.debezium.connector.mongodb.MongoDbConnector 2
  config:
   mongodb.hosts: rs0/192.168.99.100:27017 3
   mongodb.name: fulfillment 4
   collection.include.list: inventory[.]* 5
```

1 コネクターを Kafka Connect に登録するために使用される名前。

2 MongoDB コネクタークラスの名前。

3 MongoDB レプリカセットへの接続に使用するホストアドレス。

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

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

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** コレクションに対して実行を開始します。

Debezium MongoDB コネクターに設定できる設定プロパティーの完全リストは、MongoDB コネクター設定プロパティーを参照してください。

結果

コネクターが起動したら、以下のアクションを完了します。

- MongoDB レプリカセットでコレクションの スナップショット 一貫性をもたせて実行する。
- レプリカセットの oplogs を読み取る。
- 挿入、更新、削除されたすべてのドキュメントの変更イベントを生成する。
- Kafka トピックに変更イベントレコードをストリーミングする。

4.5.4. Debezium MongoDB 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 コネクターは 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-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_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

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                                                                    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-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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

- 端末画面で、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_conne
ctor_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":"1.7.2.Final-redhat-
00001","connector":"mongodb","name":"inventory_connector_mongodb","ts_ms":16389852
47805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","serve
r_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 Db2 コネクター設定プロパティーの説明

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ーが多数あります。多くのプロパティーにはデフォルト値があります。プロパティーに関する情報は、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ー
- 高度な Debezium MongoDB コネクター設定プロパティー

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

表4.9 必要な Debezium MongoDB コネクター設定プロパティー

プロパティー	デフォルト	説明
name		コネクターの一意名。同じ名前で再登録を試みると失敗します。(このプロパティーはすべての Kafka Connect コネクターに必要です)
connector.class		コネクターの Java クラスの名前。MongoDB コネクターには、常に **io.debezium.connector.mongodb.MongoDbConnector** の値を使用します。
mongodb.hosts		レプリカセットでの MongoDB サーバーのホスト名とポートのペア ('host' または 'host:port' 形式) のコンマ区切りリスト。リストには、ホスト名とポートのペアを 1 つ含めることができます。**mongodb.members.auto.discover** を **false** に設定すると、ホストとポートには、レプリカセット名 (**rs0/localhost:27017**) を接頭辞として付ける必要があります。

プロパティー	デフォルト	説明
mongodb.name		このコネクターが監視するコネクターや MongoDB レプリカセット、またはシャードクラスターを識別する一意の名前。このサーバー名は、MongoDB レプリカセットまたはクラスターから生成される永続化されたすべての Kafka トピックの接頭辞になるため、各サーバーは最大 1 つの Debezium コネクターによって監視される必要があります。使用できる文字は、英数字、ハイフン、ドット、アンダースコアのみです。
mongodb.user		MongoDB への接続時に使用されるデータベースユーザーの名前。これは MongoDB が認証を使用するように設定されている場合にのみ必要です。
mongodb.password		MongoDB への接続時に使用されるパスワード。これは MongoDB が認証を使用するように設定されている場合にのみ必要です。
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設定されている場合に必要です。
mongodb.ssl.enabled	**false**	コネクターは SSL を使用して MongoDB インスタンスに接続します。
mongodb.ssl.invalid.hostname.allowed	**false**	SSL が有効な場合、接続フェーズ中に厳密なホスト名のチェックを無効にするかどうかを制御する設定です。**true** に設定すると、接続で中間者攻撃は阻止されません。
database.include.list	**空の文字列**	監視するデータベース名と一致する正規表現のコンマ区切りリスト (任意)。**database.include.list** に含まれていないデータベース名は、監視から除外されます。デフォルトでは、すべてのデータベースが監視されます。**database.exclude.list** と併用しないでください。
database.exclude.list	**空の文字列**	監視から除外されるデータベース名と一致する正規表現のコンマ区切りリスト (任意)。**database.exclude.list** に含まれていないデータベース名が監視の対象となります。**database.include.list** と併用しないでください。

プロパティー	デフォルト	説明
collection.include.list	空の文字列	監視する MongoDB コレクションの完全修飾 namespace と一致する正規表現のコンマ区切りリスト (任意)。**collection.include.list** に含まれていないコレクションはすべて監視から除外されます。各識別子の形式は databaseName.collectionName です。デフォルトでは、**local** および **admin** データベースにあるコレクションを除くすべてのコレクションがコネクターによって監視されます。**collection.exclude.list** と併用しないでください。
collection.exclude.list	空の文字列	監視から除外される MongoDB コレクションの完全修飾 namespace と一致する正規表現のコンマ区切りリスト (任意)。**collection.exclude.list** に含まれていないコレクションはすべて監視されます。各識別子の形式は databaseName.collectionName です。**collection.include.list** と併用しないでください。
snapshot.mode	**Initial**	コネクターの起動時にスナップショットを実行する基準を指定します。デフォルトは **initial** で、オフセットが見つからない場合や oplog に以前のオフセットが含まれなくなった場合にコネクターがスナップショットを読み取るように指定します。**never** オプションは、コネクターはスナップショットを使用せずに、ログをの追跡を続行すべきであることを指定します。
snapshot.include.collection.list	**collection.include.list**に指定されたすべてのコレクション	スナップショットを作成する **collection.include.list** に指定された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)。
field.exclude.list	空の文字列	変更イベントメッセージ値から除外される必要があるフィールドの完全修飾名のコンマ区切りリスト (任意)。フィールドの完全修飾名の形式はdatabaseName.collectionName.fieldName.nestedFieldName で、databaseName および collectionName にはすべての文字と一致するワイルドカード (*) が含まれることがあります。

プロパティー	デフォルト	説明
field.renames	空の文字列	イベントメッセージ値のフィールドの名前を変更するために使用されるフィールドの完全修飾置換のコンマ区切りリスト (任意)。フィールドの完全修飾置換の形式は databaseName.collectionName.fieldName.nestedFieldName:newNestedFieldName で、databaseName および collectionName にはすべての文字と一致するワイルドカード (*) が含まれることがあります。コロン (:) は、フィールドの名前変更マッピングを決定するために使用されます。次のフィールドの置換は、リストの前のフィールド置換の結果に適用されるため、同じパスにある複数のフィールドの名前を変更する場合は、この点に注意してください。
tasks.max	1	このコネクターのために作成する必要のあるタスクの最大数。MongoDB コネクターは各レプリカセットに個別のタスクの使用しようとします。そのため、コネクターを単一の MongoDB レプリカセットと使用する場合は、デフォルトを使用できます。MongoDB のシャードクラスターでコネクターを使用する場合、クラスターのシャード数以上の値を指定して、各レプリカセットの作業が Kafka Connect によって分散されるようにすることが推奨されます。
snapshot.max.threads	1	レプリカセットでコレクションの最初の同期を実行するために使用されるスレッドの最大数を指定する正の整数値。デフォルトは 1 です。
tombstones.on.delete	true	廃棄 (tombstone) イベントの後に**削除**イベントが続くかどうかを制御します。 **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。 **false** - delete イベントのみが出力されます。 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。

プロパティー	デフォルト	説明
snapshot.delay.ms		コネクターの起動後、スナップショットを取得するまで待機する間隔 (ミリ秒単位)。 クラスター内で複数のコネクターを開始する際に スナップショットが中断されないようにするために使用でき、コネクターのリバランスが実行される可能性があります。
snapshot.fetch.size	0	スナップショットの実行中に各コレクションから 1 度に読み取る必要があるドキュメントの最大数を指定します。コネクターは、このサイズの複数のバッチでコレクションの内容を読み取ります。 デフォルトは 0 で、サーバーが適切なフェッチサイズを選択することを示します。

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

表4.10 必要な Debezium MongoDB コネクターの高度な設定プロパティー

プロパティー	デフォルト	説明
max.queue.size	8192	データベースログから読み取られた変更イベントが Kafka に書き込まれる前に配置される、ブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに、oplog リーダーにバックプレシャーを提供できます。キューに発生するイベントは、このコネクターによって定期的に記録されるオフセットには含まれません。デフォルトは 8192 で、常に **max.batch.size** プロパティーに指定された最大バッチサイズよりも大きくする必要があります。
max.batch.size	2048	このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。デフォルトは 2048 です。
max.queue.size.in.bytes	0	ブロッキングキューの最大サイズ (バイト単位) の long 値。この機能はデフォルトで無効になっています。正の long 値が設定されると有効になります。
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ーが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。

プロパティー	デフォルト	説明
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。
connect.max.attempts	16	レプリカセットのプライマリーへの接続を試行する場合の最大失敗回数を指定する正の整数値。この値を越えると、例外が発生し、タスクが中止されます。デフォルトは 16。**connect.backoff.initial.delay.ms** と **connect.backoff.max.delay.ms** のデフォルト値では、 20 分強試行した後にのみ失敗します。
mongodb.members.auto.discover	true	'mongodb.hosts' 内のアドレスがクラスターまたはレプリカセットの全メンバーを検出するために使用されるシードであるかどうか (**true**)、または **mongodb.hosts** のアドレスをそのまま使用する必要があるかどうか (**false**) を指定するブール値。デフォルトは **true** で、MongoDB が プロキシーと面する 場合を除き、すべてのケースで使用する必要があります。

プロパティー	デフォルト	説明
heartbeat.interval.ms	**0**	ハートビートメッセージが送信される頻度を制御します。 このプロパティーには、コネクターがメッセージをハートビートトピックに送信する頻度を定義する間隔 (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 oplog の読み取りを続行しますが、変更メッセージを Kafka に出力しないため、オフセットの更新が Kafka にコミットされません。これにより、oplog ファイルがローテーションされますが、コネクターはこれを認識しないため、再起動時に一部のイベントが利用できなくなり、最初のスナップショットの再実行が必要になります。 このプロパティーを **0** に設定して、ハートビートメッセージが全く送信されないようにします。 デフォルトでは無効にされています。
heartbeat.topics.prefix	**__debezium-heartbeat**	ハートビートメッセージが送信されるトピックの命名を制御します。 トピックは、**<heartbeat.topics.prefix>.<server.name>** パターンに従って名前が付けられます。
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。
skipped.operations		ストリーミング中にスキップされる操作タイプのコンマ区切りリスト。操作には、**c** (挿入/作成)、**u** (更新)、および **d** (削除) が含まれます。デフォルトでは、操作はスキップされません。

プロパティー	デフォルト	説明
snapshot.collection.filter.overrides		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。このプロパティー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。 指定する各コレクションに対して、別の設定プロパティー (**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ーの名前は **snapshot.collection.filter.overrides.customers.orders** などです。このプロパティーは、スナップショットで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。コネクター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。
provide.transaction.metadata	**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。
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)。
mongodb.poll.interval.ms	**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。
mongodb.connect.timeout.ms	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)。
mongodb.socket.timeout.ms	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。
mongodb.server.selection.timeout.ms	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)。

プロパティー	デフォルト	説明
cursor.max.await.time.ms	0	実行タイムアウトの例外を発生させる前に、oplog 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。

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ときにコネクター操作に関する情報を提供します。

Debezium モニター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>** 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	型	説明
LastEvent	string	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルター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	型	説明
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	**string[]**	コネクターによって監視されるテーブルの一覧。
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。
SnapshotAborted	**boolean**	スナップショットが中断されたかどうか。
SnapshotCompleted	**boolean**	スナップショットが完了したかどうか。
SnapshotDurationInSeconds	**long**	スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。
RowsScanned	**Map<String, Long>**	スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。

属性	型	説明
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

Debezium MongoDB コネクターは、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	型	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>**。

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	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。

属性	型	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	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。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

Debezium MongoDB コネクターは、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	型	説明
NumberOfDisconnects	long	データベースの切断数。

属性	型	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
- コネクターの長期間の停止
- 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。

再接の続試行は、3 つのプロパティーで制御されます。

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

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

再接続試行回数	試行までの遅延 (秒単位)	試行までの遅延合計 (分および秒単位)
1	1	00:01
2	2	00:03
3	4	00:07
4	8	00:15
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ータスクをそのグループの別

の Kafka Connect プロセスに移行し、新しいコネクタータスクは、以前のタスクが停止した場所で開始されます。コネクタータスクが正常に停止され、新しいプロセスで再起動されるまでの間、プロセスに短い遅延が発生します。

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

Kafka Connect プロセスのクラッシュ

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

注記

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

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

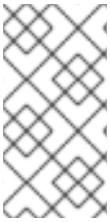
Kafka が使用不可能になる

変更イベントはコネクターによって生成されるため、Kafka Connect フレームワークは、Kafka プロデューサー API を使用してこれらのイベントを記録します。また、Kafka Connect は、これらの変更イベントに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、コネクターを実行する Kafka Connect ワーカープロセスは Kafka ブローカーへの再接続を繰り返し試行します。つまり、コネクタータスクは接続が再確立されるまで一時停止します。接続が再確立されると、コネクターは停止した場所から再開します。

コネクターの長期間の停止

コネクターが正常に停止された場合、レプリカセットは引き続き使用でき、新しい変更は MongoDB の oplog に記録されます。コネクターが再起動されると、最後に停止した場所で各レプリカセットの変更のストリーミングを再開し、コネクターが停止した間に加えられたすべての変更の記録イベントを記録します。コネクターが一定期間停止し、コネクターが読み取っていない一部の操作を MongoDB が oplog からパージするようになると、コネクターは起動時にスナップショットを実行します。

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



注記

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

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ーできません。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 コネクター

重要

本リリースの Debezium MySQL コネクターには、他の Debezium コネクターによって使用される一般的なコネクターフレームワークを基にした新しいデフォルトのキャプチャー実装が含まれています。改訂されたキャプチャーの実装はテクノロジープレビューの機能です。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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

新しいキャプチャー実装で実行中にコネクターによってエラーや予期しない動作が生成される場合は、以下の設定オプションを指定することで以前の実装に戻すことができます。

```
internal.implementation=legacy
```

MySQL には、データベースにコミットされた順序ですべての操作を記録するバイナリーログ (binlog) があります。これには、テーブルスキーマ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 コネクターは 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 コネクター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 コネクターは、スナップショットが作成された時点から binlog を読み取ります。

このコネクターと互換性のある 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 コネクター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 コネクターの仕組み」
- 「Debezium MySQL コネクターのデータ変更イベントの説明」
- 「Debezium MySQL コネクターによるデータ型のマッピング方法」
- 「Debezium コネクターを実行するための MySQL の設定」
- 「Debezium MySQL コネクターのデプロイメント」
- 「Debezium MySQL コネクターのパフォーマンスの監視」
- 「Debezium MySQL コネクターによる障害および問題の処理方法」

5.1. DEBEZIUM MYSQL コネクターの仕組み

コネクターがサポートする 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 コネクターを最適に設定および実行するには、コネクター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と便利です。

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 MySQL トポロジー」
- 「Debezium MySQL コネクターによるデータベーススキーマの変更の処理方法」
- 「Debezium MySQL コネクターによるデータベーススキーマの変更の公開方法」
- 「Debezium MySQL コネクターによるデータベーススナップショットの実行方法」
- 「Debezium MySQL 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 コネクターでサポートされる MySQL トポロジー

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

スタンドアロン

単一の MySQL サーバーを使用する場合は、Debezium MySQL コネクターがサーバーを監視できるように、binlog を有効 (**および任意で GTID を有効**) にする必要があります。バイナリーログも増分バックアップ として使用できるため、これは多くの場合で許容されます。この場合、MySQL コネクターは常にこのスタンドアロン MySQL サーバーインスタンスに接続し、それに従います。

プライマリーおよびレプリカ

Debezium MySQL コネクター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、コネクター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ー以外では問題ではありません。
コネクターは、サーバーの binlog の位置を記録します。この位置は、クラスターの各サーバーごとに異なります。そのため、コネクターは 1 つの MySQL サーバーインスタンスのみに従う必要があります。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、コネクターは継続できません。

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ア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。このトポロジーには GTID を使用する必要があります。
Debezium MySQL コネクターはこれらのマルチプライマリー 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ーを設定できるため、コネクターがデータベースやテーブルのサブセットのみを使用している場合でも機能します。

ホステッド

Debezium MySQL コネクターが Amazon RDS や Amazon Aurora などのホステッドオプションを使用するためのサポートがあります。
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 コネクター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、コネクターはどのスキーマであるかを特定できる必要があります。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ーが処理するので、コネクター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。コネクターは 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。コネクター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。別のデータベース履歴 Kafka トピックでは、コネクターは各 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。

コネクターが正常にクラッシュまたは停止された後にコネクターが再起動されると、コネクターは特定の場所 (特定の時点) から binlog の読み取りを開始します。コネクターは、データベース履歴の Kafka トピックを読み取り、コネクターが起動する 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。

このデータベース履歴トピックはコネクターのみが使用します。コネクターは任意で、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 コネクターが、**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ー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。

Debezium イベントレコードを受信する 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 コネクターによるデータベーススキーマの変更の公開方法

Debezium MySQL コネクターを設定すると、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。コネクターは、スキーマ変更イベントをすべて **<serverName>** という名前の Kafka トピックに書き込みます。**serverName** は **database.server.name** 設定プロパティーに指定されたコネクターの名前になります。コネクター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

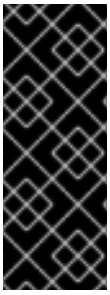
DDL ステートメントが適用されるデータベースの名前。**databaseName** の値は、メッセージキーとして機能します。

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ー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、内部データベース履歴トピックにも格納します。内部データベース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。

以下に例を示します。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": "1.7.2.Final",
        "connector": "mysql",
        "name": "dbserver1",
        "ts_ms": 0,
        "snapshot": "false",
        "db": "inventory",
        "sequence": null,
        "table": "customers",
        "server_id": 0,
        "gtid": null,
        "file": "mysql-bin.000003",
        "pos": 219,
        "row": 0,
        "thread": null,
        "query": null
    },
    "databaseName": "inventory", // (2)
    "schemaName": null,
    "ddl": "ALTER TABLE customers ADD COLUMN middle_name VARCHAR(2000)", // (3)
    "tableChanges": [ // (4)
        {
        "type": "ALTER", // (5)
        "id": "\"inventory\".\"customers\"",  // (6)
        "table": { // (7)
            "defaultCharsetName": "latin1",
            "primaryKeyColumnNames": [  // (8)
                "id"
            ],
            "columns": [ // (9)
                {
                "name": "id",
                "jdbcType": 4,
                "nativeType": null,
                "typeName": "INT",
                "typeExpression": "INT",
                "charsetName": null,
                "length": 11,
                "scale": null,
                "position": 1,
                "optional": false,
                "autoIncremented": true,
                "generated": true
            },
            {
                "name": "first_name",
                "jdbcType": 12,
                "nativeType": null,
                "typeName": "VARCHAR",
                "typeExpression": "VARCHAR",
                "charsetName": "latin1",
                "length": 255,
                "scale": null,
                "position": 2,
                "optional": false,
                "autoIncremented": false,
                "generated": false
```

```
          },                    {
            "name": "last_name",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "VARCHAR",
            "typeExpression": "VARCHAR",
            "charsetName": "latin1",
            "length": 255,
            "scale": null,
            "position": 3,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "email",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "VARCHAR",
            "typeExpression": "VARCHAR",
            "charsetName": "latin1",
            "length": 255,
            "scale": null,
            "position": 4,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          },
          {
            "name": "middle_name",
            "jdbcType": 12,
            "nativeType": null,
            "typeName": "VARCHAR",
            "typeExpression": "VARCHAR",
            "charsetName": "latin1",
            "length": 2000,
            "scale": null,
            "position": 5,
            "optional": true,
            "autoIncremented": false,
            "generated": false
          }
        ]
      }
    }
  ]
 },
 "payload": {
   "databaseName": "inventory",
   "ddl": "CREATE TABLE products ( id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(255) NOT NULL, description VARCHAR(512), weight FLOAT ); ALTER TABLE
products AUTO_INCREMENT = 101;",
   "source" : {
     "version": "1.7.2.Final",
     "name": "mysql-server-1",
     "server_id": 0,
```

```
        "ts_ms": 0,
        "gtid": null,
        "file": "mysql-bin.000003",
        "pos": 154,
        "row": 0,
        "snapshot": true,
        "thread": null,
        "db": null,
        "table": null,
        "query": null
      }
    }
  }
```

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	**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。
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。**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。
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。

項目	フィールド名	説明
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。 **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。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。

スキーマ履歴トピック も参照してください。

5.1.4. 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

Debezium MySQL コネクターが最初に起動すると、データベースの最初の **整合性スナップショット** が実行されます。以下のフローは、コネクター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 コネクター **snapshot.mode** 設定プロパティー 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、コネクターによる 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。コネクターは 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。

ステップ	アクション
3	現在の binlog の位置を読み取ります。
4	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。
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、必要なすべての **DROP...** および **CREATE...** DDL ステートメント。
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	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。

コネクターの再起動

最初のスナップショット の実行中にコネクターが失敗または停止したり、再分散された場合、コネクター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 コネクターは binlog の同じ位置から再起動するため、更新が見逃されることはありません。
コネクターが長時間停止した場合、MySQL が古い binlog ファイルをパージし、コネクターの位置が失われる可能性があります。位置が失われた場合、コネクターは **最初のスナップショット** を開始位置に戻します。Debezium MySQL コネクターのトラブルシューティングに関する詳細は、behavior when things go wrong 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

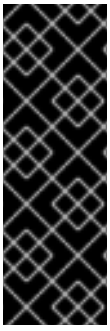
一部の環境では、グローバル読み取りロックが許可されません。Debezium MySQL コネクターがグローバル読み取りロックが許可されないことを検出すると、代わりにテーブルレベルロックを使用して、この方法でスナップショットを実行します。これには、Debezium コネクターのデータベースユーザーに **LOCK TABLES** 権限が必要になります。

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。

ステップ	アクション
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、必要なすべての **DROP...** および **CREATE...** DDL ステートメント。
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



重要

アドホックスナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。通常の状況では、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、コネクター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。

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

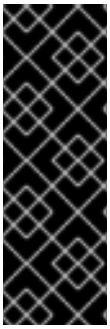
フィールド	デフォルト	値
type	**incremental**	実行するスナップショットのタイプを指定します。 タイプの設定は任意です。現在要求できるのは、**incremental** スナップショットのみです。
data-collections	**該当なし**	スナップショットを作成するテーブルの完全修飾名が含まれる配列。 名前の形式は **signal.data.collection** 設定オプションと同じです。

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



重要

増分スナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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、ストリーム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ーに提供します。スナップショットウィンドウ中に、バッファー内の **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、バッファーされた **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、バッファーに含まれるのは、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。

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。SQL **INSERT** クエリーとしてテーブルにシグナルを送信します。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、テーブルを一覧表示する **data-collections** アレイを指定します (例: **{"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。

前提条件

- シグナルが有効になっている。
  - シグナルデータコレクションがソースのデータベースに存在し、コネクターはこれをキャプチャーするように設定されています。
  - 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>_"}');
```

以下に例を示します。

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

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

以下の表では、これら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

値	説明
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。
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。
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。
data-collections	スナップショットに含めるテーブル名の配列を指定するシグナルの **data** フィールドの必須コンポーネント。 配列は、**signal.data.collection** 設定プロパティーにコネクターのシグナルテーブルの名前を指定するときに使用する形式で、完全修飾名別にテーブルを一覧表示します。
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。 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。

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。

以下に例を示します。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。
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。

5.1.5. 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。

serverName.databaseName.tableName

fulfillment はサーバー名、**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。

serverName
: **database.server.name** コネクター設定プロパティーで指定したコネクターの論理名です。

schemaName
: 操作が発生したスキーマの名前。

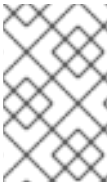
tableName
: 操作が発生したテーブルの名前。

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。

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": "0e4d5dcd-a33b-11ea-80f1-02010a22a99e:10",
  "event_count": null,
  "data_collections": null
}

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

```
    }
  ]
}
```

コネクターはトランザクションイベントを **<database.server.name>.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": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

表5.6 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ー 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ー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のようになります。JSON では、以下のようになりま

す。

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

表5.7 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**	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。 • **mysql-server-1** 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。

- **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

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": "1.7.2.Final",
      "connector": "mysql",
      "name": "mysql-server-1",
      "ts_ms": 0,
      "snapshot": false,
      "db": "inventory",
      "table": "customers",
      "server_id": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "thread": 7,
      "query": "INSERT INTO customers (first_name, last_name, email) VALUES ('Anne', 'Kretchmar',
'annek@noanswer.org')"
    }
  }
}
```


表5.8 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**	**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。 **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。 **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 スキーマには独自の進化と履歴があります。
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**	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。
6	**op**	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。 ● **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)
7	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。

項目	フィールド名	説明
8	**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。
9	**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。
10	**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。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。 ● Debezium バージョン ● コネクター名 ● 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 **binlog_rows_query_log_events** MySQL 設定オプションが有効で、コネクター設定 **include.query** プロパティーが有効な場合、**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。

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。

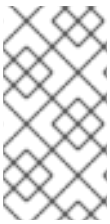
```
{
  "schema": { ... },
  "payload": {
    "before": { 1
      "id": 1004,
      "first_name": "Anne",
```

```
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": { 2
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": { 3
      "version": "1.7.2.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", 4
    "ts_ms": 1465581029523 5
  }
}
```

表5.9 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**	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。 ● Debezium バージョン ● コネクター名 ● 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 **binlog_rows_query_log_events** MySQL 設定オプションが有効で、コネクター設定 **include.query** プロパティーが有効な場合、**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。
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。
5	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。



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。

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 **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": "1.7.2.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5.10 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**	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。 ● Debezium バージョン ● コネクター名 ● 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 **binlog_rows_query_log_events** MySQL 設定オプションが有効で、コネクター設定 **include.query** プロパティーが有効な場合、**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。
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**	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。 **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。

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。

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

Debezium MySQL コネクターは、行が存在するテーブルのように構造化されたイベントで行への変更を表します。イベントには、各列の値のフィールドが含まれます。その列の MySQL データ型は、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。

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5.11 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**

MySQL 型	リテラル型	セマンティック型
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。 **numBytes = n/8 + (n%8== 0 ?0 :1)**
TINYINT	**INT16**	**該当なし**
SMALLINT[(M)]	**INT16**	**該当なし**
MEDIUMINT[(M)]	**INT32**	**該当なし**
INT, INTEGER[(M)]	**INT32**	**該当なし**
BIGINT[(M)]	**INT64**	**該当なし**
REAL[(M,D)]	**FLOAT32**	**該当なし**
FLOAT[(M,D)]	**FLOAT64**	**該当なし**
DOUBLE[(M,D)]	**FLOAT64**	**該当なし**
CHAR(M)]	**STRING**	**該当なし**
VARCHAR(M)]	**STRING**	**該当なし**
BINARY(M)]	**BYTES** または **STRING**	**該当なし** **binary.handling.mode** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、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** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、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** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、または 16 進数でエンコードされた文字列のいずれか。

MySQL 型	リテラル型	セマンティック型
TINYTEXT	**STRING**	**該当なし**
BLOB	**BYTES** または **STRING**	**該当なし** **binary.handling.mode** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、または 16 進数でエンコードされた文字列のいずれか。
TEXT	**STRING**	**該当なし**
MEDIUMBLOB	**BYTES** または **STRING**	**該当なし** **binary.handling.mode** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、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**
LONGBLOB	**BYTES** または **STRING**	**該当なし** **binary.handling.mode** コネクター設定を基にし、raw バイト (デフォルト)、base64 でエンコードされた文字列、または 16 進数でエンコードされた文字列のいずれか。
LONGTEXT	**STRING**	**該当なし**
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。
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** コネクター設定プロパティー

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 コネクターは、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含まれません。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ーのために設定) 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ーの詳細は、MySQL コネクター設定プロパティー のドキュメントを参照してください。

time.precision.mode=adaptive_time_microseconds(default)

MySQL コネクターは、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5.12 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。

MySQL 型	リテラル型	セマンティック型
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る場合のみ、**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5.13 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 コネクターは、**decimal.handling.mode** コネクター設定プロパティー の設定にしたがって 10 進数を処理します。

decimal.handling.mode=precise

表5.14 decimal.handing.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

MySQL 型	リテラル型	セマンティック型
NUMERIC[(M[,D])]	**BYTES**	**org.apache.kafka.connect.data.Decimal** **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。

decimal.handling.mode=double

表5.15 decimal.handing.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
NUMERIC[(M[,D])]	**FLOAT64**	**該当なし**
DECIMAL[(M[,D])]	**FLOAT64**	**該当なし**

decimal.handling.mode=string

表5.16 decimal.handing.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
NUMERIC[(M[,D])]	**STRING**	**該当なし**
DECIMAL[(M[,D])]	**STRING**	**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)** の両方のカラムに **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。

以下は ConfigMap の例になります。

```
converters=boolean
boolean.type=io.debezium.connector.mysql.converters.TinyIntOneToBooleanConverter
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5.17 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。 • **srid (INT32**: 構造に保存されたジオメトリーオブジェクトの型を定義する、空間参照システム ID。 • **wkb (BYTES)**: wkb (Well-Known-Binary) 形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。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

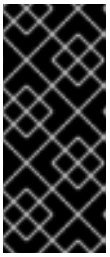
1. MySQL ユーザーを作成します。

```
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、snapshots 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5.18 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。
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。 • **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。

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

2. **OFF** の場合は、以下に説明するプロパティーで 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
```

3. 再度 binlog の状態をチェックして、変更を確認します。

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

表5.19 MySQL binlog 設定プロパティーの説明

プロパティー	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 purges binlog files 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

1. **gtid_mode** を有効にします。

```
mysql> gtid_mode=ON
```

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

```
mysql> enforce_gtid_consistency=ON
```

3. 変更を確認します。

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

結果

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

表5.20 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。 • **ON** = 有効化 • **OFF** = 無効化

オプション	説明
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。 • **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5.21 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

5.5. 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 コンテナー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ーの説明」

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 コンテナー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**コンテナーイ

メージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ー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 にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。
- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

- 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 として保存します。

- 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。

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、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

例5.1 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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-mysql
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/1.7.2.Final-redhat-<build_number>/debezium-connector-mysql-1.7.2.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.0-redhat-<build-number>.zip
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表5.22 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、クラスターオペレーターが **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ー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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメント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。

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

```
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ー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、**mysql-inventory-connector.yaml** として保

存します。

例5.2 Debezium コネクターの 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)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.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.dbname: mydatabase (9)
    database.server.name: inventory_connector_mysql (10)
    database.include.list: public.inventory (11)
```

表5.23 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 がデータベースに接続するユーザーアカウントの名前。
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。

項目	説明
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター 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 コンテナーイメージをビルドして Debezium MySQL コネクターのデプロイ

Debezium MySQL コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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 コネクターと連携するように 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 コネクターを実行するコンテナーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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 コンテナーを作成します。

   a. Debezium MySQL コネクターアーカイブをダウンロードします。

   b. Debezium MySQL コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

   c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-mysql.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) 任意のファイル名を指定できます。

(2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

   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。

   d. 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ー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ーイメージを構築します。

   e. カスタムイメージを **quay.io** などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** および **image** プロパティーを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。

```
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-mysql (2)
```

(1) KafkaConnector リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。

g. 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ーを設定する **.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コラムで 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ー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 コネクターを設定します。**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 (1)
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
```

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

表5.24 コネクター設定の説明

項目	説明
1	コネクターの名前。
2	1度に1つのタスクのみが動作する必要があります。MySQL コネクターは MySQL サーバーの **binlog** を読み取るため、単一のコネクタータスクを使用することで、順序とイベントの処理が適切に行われるようになります。Kafka Connect サービスはコネクターを使用して作業を行う1つ以上のタスクを開始し、実行中のタスクを自動的に Kafka Connect サービスのクラスター全体に分散します。いずれかのサービスが停止またはクラッシュすると、これらのタスクは稼働中のサービスに再分散されます。
3	コネクターの設定。
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ーの名前です (**mysql**)。
5	connector の一意 ID。
6	MySQL サーバーまたはクラスターの論理名。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	**inventory** データベースの変更のみがキャプチャーされます。
8	DDL ステートメントをデータベース履歴トピックに書き込み、復元するためにコネクターによって使用される Kafka ブローカーのリスト。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ーの完全リストは、MySQL コネクター設定プロパティーを参照してください。

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 コネクターは 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_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

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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

   - 端末画面で、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.connector.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":"1.7.2.Final-redhat-00001","connector":"mysql","name":"inventory_connector_mysql","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"mysql-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。

5.5.5. Debezium MySQL コネクター設定プロパティーの説明

Debezium MySQL 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ーが多数あります。多くのプロパティーにはデフォルト値があります。プロパティーに関する情報は、以下のように設定されています。

- 必要なコネクター設定プロパティー
- 高度なコネクター設定プロパティー
- 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設定プロパティー。
  - パススルーデータベースの履歴プロパティー
- 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ー。

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

表5.25 必要な Debezium MySQL コネクター設定プロパティー

プロパティー	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。このプロパティーはすべての 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。MySQL コネクターに常に **io.debezium.connector.mysql.MySqlConnector** を指定します。
tasks.max	**1**	このコネクターのために作成する必要のあるタスクの最大数。MySQL コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。
database.hostname	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。

プロパティー	デフォルト	説明
database.port	3306	MySQL データベースサーバーのポート番号 (整数)。
database.user	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。
database.server.name	デフォルトなし	Debezium が変更をキャプチャーする特定の MySQL データベースサーバー/クラスターの namespace を識別および提供する論理名。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターによって生成される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。
database.server.id	random	このデータベースクライアントの数値 ID。MySQL クラスターで現在稼働しているすべてのデータベースプロセスで一意である必要があります。このコネクターは、MySQL データベースクラスターを (この一意の ID を持つ) 別のサーバーとして結合するため、binlog を読み取ることができます。デフォルトでは、5400 から 6400 までの乱数が生成されますが、値を明示的に設定することが推奨されます。
database.include.list	空の文字列	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)。コネクターは、名前が **database.include.list** に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、コネクターはすべてのデータベースの変更をキャプチャーします。また、**database.exclude.list** コネクター設定プロパティーは設定しないでください。
database.exclude.list	空の文字列	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)。コネクターは、名前が **database.exclude.list** にないデータベースの変更をキャプチャーします。また、**database.include.list** コネクター設定プロパティーは設定しないでください。

プロパティー	デフォルト	説明
table.include.list	空の文字列	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)。コネクターは **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、コネクター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。また、**table.exclude.list** コネクター設定プロパティーは指定しないでください。
table.exclude.list	空の文字列	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 databaseName.tableName です。また、**table.include.list** コネクター設定プロパティーは指定しないでください。
column.exclude.list	空の文字列	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 databaseName.tableName.columnName です。
column.include.list	空の文字列	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 databaseName.tableName.columnName です。
column.truncate.to._length_.chars	該当なし	フィールド値が指定された文字数より長い場合に、変更イベントレコード値で値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。単一の設定で、異なる長さの複数のプロパティーを設定できます。長さは正の整数である必要があります。列の完全修飾名の形式は databaseName.tableName.columnName です。
column.mask.with._length_.chars	該当なし	変更イベントメッセージで、指定された数のアスタリスク (*) で設定されるフィールド値に値を置き換える必要のある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。単一の設定で、異なる長さの複数のプロパティーを設定できます。それぞれの長さは正の整数またはゼロである必要があります。列の完全修飾名の形式は databaseName.tableName.columnName です。

プロパティー	デフォルト	説明
column.mask.hash.hashAlgorithm.with.salt.salt	**該当なし**	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 **<databaseName>.<tableName>.<columnName>** です。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。 仮名は、指定された **hashAlgorithm** と **salt** 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、参照整合性は維持され、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 section に説明されています。 以下の例では、**CzQMA0cB5K** が無作為に選択された salt になります。 `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` 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。 使用される **hashAlgorithm**、選択された **salt**、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。

プロパティー	デフォルト	説明
column.propagate.source.type	該当なし	出力された変更イベントレコードの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。以下のスキーマパラメーターは、それぞれ可変幅型の元の型名および長さを伝達するために使用されます。 **__Debezium.source.column.type** **__Debezium.source.column.length** **__Debezium.source.column.scale** それぞれ元の型名と長さ (可変幅型の場合) を伝達するために使用されます。これは、シンクデータベースの対応する列を適切にサイズ調整するのに便利です。列の完全修飾名の形式は以下のいずれかになります。 databaseName.tableName.columnName databaseName.schemaName.tableName.columnName
datatype.propagate.source.type	該当なし	出力された変更イベントレコードの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列のデータベース固有のデータ型名と一致する、正規表現のコンマ区切りリスト (任意)。以下のスキーマパラメーターは、それぞれ可変幅型の元の型名および長さを伝達するために使用されます。 **__debezium.source.column.type** **__debezium.source.column.length** **__debezium.source.column.scale** それぞれ元の型名と長さ (可変幅型の場合) を伝達するために使用されます。これは、シンクデータベースの対応する列を適切にサイズ調整するのに便利です。完全修飾データ型名の形式は以下のいずれかになります。 databaseName.tableName.typeName databaseName.schemaName.tableName.typeName MySQL 固有のデータ型名のリストは、MySQL コネクターによるデータ型のマッピング方法 を参照してください。

プロパティー	デフォルト	説明
time.precision.mode	**adaptive_time_microseconds**	時間、日付、およびタイムスタンプは、以下を含む異なる精度の種類で表すことができます。 **adaptive_time_microseconds** (デフォルト) は、データベース列の型を基にして、ミリ秒、マイクロ秒、またはナノ秒の精度値のいずれかを使用して、データベースの値と全く同じように日付、日時、およびタイムスタンプ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。
decimal.handling.mode	**precise**	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。 **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。
bigint.unsigned.handling.mode	**long**	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: **long** 値は、Java の **long** を使用して値を表します。これは、コンシューマーで簡単に使用できますが、通常 **long** が推奨の設定です。 **precise** は **java.math.BigDecimal** を使用して値を表し、バイナリー表現と Kafka Connect の **org.apache.kafka.connect.data.Decimal** タイプを使用して変更イベントでエンコードされます。2^63 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。

プロパティー	デフォルト	説明
include.schema.changes	**true**	コネクターがデータベーススキーマの変更を、データベースサーバー ID と同じ名前の Kafka トピックに公開するかどうかを指定するブール値。各スキーマの変更はデータベース名が含まれるキーを使用して記録され、その値には DDL ステートメントが含まれます。これは、コネクターがデータベース履歴を内部で記録する方法には依存しません。
include.query	**false**	変更イベントを生成した元の SQL クエリーがコネクターに含まれる必要があるかどうかを指定するブール値。 このオプションを **true** に設定した場合は、MySQL の **binlog_rows_query_log_events** オプションを **ON** に設定する必要があります。**include.query** 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。 **include.query** 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。
event.deserialization.failure.handling.mode	**fail**	binlog イベントのデシリアライズ中にコネクターがどのように例外に反応するかを指定します。 **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。 **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。 **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。

プロパティー	デフォルト	説明
inconsistent.schema.handling.mode	fail	内部スキーマ表現に存在しないテーブルに関連する binlog イベント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性がありません。 **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。 **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。 **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。
max.queue.size	8192	データベースログから読み取られた変更イベントが Kafka に書き込まれる前に配置される、ブロッキングキューの最大サイズを指定する正の整数値。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに、binlog リーダーにバックプレシャーを提供できます。キューに発生するイベントは、このコネクターによって定期的に記録されるオフセットには含まれません。デフォルトは 8192 で、**max.batch.size** プロパティーで指定される最大バッチサイズよりも大きな値を常に指定する必要があります。
max.batch.size	2048	このコネクターの反復処理中に処理される必要があるイベントの各バッチの最大サイズを指定する正の整数値。デフォルトは 2048 です。
max.queue.size.in.bytes	0	ブロッキングキューの最大サイズ (バイト単位) の long 値。この機能はデフォルトで無効になっています。正の long 値が設定されると有効になります。
poll.interval.ms	1000	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 1000 ミリ秒 (1 秒) です。
connect.timeout.ms	30000	コネクターが MySQL データベースサーバーへの接続を試行した後、タイムアウトするまでの最大の待機期間をミリ秒単位で指定する正の整数値。デフォルトは 30 秒です。

プロパティー	デフォルト	説明
gtid.source.includes	デフォルトなし	MySQL サーバーで binlog の位置を見つけるために使用される GTID セットのソース UUID に一致する、正規表現のコンマ区切りリスト。これらの include パターンのいずれかに一致するソースを持つ GTID の範囲のみが使用されます。**gtid.source.excludes** の設定は指定しないでください。
gtid.source.excludes	デフォルトなし	MySQL サーバーで binlog の位置を見つけるために使用される GTID セットのソース UUID に一致する、正規表現のコンマ区切りリスト。これらすべての exclude パターンに一致しないソースを持つ GTID の範囲のみが使用されます。また、**gtid.source.includes** の値も指定しないでください。
tombstones.on.delete	**true**	廃棄 (tombstone) イベントの後に**削除**イベントが続くかどうかを制御します。 **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。 **false** - delete イベントのみが出力されます。 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。

プロパティー	デフォルト	説明
message.key.columns	該当なし	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。 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。 テーブルのカスタムメッセージキーを作成するには、テーブルとメッセージキーとして使用する列をリストします。各リストエントリーは以下の形式をとります。 **<fully-qualified_tableName>:_<keyColumn>_,<keyColumn>** 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。 **<databaseName>.<tableName>** プロパティーには複数のテーブルのエントリー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ーを区切ります。 以下の例では、テーブル **inventory.customers** と **purchase.orders** にメッセージキーを設定しています。 **inventory.customers:pk1,pk2;(.*).purchaseorders:pk3,pk4** テーブル **inventory.customer** では、列 **pk1** と **pk2** がメッセージキーとして指定されています。データベースで **purchaseorders** テーブルは、**pk3** および **pk4** サーバーのコラムをメッセージキーとして使用します。 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。

プロパティー	デフォルト	説明
binary.handling.mode	bytes	バイナリー列 (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: **bytes** はバイナリーデータをバイト配列として表します。 **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。 **hex** は、バイナリーデータを 16 進数でエンコードされた (base16) 文字列として表します。

高度な MySQL コネクター設定プロパティー

以下の表は、高度な MySQL コネクタープロパティー について説明しています。これらのプロパティー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5.26 MySQL コネクターの高度な設定プロパティーの説明

プロパティー	デフォルト	説明
connect.keep.alive	**true**	MySQL サーバー/クラスターへの接続を確実に維持するために、別のスレッドを使用するかどうかを指定するブール値。
table.ignore.builtin	**true**	組み込みシステムテーブルを無視するかどうかを指定するブール値。これは、テーブルの include および exclude 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。

プロパティー	デフォルト	説明
database.ssl.mode	**disabled**	暗号化された接続を使用するかどうかを指定します。可能な設定: **disabled** は暗号化されていない接続の使用を指定します。 **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。 **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。 **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。 **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。

プロパティー	デフォルト	説明
snapshot.mode	Initial	コネクターの起動時にスナップショットを実行するための基準を指定します。可能な設定は次のとおりです。 **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。 **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。 **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。 **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。 **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。 **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。データベース履歴トピックは無期限に保持する必要があります。

プロパティー	デフォルト	説明
snapshot.locking.mode	**最小**	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: **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。 **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。 **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。 **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ーが設定されていても、テーブルはロックされます。この動作は、行レベルのロックを取得する InnoDB エンジンの動作とは異なります。

プロパティー	デフォルト	説明
snapshot.include.collection.list	**table.include.list**に指定したすべてのテーブル	スナップショットに含めるテーブルの完全修飾名 (**<databaseName>.<tableName>**) 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、コネクターの table.include.list プロパティーで名前を付ける必要があります。このプロパティーは、コネクターの **snapshot.mode** プロパティーが **never** 以外の値に設定されている場合にのみ有効になります。 このプロパティーは増分スナップショットの動作には影響しません。

プロパティー	デフォルト	説明
snapshot.select.statement.overrides	デフォルトなし	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、プロパティーを使用します。このプロパティー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。 **<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、 **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** 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ーを追加して、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ーの名前を指定します。 **snapshot.select.statement.overrides.<databaseName>.<tableName>**例: **snapshot.select.statement.overrides.customers.orders**. 例: 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、以下のプロパティーを追加します。 "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" 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。

プロパティー	デフォルト	説明
min.row.count.to.stream.results	1000	スナップショットの実行中、コネクターは変更をキャプチャーするように設定されている各テーブルにクエリーを実行します。コネクターは各クエリーの結果を使用して、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。このプロパティーは、MySQL コネクターがテーブルの結果をメモリーに格納するか、またはストリーミングを行うかを決定します。メモリーへの格納はすばやく処理できますが、大量のメモリーを必要とします。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。このプロパティーの設定は、コネクター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。 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、このプロパティーを **0** に設定します。
heartbeat.interval.ms	0	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。 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、このプロパティーを、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。
heartbeat.topics.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次のようになります。 heartbeat.topics.prefix.server.name たとえば、データベースサーバー名が **fulfillment** の場合、デフォルトのトピック名は **__debezium-heartbeat.fulfillment** になります。

プロパティー	デフォルト	説明
database.initial.statements	デフォルトなし	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。 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、このプロパティー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。
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、このプロパティーは、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	デフォルトなし	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。このプロパティーは、バッチの行の最大数を指定します。
snapshot.lock.timeout.ms	**10000**	スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する正の整数。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。
enable.time.adjuster	**true**	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。 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。

プロパティー	デフォルト	説明
sanitize.field.names	コネクターが **key.converter** または **value.converter** プロパティーを 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	デフォルトなし	ストリーミング中にスキップする操作タイプのコンマ区切りリスト。以下の値を使用できます (**c** は挿入/作成、**u** は更新、**d** は削除)。デフォルトでは、操作はスキップされません。
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。 以下の形式を使用してコレクション名を指定します。 **<databaseName>.<tableName>** シグナル機能はテクノロジープレビュー機能です。
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。 増分スナップショットはテクノロジープレビュー機能です。
read.only	**false**	シグナルデータコレクションへの書き込みを回避するために、別の増分スナップショットのウォーターマーク実装に切り替えます。
provide.transaction.metadata	**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、Transaction metadata を参照してください。

Debezium コネクターデータベース履歴設定プロパティー

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 **database.history.*** プロパティー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 **database.history** プロパティーについて説明しています。

表5.27 コネクターデータベース履歴設定プロパティー

プロパティー	デフォルト	説明
database.history.kafka.topic		コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。
database.history.kafka.bootstrap.servers		Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。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。
database.history.kafka.recovery.poll.interval.ms	**100**	永続化されたデータのポーリングが行われている間にコネクターが起動/回復を待つ最大時間 (ミリ秒単位) を指定する整数値。デフォルトは 100 ミリ秒です。
database.history.kafka.recovery.attempts	**4**	エラーでコネクターのリカバリーが失敗する前に、コネクターが永続化された履歴データの読み取りを試行する最大回数。データが受信されなかった場合に最大待機する時間は、**recovery.attempts** x **recovery.poll.interval.ms** です。
database.history.skip.unparseable.ddl	**false**	コネクターが不正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。
database.history.store.only.monitored.tables.ddl **今後のリリースで非推奨になり、削除される予定です。代わりに database.history.store.only.captured.tables.ddl を使用してください。**	**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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 安全なデフォルトは **false** です。

プロパティー	デフォルト	説明
database.history.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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 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ー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履歴プロパティー

Debezium は、Kafka プロデューサーを使用して、データベース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。**database.history.producer.*** および **database.history.consumer.*** 接頭辞で始まるパススルー設定プロパティー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ー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ーデータベース履歴プロパティーは、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。

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

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

Debezium は、プロパティーを Kafka クライアントに渡す前に、プロパティー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ー および Kafka コンシューマー設定プロパティーの詳細は、Kafka のドキュメントを参照してください。

Debezium コネクター Kafka は設定プロパティー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.*** プロパティーのセットを提供します。

以下の表は **signal** プロパティーについて説明しています。

表5.28 Kafka のシグナル設定プロパティー

プロパティー	デフォルト	説明
signal.kafka.topic		コネクター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		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。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ー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 コネクターのパススルーは Kafka コンシューマークライアント設定プロパティーを示唆します。

Debezium コネクターでは、Kafka コンシューマー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ーは、接頭辞 **signals.consumer.*** で始まります。たとえば、コネクターは **signal.consumer.security.protocol=SSL** などのプロパティーを Kafka コンシューマーに渡します。

データベース履歴クライアントのパススループロパティー の場合のように、Debezium はプロパティーから接頭辞を削除してから Kafka シグナルコンシューマーに渡します。

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ー

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ーは、接頭辞 **database.*** で始まります。たとえば、コネクターは **database.foobar=false** などのプロパティーを JDBC URL に渡します。

データベース履歴クライアントのパススループロパティー の場合のように、Debezium はプロパティー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 コネクターのパフォーマンスの監視

Debezium MySQL コネクターは、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ー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、コネクターが binlog を読み取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 モニター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。

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	型	説明
LastEvent	string	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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。

属性	型	説明
SnapshotRunning	boolean	スナップショットが起動されたかどうか。
SnapshotAborted	boolean	スナップショットが中断されたかどうか。
SnapshotCompleted	boolean	スナップショットが完了したかどうか。
SnapshotDurationInSeconds	long	スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。
RowsScanned	Map<String, Long>	スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

コネクターは、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	型	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。

属性	型	説明
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。

重要

増分スナップショットの使用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 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 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ーが有効になっている場合にのみ利用できます。詳細は、高度な MySQL コネクター設定プロパティーの **binlog.buffer.size** を参照してください。

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	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。

属性	型	説明
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables 非推奨、今後のリリースで削除される予定ですので、代わりに 'CapturedTables' メトリクスを使用してください。	string[]	コネクターによって監視されるテーブルの一覧。
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。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。

属性	型	説明
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (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5.29 追加のストリーミングメトリクスの説明

属性	型	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。
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 コネクター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	型	説明
Status	string	データベース履歴の状態を示す **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。

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 プロセスがクラッシュする

- 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 クラスターで 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。これはクラスターの別の 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 プロセスがクラッシュする

Kafka Connect がクラッシュすると、プロセスが停止し、最後に処理されたオフセットが記録されずに Debezium MySQL コネクタータスクが終了します。分散モードでは、Kafka Connect は他のプロセスでコネクタータスクを再起動します。ただし、MySQL コネクターは以前のプロセスで記録された最後のオフセットから再開します。つまり、代替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 コネクターが長時間停止すると、MySQL サーバーは古い binlog ファイルをパージするため、コネクターの最後の位置が失われる可能性があります。コネクターが再起動すると、MySQL サーバーに開始点がなくなり、コネクターは別の最初のスナップショットを実行します。スナップショットが無効の場合、コネクターはエラーによって失敗します。

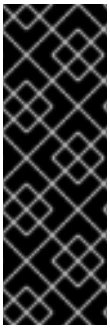
MySQL コネクターが最初のスナップショットを実行する方法に関する詳細は、Debezium MySQL コネクターによるデータベーススナップショットの実行方法 を参照してください。

第6章 ORACLE 向けの DEBEZIUM コネクター (テクノロジープレビュー)

Debezium の Oracle コネクターは、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、コネクターの実行中に追加されたテーブルが含まれます。コネクターを設定して、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、または切り捨てしたりするように設定できます。

このコネクターと互換性のある 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。



重要

Debezium Oracle はテクノロジープレビューの機能です。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 Oracle コネクター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 コネクターの仕組み」
- 「Debezium Oracle コネクターのデータ変更イベントの説明」
- 「Debezium Oracle コネクター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 コネクターのデプロイメント」
- 「Debezium Oracle コネクターのパフォーマンスの監視」
- 「Debezium Oracle コネクターによる障害および問題の処理方法」

6.1. DEBEZIUM ORACLE コネクターの仕組み

Debezium Oracle コネクターを最適に設定および実行するには、コネクター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 Oracle コネクターによるデータベーススナップショットの実行方法」
- 「Debezium 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 コネクター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 コネクターによって生成されたイベント」

6.1.1. Debezium Oracle 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

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 コネクターはログからデータベースの履歴全体を取得できません。コネクター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、コネクターの初回起動時に、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 コネクター設定プロパティーの値を設定することで、コネクター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、コネクター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ーワークフロー

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、コネクター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監視されているテーブルごとに **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ーします。
5. ステップ 2 で取得したロックを解放します。
6. 手順 3 で読み取った SCN 位置で、関連するすべてのデータベース表およびスキーマを有効としてスキャン (**SELECT * FROM ... AS OF SCN 123**) して、行ごとに **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. コネクターオフセットにスナップショットの正常な完了を記録します。

スナップショットプロセスが開始されたら、コネクターの障害、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、コネクターの再起動後にプロセスが再起動されます。コネクター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ーが再び停止した場合に、コネクター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6.1snapshot.mode コネクター設定プロパティーの設定

設定	説明
Initial	コネクターは、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、コネクターは、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。
schema_only	コネクターは関連するすべてのテーブルの構造をキャプチャーし、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、コネクターの起動時 (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。

6.1.1.1. 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の使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。通常の状況では、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、コネクター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。

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

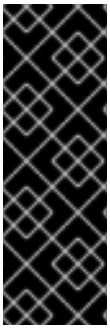
フィールド	デフォルト	値
type	**incremental**	実行するスナップショットのタイプを指定します。 タイプの設定は任意です。現在要求できるのは、**incremental** スナップショットのみです。
data-collections	**該当なし**	スナップショットを作成するテーブルの完全修飾名が含まれる配列。 名前の形式は **signal.data.collection** 設定オプションと同じです。

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



重要

増分スナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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、ストリーム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ーに提供します。スナップショットウィンドウ中に、バッファー内の **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、バッファーされた **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、バッファーに含まれるのは、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。

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。SQL **INSERT** クエリーとしてテーブルにシグナルを送信します。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、テーブルを一覧表示する **data-collections** アレイを指定します (例: **{"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。

前提条件

- シグナルが有効になっている。
  - シグナルデータコレクションがソースのデータベースに存在し、コネクターはこれをキャプチャーするように設定されています。
  - 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>_"}');
```

以下に例を示します。

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

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

以下の表では、これら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

値	説明
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。
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。

値	説明
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。
data-collections	スナップショットに含めるテーブル名の配列を指定するシグナルの **data** フィールドの必須コンポーネント。 配列は、**signal.data.collection** 設定プロパティーにコネクターのシグナルテーブルの名前を指定するときに使用する形式で、完全修飾名別にテーブルを一覧表示します。
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。 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。

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。

以下に例を示します。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。
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。

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。

6.1.2. 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。

serverName.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。

serverName
: **database.server.name** コネクター設定プロパティーで指定したコネクターの論理名です。

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 コネクターは、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
```

コネクターは同様の命名規則を適用して、内部データベース履歴トピック (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 コネクターによるデータベーススキーマの変更の公開方法

Debezium Oracle コネクターを設定すると、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。コネクターは、スキーマ変更イベントをすべて **<serverName>** という名前の Kafka トピックに書き込みます。 **serverName** は **database.server.name** 設定プロパティーに指定された論理サーバー名になります。

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。

コネクター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

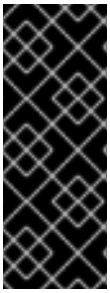
スキーマの変更につながる SQL **CREATE**、**ALTER**、または **DROP** ステートメントを提供します。

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。

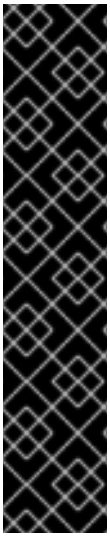
tableChanges

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ー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

コネクター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、内部データベース履歴トピックにも格納されます。内部データベース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。

以下に例を示します。Oracle コネクター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": "1.7.2.Final",
      "connector": "oracle",
      "name": "server1",
```

```
    "ts_ms": 1588252618953,
    "snapshot": "true",
    "db": "ORCLPDB1",
    "schema": "DEBEZIUM",
    "table": "CUSTOMERS",
    "txId" : null,
    "scn" : "1513734",
    "commit_scn": "1513734",
    "lcr_position" : null
  },
  "databaseName": "ORCLPDB1", (1)
  "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\" ", (2)
  "tableChanges": [ (3)
    {
      "type": "CREATE", (4)
      "id": "\"ORCLPDB1\".\"DEBEZIUM\".\"CUSTOMERS\"", (5)
      "table": { (6)
        "defaultCharsetName": null,
        "primaryKeyColumnNames": [ (7)
          "ID"
        ],
        "columns": [ (8)
          {
            "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
          }
        ]
      }
    }
  ]
 }
}
```

表6.4 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	**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。
2	**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。
3	**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。

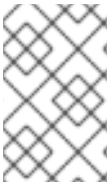
項目	フィールド名	説明
4	**type**	変更の種類を説明します。値は以下のいずれかになります。 **CREATE** テーブルの作成 **ALTER** テーブルの変更 **DROP** テーブルの削除
5	**id**	作成、変更、または破棄されたテーブルの完全な識別子。テーブルの名前が変更されると、この識別子は**<old>,<new>** のテーブル名が連結されます。
6	**table**	適用された変更後のテーブルメタデータを表します。
7	**primaryKeyColumnNames**	テーブルのプライマリーキーを設定する列のリスト。
8	**列**	変更されたテーブルの各列のメタデータ。

コネクター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.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。

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。

以下に例を示します。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",
  "event_count": null,
  "data_collections": null
}

{
  "status": "END",
  "id": "5.6.641",
  "event_count": 2,
  "data_collections": [
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.CUSTOMER",
      "event_count": 1
    },
    {
      "data_collection": "ORCLPDB1.DEBEZIUM.ORDER",
      "event_count": 1
    }
  ]
}
```

コネクターはトランザクションイベントを **<database.server.name>.transaction** トピックに出力します。

6.1.4.1. 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"
  }
}
```

イベントバッファー

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ットアンドされます。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ー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ーから 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。

プロパティー **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ー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ーイベントレコードを割り当て、管理します。**memory** バッファー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。

6.1.5. Oracle SCN 値間のギャップ

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、コネクターが変更をほぼリアルタイムでストリーミングできるかどうかに応じて範囲を増減するか、データベースのサイズが大きいトランザクションまたは一括トランザクションのためにバックログを処理する必要があります。

特定の状況では、Oracle データベースは一定のレートで増加するのではなく、通常は高い量でシステムの変更番号を事前に設定します。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ーに依存して 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。

コネクターは最初に、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。この差が最小ギャップサイズよりも大きい場合、コネクターは SCN ギャップが検出されている可能性があります。ギャップが存在するかどうかを確認するために、コネクター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 コネクターは現在のマイマイ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、コネクター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。さらに、コネクターはこれが発生する場合にのみ、この反復の最大バッチサイズを無視します。

警告

SCN ギャップ検出は、コネクターが動作していて、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。

6.2. DEBEZIUM ORACLE コネクターのデータ変更イベントの説明

Oracle コネクター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。コネクター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、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、コネクター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 コネクター変更イベントのキー」
- 「Debezium Oracle 変更イベントの値」

6.2.1. Debezium Oracle コネクター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
);
```

-

<database.server.name>.transaction 設定プロパティーの値が **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

message キーと同様に、変更イベントメッセージの値には **schema** セクションと **payload** セクションがあります。Oracle コネクターによって生成されたすべての変更イベント値の payload セクションは、以下のフィールドを含む **envelope** 構造となっています。

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 コネクターの値は、**c** (作成または挿入)、**u** (更新)、**d** (削除)、および **r** (読み取り、初回のスナップショットでない場合) です。

before

任意のフィールド。存在する場合は、イベント発生**前**の行の状態が含まれます。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

任意のフィールド。存在する場合は、イベント発生**後**の行の状態が含まれます。この構造は、**before** で使用されるのと同じ **server1.INVENTORY.CUSTOMERS.Value** Kafka Connect スキーマで記述されます。

source

Oracle に、Debezium バージョン、コネクター名、イベントが作成中のスナップショットかどうか、トランザクション ID (スナップショット作成中ではない)、変更の SCN、ソースとなるデータベースでレコードが変更された時点を表すタイムスタンプ (スナップショット中、スナップショット

の作成時点) といったフィールドが含まれる場合に、イベントのソースメタデータを記述する構造を含む必須のフィールドです。

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。このフィールドは、LogMiner 接続ファクトリーを使用している場合にのみ表示されます。

ts_ms

任意のフィールド。存在する場合は、コネクターがイベントを処理した時間 (Kafka Connect タスクを実行する JVM のシステムクロックを使用) が含まれます。

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

作成 イベント

customers テーブルの **create** イベント値を見てみましょう。


```
{
    "schema": {
        "type": "struct",
        "fields": [
            {
                "type": "struct",
                "fields": [
                    {
                        "type": "int32",
                        "optional": false,
                        "field": "ID"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "FIRST_NAME"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "LAST_NAME"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "EMAIL"
                    }
                ],
                "optional": true,
                "name": "server1.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": "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": "1.7.2.Final",
        "name": "server1",
        "ts_ms": 1520085154000,
        "txId": "6.28.807",
        "scn": "2122185",
        "commit_scn": "2122185",
        "snapshot": false
    },
    "op": "c",
    "ts_ms": 1532592105975
}
}
```


前述のイベントの **value** の **schema** 部分を調べると、以下のスキーマが定義されている方法を確認できます。

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

このイベントの **値** の **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。JSON 表現にはメッセージの**スキーマ** 部分と **ペイロード** 部分を含める必要があるため、これは True です。Avro コンバーター を使用すると、コネクターが Kafka トピックに記述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。

更新イベント

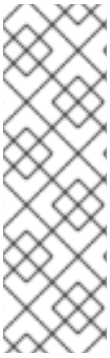
このテーブル **更新** 変更イベントの値には、**作成** イベントと同じ **スキーマ** があります。ペイロードは同じ構造を使用しますが、保持する値が異なります。以下に例を示します。

```
{
  "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": "1.7.2.Final",
      "name": "server1",
      "ts_ms": 1520085811000,
      "txId": "6.9.809",
      "scn": "2125544",
      "commit_scn": "2125544",
      "snapshot": false
    },
    "op": "u",
    "ts_ms": 1532592713485
  }
}
```

更新 イベントの値を **作成** (insert) イベントと比較して、**payload** セクションの以下の相違点に注意してください。

- **op** フィールドの値は **u** になっており、更新によってこの行が変更されたことを示しています。
- **before** フィールドは、データベースのコミット前の行と値の状態を表しています。
- **after** フィールドの行の状態が更新され、ここで **EMAIL** の値が **anne@example.com** であることを確認することができます。
- **source** フィールドの構造には以前と同じフィールドがありますが、このイベントは redo ログの位置とは異なるため、値は異なります。
- **ts_ms** は、Debezium がこのイベントを処理したタイムスタンプを示します。

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。

削除 イベント

ここまでで、**create** と **update** イベントのサンプルを確認しました。次に、同じテーブルの **削除** イベントの値を見てみましょう。**create** と **update** イベントと同様に、**delete** イベントの場合は、値の **schema** 部分は全く同じです。

```
{
    "schema": { ... },
    "payload": {
        "before": {
            "ID": 1004,
            "FIRST_NAME": "Anne",
            "LAST_NAME": "Kretchmar",
            "EMAIL": "anne@example.com"
        },
        "after": null,
        "source": {
            "version": "1.7.2.Final",
            "name": "server1",
            "ts_ms": 1520085153000,
            "txId": "6.28.807",
            "scn": "2122184",
            "commit_scn": "2122184",
            "snapshot": false
        },
        "op": "d",
```

```
        "ts_ms": 1532592105960
    }
}
```

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

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

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

Oracle コネクターのイベントは、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。

行が削除された場合でも、Kafka は同じキーを持つ以前のメッセージをすべて削除できるため、上記の**削除** イベントの値はログコンパクションで動作します。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 コネクターは、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。コネクタープロパティー **tombstones.on.delete** を設定すると、デフォルトの動作を変更できます。

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

テーブル行で発生する変更を表すために、Debezium Oracle 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変更イベントを出力します。イベントには、各列の値のフィールドが含まれます。列の値は、この列の Oracle のデータ型に従って表示されます。以下のセクションでは、Oracle 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。

リテラル型

Kafka Connect スキーマ型を使用して、値をリテラルで表す方法を記述します。**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。

詳細は以下を参照してください。

- 文字タイプ
- 数字型
- 10 進数型

- Temporal (一時) 型

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6.5 Oracle 基本文字型のマッピング

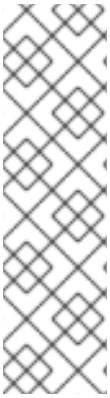
Oracle データ型	リテラル型 (スキーマ型)	セマンティック型 (スキーマ名) および注記
CHAR[(M)]	**STRING**	該当なし
NCHAR[(M)]	**STRING**	該当なし
NVARCHAR2[(M)]	**STRING**	該当なし
VARCHAR[(M)]	**STRING**	該当なし
VARCHAR2[(M)]	**STRING**	該当なし

バイナリーおよび文字の LOB 型

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6.6 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
BLOB	**BYTES**	raw バイト。
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。**
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** データタイプのカラム値を、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。つまり、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なく、コネクターのプロパティーである**unavailable.value.placeholder** で定義されるプレースホルダが含まれます。

CLOB、**NCLOB**、または **BLOB** カラムの値が更新された場合、新しい値は対応する update change イベントの **after** 部分に含まれ、利用できない値のプレースホルダーは **before** 部分に使用されます。

数字型

以下の表は、コネクターによる数字型へのマッピング方法を説明しています。

表6.7 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
DECIMAL[(P, S)]	**BYTES** / **INT8** / **INT16** / **INT32** / **INT64**	**org.apache.kafka.connect.data.Decimal BYTES** を使用する場合: **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)。
DOUBLE PRECISION	**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。
FLOAT[(P)]	**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。
INTEGER, **INT**	**BYTES**	**org.apache.kafka.connect.data.Decimal** **INTEGER** 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。
NUMBER[(P[, *])]	**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。

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**	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。 以下のように、精度とスケーリングに応じて、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 - 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**).
NUMBER(P, S > 0)	**BYTES**	**org.apache.kafka.connect.data.Decimal**
NUMERIC[(P, S)]	**BYTES** / **INT8** / **INT16** / **INT32** / **INT64**	**org.apache.kafka.connect.data.Decimal BYTES** を使用する場合: **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)。
SMALLINT	**BYTES**	**org.apache.kafka.connect.data.Decimal** **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。
REAL	**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。

ブール値型

Oracle ではネイティブで **BOOLEAN** データ型をサポートしていませが、特定のセマンティクスで他のデータ型を使用して、論理的に **BOOLEAN** データ型の概念をシミュレートするのが一般的です。

この演算子は、同梱の **NumberOneToBooleanConverter** カスタムコンバーターを設定できます。すべての **NUMBER(1)** 列を **BOOLEAN** にマッピングするか、**selector** パラメーターが設定されている場合には、コンマ区切りの正規表現を使用して、列のサブセットを列挙します。

以下は ConfigMap の例になります。

```
converters=boolean
boolean.type=io.debezium.connector.oracle.converters.NumberOneToBooleanConverter
boolean.selector=.*MYTABLE.FLAG,.*.IS_ARCHIVED
```

-

10 進数型

Oracle 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 プロパティーが **precise** に設定されている場合には コネクターは**DECIMAL** と **NUMERIC** 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。

ただし、**decimal.handling.mode** プロパティーが **double** に設定されている場合には、コネクターはスキーマタイプが **FLOAT64** の Java の double 値として表します。

string オプションを使用するように **decimal.handling.mode** 設定プロパティーを設定することもできます。プロパティーが **string** に設定されている場合には、コネクターは **DECIMAL** と **NUMERIC** の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。

Temporal (一時) 型

Oracle の **INTERVAL**、**TIMESTAMP WITH TIME ZONE**、および **TIMESTAMP WITH LOCAL TIME ZONE** データ型を除き、他の一時型は **time.precision.mode** 設定プロパティーの値により異なります。

time.precision.mode 設定プロパティーが **adaptive** (デフォルト) に設定された場合、コネクターは列のデータ型を基に時間型のリテラルおよびセマンティック型を決定し、イベントが**正確**にデータベースの値を表すようにします。

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**	**io.debezium.time.Timestamp** エポック以降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。
INTERVAL DAY[(M)] TO SECOND	**FLOAT64**	**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 YEAR[(M)] TO MONTH	**FLOAT64**	**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)。
TIMESTAMP(0 - 3)	**INT64**	**io.debezium.time.Timestamp** エポック以降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, TIMESTAMP(4 - 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。

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(7 - 9)	**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。
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 設定プロパティーが **connect** に設定された場合、コネクターは事前定義された Kafka Connect の論理型を使用します。これは、コンシューマーが組み込みの Kafka Connect の論理型のみを認識し、可変精度の時間値を処理できない場合に便利です。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** エポックからの日数を表します。
INTERVAL DAY[(M)] TO SECOND	**FLOAT64**	**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 YEAR[(M)] TO MONTH	**FLOAT64**	**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)。
TIMESTAMP(0 - 3)	**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。
TIMESTAMP(4 - 6)	**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。

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(7 - 9)	**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。
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。

6.4. DEBEZIUM と連携させるための ORACLE の設定

Debezium Oracle コネクターと連携するように 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、コンテナー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。

Vagrant を使用して仮想マシンに Oracle を設定する方法は、Debezium Vagrant Box for Oracle database GitHub リポジトリーを参照してください。

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

- 「Debezium で使用する Oracle データベースの準備」
- 「ログサイジングのやり直し」
- 「Debezium Oracle コネクターの Oracle ユーザーの作成」

6.4.1. 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;
```

さらに、変更したデータベースの行の **before** の状態をデータ変更でキャプチャーされるように、キャプチャーしたテーブルとデータベース向けに、補完用のロギングを有効にしておく必要があります。以下は、特定のテーブルでこの設定を行う方法を示しています。これは、Oracle の redo ログでキャプチャーする情報量を最小限に抑えるのに適しています。

```
ALTER TABLE inventory.customers ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
```

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、以下のように設定できます。

```
ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
```

6.4.2. ログサイジングのやり直し

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 コネクターを設定する前に、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カラム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 コネクター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。

6.4.3. Debezium Oracle コネクターの Oracle ユーザーの作成

Debezium Oracle コネクター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ーの 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

コネクターは、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。

コネクターの 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;
GRANT SET CONTAINER TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$DATABASE to c##dbzuser CONTAINER=ALL;
GRANT FLASHBACK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE_CATALOG_ROLE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ANY TRANSACTION TO c##dbzuser CONTAINER=ALL;
GRANT LOGMINING TO c##dbzuser CONTAINER=ALL;

GRANT CREATE TABLE TO c##dbzuser CONTAINER=ALL;
GRANT LOCK ANY TABLE TO c##dbzuser CONTAINER=ALL;
GRANT CREATE SEQUENCE TO c##dbzuser CONTAINER=ALL;

GRANT EXECUTE ON DBMS_LOGMNR TO c##dbzuser CONTAINER=ALL;
GRANT EXECUTE ON DBMS_LOGMNR_D TO c##dbzuser CONTAINER=ALL;

GRANT SELECT ON V_$LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOG_HISTORY TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_LOGS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_CONTENTS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGMNR_PARAMETERS TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$LOGFILE TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVED_LOG TO c##dbzuser CONTAINER=ALL;
GRANT SELECT ON V_$ARCHIVE_DEST_STATUS TO c##dbzuser CONTAINER=ALL;

exit;
```

6.5. DEBEZIUM ORACLE 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 Oracle 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 コンテナーイメージをビルドします。

Debezium Oracle コネクターでは、Oracle データベースに接続するために Oracle JDBC ドライバー (ojdbc8.jar) が必要です。ドライバーの入手方法については、Oracle JDBC ドライバーの入手 を参照してください。

関連情報

- 「Debezium Oracle コネクター設定プロパティーの説明」

6.5.1. AMQ Streams を使用した Debezium Oracle コネクターのデプロイメント

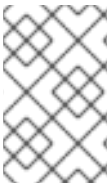
Debezium 1.7 以降、Debezium コネクターのデプロイに推奨される方法は、AMQ Streams を使用してコネクタープラグインが含まれる Kafka Connect コンテナーイメージをビルドすることです。

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**コンテナーイメージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ーイメージの自動作成と OpenShift での AMQ Streams のアップグレード

6.5.2. AMQ Streams を使用した Debezium Oracle 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 にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。

- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。
- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

  - 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 として保存します。

  - 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。

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、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

   例6.1 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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  version: 3.00
  build: 2
    output: 3
      type: imagestream 4
      image: debezium-streams-connect:latest
    plugins: 5
      - name: debezium-connector-oracle
        artifacts:
          - type: zip 6
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-
```

```
oracle/1.7.2.Final-redhat-<build_number>/debezium-connector-oracle-1.7.2.Final-
redhat-<build_number>-plugin.zip 7
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-
connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-
2.0-redhat-<build-number>.zip
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表6.8 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、クラスターオペレーターが **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ー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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメント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。

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

```
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ー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.MySqlConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: oracle.debezium-oracle.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_oracle (10)
    database.include.list: public.inventory (11)
```

表6.9 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。

項目	説明
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 がデータベースに接続するユーザーアカウントの名前。
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター 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。

6.5.3. Dockerfile からカスタム Kafka Connect コンテナーイメージをビルドして Debezium Oracle コネクターのデプロイ

Debezium Oracle コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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 コネクターと連携する 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ーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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。
- Oracle JDBC ドライバーのコピーがある。ライセンス要件により、Debezium Oracle コネクターには、必要とされるドライバーファイルは含まれません。
  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 コンテナーを作成します。
   a. Debezium Oracle コネクターアーカイブ をダウンロードします。
   b. Debezium Oracle コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

   c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-oracle.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) 任意のファイル名を指定できます。

(2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

   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。
   d. 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ー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ーイメージを構築します。

e. カスタムイメージを quay.io などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** と **image** プロパティーを指定する **dbz-connect.yaml** という名前の KafkaConnect CR を作成します。

```
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-oracle 2
```

1. KafkaConnector リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。

g. 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 Oracle 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、コネクターに使用できる設定プロパティーを使用して、**.yaml** ファイルに Debezium Db2 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コラムで 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ー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 コネクターを設定します。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 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
    database.server.name: server1 9
    database.history.kafka.bootstrap.servers: kafka:9092 10
    database.history.kafka.topic: schema-changes.inventory 11
```

表6.10 コネクター設定の説明

項目	説明
1	Kafka Connect サービスに登録する場合のコネクターの名前。
2	この Oracle コネクタークラスの名前。
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	コネクターのユーザーの作成 で指定した Oracle ユーザーの名前。
6	コネクターのユーザーの作成 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前。
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。コンテナーデータベース (CDB) のインストールのみで使用されます。
9	コネクターが変更を取得する Oracle データベース・サーバーの nampespace を特定して提供する論理名。

項目	説明
10	DDL ステートメントをデータベース履歴トピックに書き込み、復元するためにコネクターによって使用される Kafka ブローカーのリスト。
11	コネクターが DDL ステートメントを書き、復元するデータベース履歴トピックの名前。このトピックは内部使用のみを目的としており、コンシューマーが使用しないようにしてください。

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 **server1** データベースに対して実行を開始します。

6.5.4. Oracle JDBC ドライバーの取得

ライセンス要件により、必要なドライバーファイルは Debezium Oracle コネクターアーカイブには含まれません。使用するデプロイメント方法に関係なく、デプロイメントを完了するための Oracle JDBC ドライバーを入手します。

使用するデプロイメント方法に応じて、ドライバーを取得する方法は 2 つあります。

- AMQ Streams を使用してコネクターを Kafka Connect イメージに 追加する場合は、アーティファクト参照を **KafkaConnect** カスタムリソースに追加し、アーティファクトの場所を **url** 値として追加します。
- Dockerfile を使用してコネクターを構築 する場合は、必要なドライバーファイルを Oracle から直接ダウンロードし、Kafka Connect 環境に追加します。

以下の手順では、お使いの環境でドライバーと利用可能にする方法を説明します。

手順

1. デプロイメントのタイプに応じて、以下のいずれかの手順を実行します。
   - AMQ Streams を使用してコネクターをデプロイします。
     a. Maven Central に移動し、Oracle Database のリリース用に **ojdbc8.jar** ファイルを見つけます。
     b. **KafkaConnector** カスタムリソース (CR) の YAML で、 **debezium-connector-oracle** アーティファクトの **artifacts.url** フィールドにドライバーの URL パスを追加します。**KafkaConnector** CR の YAML ファイルの詳細は、AMQ ストリームを使用した Debezium Oracle コネクターのデプロイを参照してください。
   - Dockerfile を使用してコネクターをデプロイする場合:
     a. ブラウザーから、Oracle JDBC および UCP Downloads ページ に移動します。
     b. お使いのバージョンの Oracle Database 用の **ojdbc8.jar** ドライバーファイルを見つけてダウンロードします。

c. ダウンロードしたファイルを、**<kafka_home>/libs** などの Debezium Oracle コネクターファイルを保存するディレクトリーにコピーします。
コネクターが起動すると、指定されたドライバーを使用するように自動的に設定されます。

6.5.5. コンテナーデータベースとノンコンテナー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。

コンテナーデータベース (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。

ノンコンテナーデータベース (非 CDB)

プラガ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。

Debezium Oracle コネクターに設定できる設定プロパティーの完全リストは Oracle コネクタープロパティー 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。

6.5.6. 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 コネクターは 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-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_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

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                                                                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-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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

- 端末画面で、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_conne
ctor_oracle.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
_oracle.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
_oracle.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.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":"1.7.2.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 コネクター設定プロパティーの説明

Debezium Oracle 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ーが多数あります。多くのプロパティーにはデフォルト値があります。プロパティーに関する情報は、以下のように設定されています。

- 必要な Debezium Oracle コネクター設定プロパティー
- 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設定プロパティー。
  - パススルーデータベースの履歴プロパティー
- 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ー。

必要な Debezium Oracle コネクター設定プロパティー

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

プロパティー	デフォルト	説明
name	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。(このプロパティーはすべての 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。Oracle コネクターには、常に **io.debezium.connector.oracle.OracleConnector** の値を使用します。
tasks.max	**1**	このコネクターのために作成する必要のあるタスクの最大数。Oracle 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	Oracle 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	デフォルトなし	Oracle 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	コネクターが 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。
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。このプロパティーを使用すると、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。このプロパティーは、コンテナーデータベース (CDB) のインストールでのみ使用してください。
database.server.name	デフォルトなし	コネクターが変更を取得する Oracle データベース・サーバーの nampespace を特定して提供する論理名。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論理名を指定します。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。

database.connection.adapter	logminer	コネクター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。以下の値を設定できます。 **logminer(デフォルト)** このコネクターはネイティブの Oracle LogMiner API を使用します。 **xstream** このコネクターは Oracle XStreams API を使用します。
snapshot.mode	Initial	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。 **Initial** 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。 **schema_only** 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。 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。
snapshot.locking.mode	shared	コネクター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、コネクター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。 **shared** 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。コネクターは、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。 **none** スナップショット中にコネクターがテーブル 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。

snapshot.include.collection.list	**table.include.list**に指定したすべてのテーブル	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)。指定する項目は、コネクターの **table.include.list** プロパティーで名前を付ける必要があります。このプロパティーは、コネクターの **snapshot.mode** プロパティーが **never** 以外の値に設定されている場合にのみ有効になります。 このプロパティーは増分スナップショットの動作には影響しません。

snapshot.select.statement.overrides	デフォルトなし	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、プロパティーを使用します。このプロパティー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。 プロパティーには、**<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、 **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** 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ーを追加して、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** 文のプロパティーの名前を指定するには、次の形式を使用します。 **snapshot.select.statement.overrides.<schemaName>.<tableName>** 例: **snapshot.select.statement.overrides.customers.orders** 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、以下のプロパティーを追加します。 `"snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"` 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。

schema.include.list	デフォルトなし	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。また、**schema.exclude.list** プロパティーも設定しないでください。LogMiner 実装を使用する環境では、POSIX 正規表現のみを使用する必要があります。
schema.exclude.list	デフォルトなし	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。また、**schema.include.list** プロパティーも設定しないでください。LogMiner 実装を使用する環境では、POSIX 正規表現のみを使用する必要があります。
table.include.list	デフォルトなし	監視するテーブルの完全修飾テーブル識別子と一致する正規表現のコンマ区切りリスト (任意)。include リストに含まれていないテーブルは監視から除外されます。各テーブルの識別子は以下の形式を使用します。 **<schema_name>.<table_name>** デフォルトでは、コネクターは各監視対象データベースのすべての非システムテーブルを監視します。このプロパティーは **table.exclude.list** と組み合わせて使用しないでください。LogMiner 実装を使用する場合は、このプロパティーで POSIX 正規表現のみを使用してください。
table.exclude.list	デフォルトなし	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)。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。 **<schemaName>.<tableName>** 形式を使用して、各テーブルの識別子を指定します。 このプロパティーは **table.include.list** と組み合わせて使用しないでください。LogMiner 実装を使用する場合は、このプロパティーで POSIX 正規表現のみを使用してください。

column.include.list	デフォルトなし	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)。カラムの完全修飾名は以下の形式を使用します。**+ `<Schema_name>.<table_name>.<column_name>** 主キーカラムは、このプロパティーを使っ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。このプロパティーを設定に含める場合は、**column.exclude.list** プロパティーを設定しないでください。
column.exclude.list	デフォルトなし	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)。完全修飾のカラム名は以下の形式を使用します。 **<schema_name>.<table_name>.<column_name>** 主キーカラムは、このプロパティー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。このプロパティーを設定に含める場合は、**column.include.list** プロパティーを設定しないでください。

column.mask.hash.<hashAlgorithm>.with.salt.<salt>	デフォルトなし	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 **<schemaName>.<tableName>.<columnName>** です。 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。 仮名は、指定された **hashAlgorithm** と **salt** 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、参照整合性は維持され、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 section に説明されています。 以下の例では、**CzQMA0cB5K** が無作為に選択された salt になります。 `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` 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。 使用される **hashAlgorithm**、選択された **salt**、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。

decimal.handling.mode	**precise**	コネクターが **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。 **precise (デフォルト)** 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。 **double** **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。 **string** 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、10 進数型 を参照してください。
event.processing.failure.handling.mode	**fail**	イベントの処理中にコネクターが例外に対応する方法を指定します。以下のオプションのいずれかを使用できます。 **fail** 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。 **warn** 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。 **skip** 問題のあるイベントがスキップされるようにします。
max.queue.size	**8192**	ブロッキングキューの最大サイズの正の整数値。データベースログから読み取られる変更イベントは、Kafka に書き込まれる前にブロッキングキューに置かれます。このキューは、Kafka への書き込みが遅い場合や Kafka が利用できない場合などに、binlog リーダーにバックプレシャーを提供できます。キューに表示されるイベントは、コネクターによって定期的に記録されるオフセットには含まれません。**max.batch.size** プロパティーに指定された最大バッチサイズよりも大きい値を常に指定します。
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.in.bytes	**0** (無効)	ブロッキングキューの最大サイズ (バイト単位) の long 値。機能を有効にするには、値を正の long データ型に設定します。
poll.interval.ms	**1000** (1 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ーが待機する時間 (ミリ秒単位) を指定する正の整数値。
tombstones.on.delete	**true**	**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。 **true** 削除操作ごとに、コネクターは、**削除** イベントと後続の廃棄 (tombstone) イベントを出力します。 **false** 削除操作ごとに、コネクターは **削除** イベントのみを出力します。 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 log compaction 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。

message.key.columns	デフォルトなし	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。 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。各リストエントリーは以下の形式をとります。 **<fullyQualifiedTableName>:<keyColumn>,<keyColumn>** 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。 **<schemaName>.<tableName>** プロパティーには複数のテーブルのエントリー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ーを区切ります。 以下の例では、テーブル **inventory.customers** と **purchase.orders** にメッセージキーを設定しています。 **inventory.customers:pk1,pk2;(.*).purchaseorders:pk3,pk4** テーブル **inventory.customer** では、列 **pk1** と **pk2** がメッセージキーとして指定されています。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。 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。

column.truncate.to.length.chars	デフォルトなし	指定された文字数より長い場合に、変更イベントメッセージで値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。長さは正の整数として指定されます。設定には、異なる長さ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。**<schemaName>.<tableName>.<columnName>** 形式を使用して列の完全修飾名を指定します。
column.mask.with.length.chars	デフォルトなし	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)。 プロパティー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でなければなりません。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。**<schemaName>.<tableName>.<columnName>** の形式を使用して、列の完全修飾名を指定します。 コネクター設定には、異なる長さ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。
column.propagate.source.type	デフォルトなし	出力された変更メッセージの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。 列の完全修飾名の形式は **<tableName>.<columnName>** または **<schemaName>.<tableName>.<columnName>** です。

datatype.propagate.source.type	デフォルトなし	出力された変更メッセージフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列のデータベース固有のデータ型名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。 完全修飾データ型名の形式は **<tableName>.<typeName>** または **<schemaName>.<tableName>.<typeName>** です。 Oracle 固有のデータ型名のリストを参照してください。
heartbeat.interval.ms	**0**	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。 このプロパティーを使用して、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。 コネクターで長期間、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、このプロパティーを設定すると便利です。 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。 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。
heartbeat.topics.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名の接頭辞を使用する文字列を指定します。 トピックは、**<heartbeat.topics.prefix>.<serverName>** パターンに従って名前が付けられます。

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、コネクター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。 このプロパティーを使用して、クラスターで複数の接続を開始するときに (コネクター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。コネクターは、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。
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.metadata	**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**	Oracle Log Miner がテーブルとカラムの ID を名前に解決するために、指定されたデータディクショナリーを構築して使用する方法を制御するマイニング戦略を指定します。 **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、時間の経過とともにアーカイブログが多く生成されます。また、キャプチャしたテーブルに対する DDL の変更を追跡することができるので、スキーマが頻繁に変更される場合には理想的な選択です。 **online_catalog**::データベースの現在のデータ辞書を使用してオブジェクト ID を解決し、オンライン REDO ログに余分な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。

log.mining.buffer.type	memory	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ーリングをどのように管理するかを制御します。 **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ーリングします。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ー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 値からバッファーを再作成します。
log.mining.batch.size.min	1000	このコネクターが 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ー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。
log.mining.batch.size.max	100000	このコネクターが 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。
log.mining.batch.size.default	20000	コネクターが 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。
log.mining.sleep.time.min.ms	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ーのスリープ時間の最小値です。値はミリ秒単位です。
log.mining.sleep.time.max.ms	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ーイルのスリープ時間の最大値。値はミリ秒単位です。
log.mining.sleep.time.default.ms	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ーのスリープ時間の開始値。値はミリ秒単位です。
log.mining.sleep.time.increment.ms	200	logminer からデータを読み取る際に、コネクター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。

log.mining.view.fetch.size	**10000**	コネクターが LogMiner コンテンツビューからフェッチしたコンテンツレコードの数。
log.mining.archive.log.hours	**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、コネクターはすべてのアーカイブログを粉砕します。
log.mining.archive.log.only.mode	**false**	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て挽くのか (デフォルト) を制御します。 redo ログは円形のバッファーを使用しており、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ことがあります。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。
log.mining.archive.log.only.scn.poll.interval.ms	**10000**	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。
log.mining.transaction.retention.hours	**0**	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数を指定します。**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。 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ーを維持します。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ーされるため、そのバッファー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。

log.mining.archive.destination.name	デフォルトなし	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。 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。
log.mining.username.exclude.list	デフォルトなし	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、このプロパティーを設定すると便利です。
log.mining.scn.gap.detection.gap.size.min	**1000000**	SCN ギャップがあるかどうかを判断するために、コネクターが現在の 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、コネクター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。
log.mining.scn.gap.detection.time.interval.max.ms	**20000**	SCN ギャップがあるかどうかを判断するために、コネクターが現在の 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、コネクターは設定された最大バッチより大きいマイニングウィンドウを使用します。
lob.enabled	**false**	ラージオブジェクト (CLOB または BLOB) のカラム値を変更イベントで出力するかどうかを制御します。 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カラム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。

unavailable.value.placeholder	**__debezium_unavailable_value**	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。

rac.nodes	デフォルトなし	Oracle Real Application Clusters (RAC) ノードのホスト名またはアドレスをコンマで区切って入力してください。このフィールドは、Oracle RAC サポートを有効にするために必要です。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。
		• 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。
		`database.port=1521`
		`rac.nodes=192.168.1.100,192.168.1.101`
		• 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。
		`database.port=1521`
		`rac.nodes=192.168.1.100,192.168.1.101:1522`
		database.url プロパティー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。

skipped.operations	デフォルトなし	ストリーミング中にコネクターがスキップする操作タイプをコンマで区切ったリスト。以下のタイプの操作をスキップするようにコネクターを設定することができます。 • **c** (挿入/作成) • **u** (更新) • **d** (削除) デフォルトでは、操作はスキップされません。

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次のフォーマットを使用します。 **<databaseName>.<schemaName>.<tableName>**
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。

Debezium Oracle コネクターデータベース履歴設定プロパティー

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 **database.history.*** プロパティー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 **database.history** プロパティーについて説明しています。

表6.11 コネクターデータベース履歴設定プロパティー

プロパティー	デフォルト	説明
database.history.kafka.topic		コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。
database.history.kafka.bootstrap.servers		Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。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。
database.history.kafka.recovery.poll.interval.ms	**100**	永続化されたデータのポーリングが行われている間にコネクターが起動/回復を待つ最大時間 (ミリ秒単位) を指定する整数値。デフォルトは 100 ミリ秒です。
database.history.kafka.recovery.attempts	**4**	エラーでコネクターのリカバリーが失敗する前に、コネクターが永続化された履歴データの読み取りを試行する最大回数。データが受信されなかった場合に最大待機する時間は、**recovery.attempts** x **recovery.poll.interval.ms** です。

プロパティー	デフォルト	説明
database.history.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。
database.history.store.only.monitored.tables.ddl **今後のリリースで非推奨になり、削除される予定です。代わりに database.history.store.only.captured.tables.ddl を使用してください。**	**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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 安全なデフォルトは **false** です。
database.history.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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 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ー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履歴プロパティー

Debezium は、Kafka プロデューサーを使用して、データベース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。**database.history.producer.*** および **database.history.consumer.*** 接頭辞で始まるパススルー設定プロパティー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ー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ーデータベース履歴プロパティーは、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。

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

database.history.consumer.security.protocol=SSL
```

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

Debezium は、プロパティーを Kafka クライアントに渡す前に、プロパティー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ー および Kafka コンシューマー設定プロパティーの詳細は、Kafka のドキュメントを参照してください。

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

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ーは、接頭辞 **database.*** で始まります。たとえば、コネクターは **database.foobar=false** などのプロパティーを JDBC URL に渡します。

データベース履歴クライアントのパススループロパティー の場合のように、Debezium はプロパティー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。

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	型	説明
LastEvent	**string**	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	**long**	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。

属性	型	説明
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 リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨で、将来のリリースで削除される予定です。代わりに CapturedTables メトリックを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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。
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)。

属性	型	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ー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

コネクターは、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	型	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。

重要

増分スナップショットの使用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 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。

6.7.2. Debezium Oracle コネクター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	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨で、将来のリリースで削除される予定です。代わりに CapturedTables メトリックを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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。

属性	型	説明
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。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

Debezium Oracle 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6.12 追加のストリーミングメトリクスの説明

属性	型	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ー内の最も古いシステム変更番号。

属性	型	説明
ComittedScn	string	トランザクションバッファーからの最後のコミットされたシステム変更番号。
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)。

属性	型	説明
LastBatchProcessingTimeInMilliseconds	long	最後の LogMiner クエリーバッチ処理の時間 (ミリ秒単位)。
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トリーの数。

属性	型	説明
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)。
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ー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ー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ー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ー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ーのコミットされた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ーに登録された DML 操作の数。

属性	型	説明
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ーに追加された時刻の差 (ミリ秒単位)。
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ー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ーに追加された時刻の差の最小値 (ミリ秒単位)。
AbandonedTransactionIds	string[]	経過時間によりトランザクションバッファーから削除された放棄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ー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ー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ー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。

属性	型	説明
ScnFreezeCount	int	昇格のためにシステム変更番号がチェックされ、変更されなかった回数。これは、長時間実行されるトランザクションが継続され、コネクターが処理された最新のシステム変更番号をコネクターのオフセットにフラッシュできないことを示しています。最適な操作では、これは常に **0** かそれに近い状態のままになります。
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	型	説明
Status	string	データベース履歴の状態を示す **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。

6.8. DEBEZIUM ORACLE 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、Debezium 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。

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

ORA-25191: インデックス編集テーブルのオーバーフローテーブルを参照できない

Oracle は、インデックス編集テーブル (IOT) に遭遇すると、スナップショットフェーズでこのエラーが発生する可能性があります。このエラーは、コネクターが、指定のオーバーフローテーブルが含まれる親のインデックス編集テーブルに対して実行する必要がある操作の実行を試行したことを意味します。

これを解決するには、SQL 操作で使用される IOT 名を、親のインデックス編集テーブル名に置き換える必要があります。親のインデックス編集テーブル名を確認するには、以下の SQL を使用します。

```
SELECT IOT_NAME
  FROM DBA_TABLES
 WHERE OWNER='<tablespace-owner>'
   AND TABLE_NAME='<iot-table-name-that-failed>'
```

コネクターの **table.include.list** または **table.exclude.list** 設定オプションを調整して、コネクターが子のインデックス編集テーブルから変更をキャプチャーしないように、適切なテーブルを明示的に追加または除外する必要があります。

LogMiner アダプターは、SYS または SYSTEM による変更をキャプチャーしません。

Oracle は多くの内部変更に **SYS** および **SYSTEM** アカウントを使用するため、コネクターは LogMiner から変更を取得するときに、これらのユーザーによる変更を自動的にフィルターします。Debezium Oracle コネクターによって出力される変更には、**SYS** または **SYSTEM** ユーザーアカウントを使用しないでください。

第7章 POSTGRESQL の DEBEZIUM コネクター

Debezium の PostgreSQL コネクターは、PostgreSQL データベースのスキーマで行レベルの変更をキャプチャーします。このコネクターと互換性のある 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 サーバーまたはクラスターに初めて接続すると、コネクター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、コネクター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。コネクター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、コネクター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 コネクター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 コネクターの概要」
- 「Debezium PostgreSQL コネクターの仕組み」
- 「Debezium PostgreSQL コネクターのデータ変更イベントの説明」
- 「Debezium PostgreSQL コネクターによるデータ型のマッピング方法」
- 「Debezium コネクターを実行するための PostgreSQL の設定」
- 「Debezium PostgreSQL コネクターのデプロイメント」
- 「Debezium PostgreSQL コネクターのパフォーマンスの監視」
- 「Debezium PostgreSQL コネクターによる障害および問題の処理方法」

7.1. DEBEZIUM POSTGRESQL コネクターの概要

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 コネクターには、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 コミュニティーにより維持され、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 コネクターは、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 コネクター)。

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

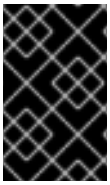
コネクターはフォールトトラレントです。コネクターは変更を読み取り、イベントを生成するため、各イベントの WAL の位置を記録します。コネクターが何らかの理由で停止した場合 (通信障害、ネットワークの問題、クラッシュなど)、コネクターは再起動後に最後に停止した場所から WAL の読み取りを続行します。これにはスナップショットが含まれます。スナップショット中にコネクターが停止した場合、コネクターは再起動時に新しいスナップショットを開始します。

重要

コネクターは 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、コネクターは 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、コネクターはアクティブな **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、コネクターは停止します。**primary** サーバーの回復後に、コネクターを再起動できます。別の PostgreSQL サーバーが **primary** に昇格された場合は、コネクターの設定を調整してからコネクターを再起動します。

Debezium PostgreSQL コネクターによる障害および問題の処理方法 には、問題が発生した場合のコネクターの動作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 コネクターの仕組み

Debezium PostgreSQL コネクターを最適に設定および実行するには、コネクター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 コネクターによるデータベーススナップショットの実行方法」
- 「Debezium PostgreSQL コネクター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」
- 「Debezium PostgreSQL 変更イベントレコードのメタデータ」
- 「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」

7.2.1. PostgreSQL コネクターのセキュリティー

Debezium コネクターを使用して PostgreSQL データベースから変更をストリーミングするには、コネクター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ああります。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 コネクター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 コネクターは WAL 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、コネクターが最初に起動すると、データベースの最初の **整合性スナップショット** が実行されます。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** コネクター設定プロパティー 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. コネクターオフセットにスナップショットの正常な完了を記録します。

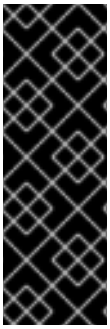
コネクターに障害が発生した場合、コネクターのリバランスが発生した場合、または 1. の後で 6. の完了前に停止した場合、コネクターは再起動後に新しいスナップショットを開始します。コネクターによって最初のスナップショットが完了した後、PostgreSQL コネクターは 3. で読み取りした位置からストリーミングを続行します。これにより、コネクターが更新を見逃さないようします。何らかの理由でコネクターが再び停止した場合、コネクターは再起動後に最後に停止した位置から変更のストリーミングを続行します。

表7.1 snapshot.mode コネクター設定プロパティーの設定

設定	説明

設定	説明
always	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。 ● 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。 ● クラスターの障害後に、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	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。
initial_only	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。
exported	非推奨、全てのモードがロックレスになります。

7.2.2.1. 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の使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。通常の状況では、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、コネクター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。

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**	スナップショットを作成するテーブルの完全修飾名が含まれる配列。 名前の形式は **signal.data.collection** 設定オプションと同じです。

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

重要

増分スナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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、ストリーム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ーに提供します。スナップショットウィンドウ中に、バッファー内の **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、バッファーされた **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、バッファーに含まれるのは、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。

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。SQL **INSERT** クエリーとしてテーブルにシグナルを送信します。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、テーブルを一覧表示する **data-collections**アレイを指定します (例: **{"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。

前提条件

- シグナルが有効になっている。

- シグナルデータコレクションがソースのデータベースに存在し、コネクターはこれをキャプチャーするように設定されています。
- 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>_"}');
```

以下に例を示します。

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

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

以下の表では、これら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

値	説明
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。
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。
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。
data-collections	スナップショットに含めるテーブル名の配列を指定するシグナルの **data** フィールドの必須コンポーネント。 配列は、**signal.data.collection** 設定プロパティーにコネクターのシグナルテーブルの名前を指定するときに使用する形式で、完全修飾名別にテーブルを一覧表示します。
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。 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。

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。

以下に例を示します。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。
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。

警告

PostgreSQL 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。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。

7.2.3. 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

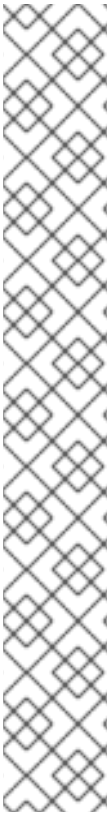
通常、PostgreSQL コネクターは、接続されている PostgreSQL サーバーから変更をストリーミングするのに大半の時間を費やします。このメカニズムは、**PostgreSQL のレプリケーションプロトコル** に依存します。このプロトコルにより、クライアントはログシーケンス番号 (LSN) と呼ばれる特定の場所で変更がサーバーのトランザクションログにコミットされる際に、サーバーから変更を受信することができます。

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 コネクターでは、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、コネクターを停止し、すべてのイベントレコードを Kafka にフラッシュして、各コネクター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、各コネクター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ーを起動します。コネクターを再起動すると、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 コネクターは、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、コネクター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。コネクターは 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 コネクター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。

serverName.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。

serverName
: **database.server.name** コネクター設定プロパティーで指定したコネクターの論理名です。

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。

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 インスト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. Debezium PostgreSQL 変更イベントレコードのメタデータ

PostgreSQL コネクターによって生成された各レコードには、**データベース変更イベント** の他に、一部のメタデータも含まれています。メタデータには、サーバーでイベントが発生した場所、ソースパーティションの名前、イベントが置かれる Kafka トピックおよびパーティションの名前が含まれています。

```
"sourcePartition": {
```

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

- **source Partition** は、常に **database.server.name** コネクター設定プロパティーの設定をデフォルトとします。
- **sourceOffset** にはイベントが発生したサーバーの場所に関する情報が含まれています。
  - **lsn** はトランザクションログの PostgreSQL ログシーケンス番号 または **offset** を表します。
  - **txId** はイベント発生の原因となったサーバートランザクションの識別子を表します。
  - **ts_ms** はトランザクションがコミットされたサーバー時間をエポックからの経過時間 (ミリ秒単位) で表します。
- **kafkaPartition** に **null** が設定されると、コネクターは特定の Kafka パーティションを使用しません。PostgreSQL コネクターは Kafka Connect パーティションを 1 つだけ使用し、生成されたイベントを 1 つの Kafka パーティションに配置します。

7.2.6. 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** - 一意のトランザクション識別子の文字列表現。
- **event_count** (**END** イベントの場合) - トランザクションによって出力されたイベントの合計数。
- **data_collections** (**END** イベントの場合): 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。

例

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

```
  "id": "571",
  "event_count": null,
  "data_collections": null
}

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

トランザクションイベントは、**database.server.name.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",
    "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、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": { 1
   ...
 },
 "payload": { 2
   ...
 },
 "schema": { 3
   ...
 },
 "payload": { 4
   ...
 },
}
```

表7.4 変更イベントの基本内容の概要

項目	フィールド名	説明

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ー 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ー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。

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

PosgreSQL 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)
);
```

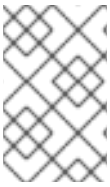
変更イベントキーの例

database.server.name コネクター設定プロパティーに **PostgreSQL_server** の値がある場合、この定義がある限り **customers** テーブルの変更イベントはすべて同じキー構造を持ち、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7.5 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	**PostgreSQL_server.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。 ● **PostgreSQL_server** 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** コネクター設定プロパティーは、テーブル列のサブセットのみをキャプチャー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、コネクターはそのテーブルの **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、コネクターは **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": "PostgreSQL_server.inventory.customers.Value", 2
                "field": "before"
            },
            {
                "type": "struct",
                "fields": [
                    {
                        "type": "int32",
                        "optional": false,
                        "field": "id"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "first_name"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "last_name"
                    },
                    {
                        "type": "string",
                        "optional": false,
                        "field": "email"
                    }
                ],
```

```
            "optional": true,
            "name": "PostgreSQL_server.inventory.customers.Value",
            "field": "after"
        },
        {
            "type": "struct",
            "fields": [
                {
                    "type": "string",
                    "optional": false,
                    "field": "version"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "connector"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "name"
                },
                {
                    "type": "int64",
                    "optional": false,
                    "field": "ts_ms"
                },
                {
                    "type": "boolean",
                    "optional": true,
                    "default": false,
                    "field": "snapshot"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "db"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "schema"
                },
                {
                    "type": "string",
                    "optional": false,
                    "field": "table"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "txId"
                },
                {
                    "type": "int64",
                    "optional": true,
```

```
                    "field": "lsn"
                },
                {
                    "type": "int64",
                    "optional": true,
                    "field": "xmin"
                }
            ],
            "optional": false,
            "name": "io.debezium.connector.postgresql.Source", 3
            "field": "source"
        },
        {
            "type": "string",
            "optional": false,
            "field": "op"
        },
        {
            "type": "int64",
            "optional": true,
            "field": "ts_ms"
        }
    ],
    "optional": false,
    "name": "PostgreSQL_server.inventory.customers.Envelope" 4
},
"payload": { 5
    "before": null, 6
    "after": { 7
        "id": 1,
        "first_name": "Anne",
        "last_name": "Kretchmar",
        "email": "annek@noanswer.org"
    },
    "source": { 8
        "version": "1.7.2.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7.6 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**	**schema** セクションでは、各**name** フィールドが、値のペイロード内のフィールドのスキーマを指定します。 **Postgre SQL_server.inventory.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.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 スキーマには独自の進化と履歴があります。
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**	**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。
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。
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。 **注記** 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。
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**	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。 ● 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 ● 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
9	**op**	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。 ● **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**	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。 **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。

更新イベント

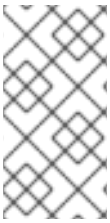
サンプル **customers** テーブルにある更新の変更イベントの値には、そのテーブルの **作成** イベントと同じスキーマがあります。同様に、イベント値のペイロードは同じ構造を持ちます。ただし、イベント値ペイロードでは **更新** イベントに異なる値が含まれます。以下は、コネクターによって **customers** テーブルでの更新に生成されるイベントの変更イベント値の例になります。

```
{
    "schema": { ... },
    "payload": {
        "before": { 1
            "id": 1
        },
        "after": { 2
            "id": 1,
            "first_name": "Anne Marie",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": { 3
            "version": "1.7.2.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7.7 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。

項目	フィールド名	説明
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。 ● 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。
5	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。



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。

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 **payload** 部分は以下のようになります。

```
{
    "schema": { ... },
    "payload": {
        "before": { 1
            "id": 1
        },
        "after": null, 2
        "source": { 3
            "version": "1.7.2.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7.8 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。 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。
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**	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。 ● 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**	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**	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。 **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。

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": { 1
            "version": "1.7.2.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7.9 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**	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。 ● 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**	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。
3	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。

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。

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。

- **リテラル型** は、Kafka Connect スキーマ型を使用して、値をリテラルで表す方法を記述します。**INT8**、**INT16**、**INT32**、**INT64**、**FLOAT32**、**FLOAT64**、**BOOLEAN**、**STRING**、**BYTES**、**ARRAY**、**MAP**、および **STRUCT**。
- **セマンティック型** 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。

詳細は以下を参照してください。

- 基本型
- Temporal (一時) 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、コネクターによる基本型へのマッピング方法を説明しています。

表7.10 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。

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
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
CHAR[(M)]	**STRING**	該当なし
VARCHAR[(M)]	**STRING**	該当なし
CHARACTER[(M)]	**STRING**	該当なし
CHARACTER VARYING[(M)]	**STRING**	該当なし
TIMESTAMPTZ, **TIMESTAMP WITH TIME ZONE**	**STRING**	**io.debezium.time.ZonedTimestamp** タイムゾーン情報を含むタイムスタンプの文字列表現。タイムゾーンは GMT です。
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)。

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
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 コネクターの バイナリー処理モード 設定に基づいた raw バイト (デフォルト)、base64 でエンコードされた文字列、または 16 進数でエンコードされた文字列。
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。

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
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。
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** Postgre SQL 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。

Temporal (一時) 型

タイムゾーン情報が含まれる PostgreSQL の **TIMESTAMPTZ** 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ー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 プロパティーがデフォルトの **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7.11 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

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。
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、コネクター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7.12 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。

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
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7.13 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。

Postgre SQL 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、Postgre SQL JDBC ドライバーの標準的な動作を模倣しています。参考として **org.postgresql.PGStatement** インタフェー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 プロパティーが **precise** に設定されている場合には コネクターは**DECIMAL** と **NUMERIC** 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7.14 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。

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、コネクターは **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7.15 スケーリング制約がない場合の 10 進数型のマッピング

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。

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
DECIMAL	**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 プロパティーが **double** に設定されている場合、コネクターはすべての **DECIMAL** および **NUMERIC** 値を Java の double 値として表し、次の表のようにエンコードします。

表7.16 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**	

decimal.handling.mode 設定プロパティーの最後の設定は **string** です。この場合、コネクターは **DECIMAL** および **NUMERIC** 値をフォーマットされた文字列表現として表し、それらを以下の表のようにエンコードします。

表7.17 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**	

Postgre SQL 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 **Na N**(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

dhstore.handling.mode コネクター設定プロパティーが **json** (デフォルト) に設定されている場合、コネクターは **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** プロパティーが **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7.18 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 **{"key" : "val"}**
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と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7.19 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

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
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7.20 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。 ● **srid (INT32)** - 構造に保存されるジオメトリーオブジェクトの型を定義する、空間参照システム識別子。 ● **wkb (BYTES)** - Well-Known-Binary 形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。 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。 ● **srid (INT32)** - 構造に保存されるジオグラフィーオブジェクトの型を定義する、空間参照システム識別子。 ● **wkb (BYTES)** - Well-Known-Binary 形式でエンコードされたジオメトリーオブジェクトのバイナリー表現。 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 ストレージを使って保存する必要があるのです。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ー **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カラムにデフォルト値が指定されている場合、Postgre SQL コネクター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、コネクター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、コネクターは代わりに、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 Postgre SQL コネクターを使用する際に、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。
- コネクター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、カラム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、コネクター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ーが一時的に停止した場合に、コネクター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 コネクターを実行するための 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 コネクターの 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 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います。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 コネクターを使用する場合に役立ちます。

7.5.2. Debezium コネクターの PostgreSQL パーミッションの設定

PostgreSQL サーバーを設定して Debezium コネクターを実行するには、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。

セキュリティー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で必要な **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 コネクターが作成できます。Debezium PostgreSQL コネクターに、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、コネクター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、コネクターを設定する前に、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 プロパティーを使用して、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になります。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)
```

(1) 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。

(2) **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。

(3) **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ーを設定します。

- 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** コネクター設定プロパティーを使用して、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に送信します。

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** コネクター設定プロパティーを使用して自動化できます。

7.6. DEBEZIUM POSTGRESQL 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 PostgreSQL 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 コンテナーイメージをビルドします。

関連情報

- 「Debezium PostgreSQL コネクター設定プロパティーの説明」

7.6.1. AMQ Streams を使用した PostgreSQL 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 コンテナー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**コンテナーイメージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ー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 にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。
- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

  - 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 として保存します。

  - 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。

手順

1. OpenShift クラスターにログインします。

2. コネクターの新しい Debezium **KafkaConnect** カスタムリソース (CR) を作成します。たとえば、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

例7.1 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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-postgres
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-postgres/1.7.2.Final-redhat-<build_number>/debezium-connector-postgres-1.7.2.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.0-redhat-<build-number>.zip
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表7.21 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、クラスターオペレーターが **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ー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。

項目	説明
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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメント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。

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

```
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ー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、**postgresql-inventory-connector.yaml** として保存します。

例7.2 Debezium コネクターの 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.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-
bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: postgresql.debezium-postgresql.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_postgresql (10)
    database.include.list: public.inventory (11)
```

表7.22 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 がデータベースに接続するユーザーアカウントの名前。
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター 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 コンテナーイメージをビルドして Debezium PostgreSQL コネクターのデプロイ

Debezium PostgreSQL コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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

- PostgreSQL が実行され、PostgreSQL を設定して Debezium コネクターを実行する 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 コネクターを実行するコンテナーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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 コンテナーを作成します。
   a. Debezium PostgreSQL コネクターアーカイブ をダウンロードします。
   b. Debezium PostgreSQL コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-postgresql.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1)(1)(1) 任意のファイル名を指定できます。

(2)(2)(2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

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。

d. 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ー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ーイメージを構築します。

e. カスタムイメージを **quay.io** などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** および **image** プロパティーを指定する **dbz-connect.yaml** という名前の **KafkaConnect** CR を作成します。

```
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-postgresql (2)
```

(1) **KafkaConnector** リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の

g. 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 Connector インスタンスが追加されます。

2. Debezium PostgreSQL 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ーを設定する **.yaml** ファイルに Debezium PostgreSQL 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コラムで 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ーを設定する可能性もあります。Debezium PostgreSQL コネクターに設定できる設定プロパティーの完全リストは PostgreSQL コネクタープロパティー を参照してください。

以下の例では、ポート **5432** で PostgreSQL サーバーホスト **192.168.99.100** に接続する Debezium コネクターを設定します。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があり、**fulfillment** はサーバーの論理名です。

fulfillment-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
 kind: KafkaConnector
 metadata:
   name: fulfillment-connector 1
   labels:
     strimzi.io/cluster: my-connect-cluster
 spec:
   class: io.debezium.connector.postgresql.PostgresConnector
   tasksMax: 1 2
   config: 3
     database.hostname: 192.168.99.100 4
     database.port: 5432
     database.user: debezium
     database.password: dbz
     database.dbname: sampledb
     database.server.name: fulfillment 5
     schema.include.list: public 6
     plugin.name: pgoutput 7
```

1 コネクターの名前。

2 1 度に 1 つのタスクのみが動作する必要があります。PostgreSQL コネクターは PostgreSQL サーバーの **192.168.99.100** を読み取るため、単一のコネクタータスクを使用することで、順序とイベントの処理が適切に行われるようになります。Kafka Connect サービスはコネクターを使用して作業を行う 1 つ以上のタスクを開始し、実行中のタスクを自動的に Kafka Connect サービスのクラスター全体に分散します。いずれかのサービス

が停止またはクラッシュすると、これらのタスクは稼働中のサービスに再分散されます。

3 コネクターの設定。

4 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。

5 一意のサーバー名。サーバー名は、PostgreSQL サーバーまたはサーバーのクラスターの論理識別子です。この名前は、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。

6 コネクターは **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ーを設定できます。**table.include.list** コネクター設定プロパティーを参照してください。

7 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。Postgre SQL 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 でコネクターインスタンスを作成します。たとえば、**KafkaConnector** リソースを **fulfillment-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 fulfillment-connector.yaml
```

このコマンドは **meetment-connector** を登録して、コネクターが **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

コネクターが起動すると、コネクターが設定された 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。

7.6.4. Debezium PostgreSQL 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 コネクターは 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-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_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

  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
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-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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

- 端末画面で、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_conne
ctor_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.connector.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":"transaction"}],"optional":false,"name":"inventory_connector_postgresql.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":{"version":"1.7.2.Final-redhat-00001","connector":"postgresql","name":"inventory_connector_postgresql","ts_ms":1638985247805,"snapshot":"true","db":"inventory","sequence":null,"table":"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,"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。

7.6.5. Debezium PostgreSQL コネクター設定プロパティーの説明

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ーが多数あります。多くのプロパティーにはデフォルト値があります。プロパティーに関する情報は、以下のように設定されています。

- 必要な設定プロパティー
- 高度な設定プロパティー
- パススルー設定プロパティー

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

表7.23 必要なコネクター設定プロパティー

プロパティー	デフォルト	説明

プロパティー	デフォルト	説明
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。このプロパティーはすべての 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。Postgre SQL コネクターには、常に **io.debezium.connector.postgresql.PostgresConnector** の値を使用してください。
tasks.max	**1**	このコネクターのために作成する必要のあるタスクの最大数。PostgreSQL コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。
plugin.name	**decoderbufs**	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。 サポートされている値は **pgoutput** のみです。**pgoutput** には **plugin.name** を明示的に設定する必要があります。
slot.name	**debezium**	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 コネクターにイベントをストリーミングします。 スロット名は、Postgre SQL のレプリケーションスロットネーミングルールに準拠していなければなりません。**"各レプリケーションスロットには名前があり、小文字のアルファベット、数字、アンダースコア文字を含むことができます。"**

プロパティー	デフォルト	説明
slot.drop.on.stop	**false**	コネクター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、コネクター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ーに設定された状態を保持します。コネクターが再起動すると、同じレプリケーションスロットがあるため、コネクターは停止した場所から処理を開始できます。 テストまたは開発環境でのみ **true** 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。コネクターが再起動すると、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。
publication.name	**dbz_publication**	**pgoutput** の使用時に変更をストリーミングするために作成される PostgreSQL パブリケーションの名前。 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。 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。
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。

プロパティー	デフォルト	説明
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。
database.server.name	デフォルトなし	Debezium が変更をキャプチャーする特定の PostgreSQL データベースサーバーまたはクラスターの namespace を識別および提供する論理名。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。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターから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。
schema.include.list	デフォルトなし	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。また、**schema.exclude.list** プロパティーも設定しないでください。
schema.exclude.list	デフォルトなし	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。また、**schema.include.list** プロパティーも設定しないでください。
table.include.list	デフォルトなし	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)。**table.include.list** に含まれていないテーブルの変更はキャプチャーされません。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。また、**table.exclude.list** プロパティーを設定しないでください。
table.exclude.list	デフォルトなし	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)。**table.exclude.list** に含まれていないテーブルは、その変更がキャプチャされます。各識別子の形式は schemaName.tableName です。また、**table.include.list** プロパティーを設定しないでください。

プロパティー	デフォルト	説明
column.include.list	デフォルトなし	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。また、**column.exclude.list** プロパティーも設定しないでください。
column.exclude.list	デフォルトなし	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。また、**column.include.list** プロパティーも設定しないでください。
time.precision.mode	**adaptive**	時間、日付、およびタイムスタンプは、異なる精度の種類で表すことができます。 **adaptive** 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。 **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。 **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時間的価値 を参照します。

プロパティー	デフォルト	説明
decimal.handling.mode	**precise**	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。 **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。 **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。Decimal types を参照してください。
hstore.handling.mode	**map**	コネクターによる **hstore** 列の値の処理方法を指定します。 **map** は **MAP** を使用して値を表します。 **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。Postgre SQL**HSTORE** タイプを参照してください。
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。 **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。以下に例を示します。**P1Y2M3DT4H5M6.78S**。PostgreSQL basic types を参照してください。

プロパティー	デフォルト	説明
database.sslmode	**disable**	PostgreSQL サーバーへの暗号化された接続を使用するかどうか。オプションには以下が含まれます。 **disable** は暗号化されていない接続を使用します。 **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。 **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。 **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 PostgreSQL のドキュメント を参照してください。
database.sslcert	デフォルトなし	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 PostgreSQL のドキュメント を参照してください。
database.sslkey	デフォルトなし	クライアントの SSL 秘密鍵が含まれるファイルへのパス。詳細は PostgreSQL のドキュメント を参照してください。
database.sslpassword	デフォルトなし	**database.sslkey** で指定されたファイルからクライアントの秘密鍵にアクセスするためのパスワード。詳細は PostgreSQL のドキュメント を参照してください。
database.sslrootcert	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 PostgreSQL のドキュメント を参照してください。
database.tcpKeepAlive	**true**	TCP keep-alive プローブを有効にして、データベース接続がまだ有効であることを確認します。詳細は PostgreSQL のドキュメント を参照してください。

プロパティー	デフォルト	説明
tombstones.on.delete	**true**	廃棄 (tombstone) イベントの後に**削除**イベントが続くかどうかを制御します。 **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。 **false** - delete イベントのみが出力されます。 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。
column.truncate.to._length_.chars	**該当なし**	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 schemaName.tableName.columnName です。変更イベントレコードでは、これらの列の値がプロパティー名の **長さ** によって指定される文字数よりも長い場合は切り捨てられます。単一の設定で、異なる長さを持つ複数のプロパティーを指定できます。長さは正の整数である必要があります (例:**column.truncate.to.20.chars**)。
column.mask.with._length_.chars	**該当なし**	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 schemaName.tableName.columnName です。変更イベント値では、指定のテーブルコラムの値はアスタリスク (*) の **長さ** (数) に置き換えられます。単一の設定で、異なる長さを持つ複数のプロパティーを指定できます。長さは正の整数またはゼロでなければなりません。ゼロを指定すると、コネクターは値を空の文字列に置き換えます。

プロパティー	デフォルト	説明
column.mask.hash.hashAlgorithm.with.salt.salt	**該当なし**	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 **schemaName.tableName.columnName** です。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。 仮名は、指定された **hashAlgorithm** と **salt** 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、参照整合性は維持され、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 section に説明されています。 以下の例では、**CzQMA0cB5K** が無作為に選択された salt になります。 `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` 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。 使用される **hashAlgorithm**、選択された **salt**、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。

プロパティー	デフォルト	説明
column.propagate.source.type	該当なし	列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。列の完全修飾名の形式は、databaseName.tableName.columnName または databaseName.schemaName.tableName.columnName です。 コネクターは指定された各列に対して、列の元の型と元の長さをパラメーターとして、出力された変更レコードの対応するフィールドスキーマに追加します。以下の追加されたスキーマパラメーターは、元の型名と可変幅型の元の長さを伝播します。 **__debezium.source.column.type** + **__debezium.source.column.length** + **__debezium.source.column.scale** このプロパティーは、シンクデータベースの対応するコラムのサイズを適切に調整する場合に便利です。
datatype.propagate.source.type	該当なし	一部の列のデータベース固有のデータ型名と一致する正規表現のコンマ区切りリスト (任意)。完全修飾データ型名の形式は、databaseName.tableName.typeName または databaseName.schemaName.tableName.typeName です。 これらのデータタイプでは、コネクターは出力された変更レコードの対応するフィールドスキーマにパラメーターを追加します。追加されたパラメーターは、列の元の型と長さを指定します。 **__debezium.source.column.type** + **__debezium.source.column.length** + **__debezium.source.column.scale** これらのパラメーターは、それぞれ可変幅型の列の元の型名と長さを伝播します。このプロパティーは、シンクデータベースの対応する列のサイズを適切に調整するのに便利です。 list of PostgreSQL-specific data type namesを参照してください。

プロパティー	デフォルト	説明
message.key.columns	空の文字列	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。 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。 テーブルのカスタムメッセージキーを作成するには、テーブルとメッセージキーとして使用する列をリストします。各リストエントリーは以下の形式をとります。 **<fully-qualified_tableName>:_<keyColumn>_,<keyColumn>** 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。 **<schemaName>.<tableName>** プロパティーには複数のテーブルのエントリー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ーを区切ります。 以下の例では、テーブル **inventory.customers** と **purchase.orders** にメッセージキーを設定しています。 **inventory.customers:pk1,pk2; (.*).purchaseorders:pk3,pk4** テーブル **inventory.customer** では、列 **pk1** と **pk2** がメッセージキーとして指定されています。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。 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。

プロパティー	デフォルト	説明
publication.autocreate.mode	all_tables	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。可能な設定: **all_tables** - コネクターはパブリケーションが存在すればそれを使用します。パブリケーションが存在しない場合は、コネクター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。レプリケーションを実行する権限を持つデータベースユーザーには、パブリケーションを作成する権限も必要です。これは **CREATE PUBLICATION <publication_name> FOR ALL TABLES;**. **disabled**で許可されます。コネクター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、コネクター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。コネクターがパブリケーションを見つけられない場合は、コネクターは例外を出力し、停止します。 **filtered**: パブリケーションが存在する場合、コネクター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、**database.exclude.list**、**schema.include.list**、**schema.exclude.list**、**table.include.list** の各コネクター設定プロパティー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。以下に例を示します。**CREATE PUBLICATION <publication_name> FOR TABLE <tbl1, tbl2, tbl3>**.
binary.handling.mode	bytes	バイナリー (**bytea**) 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。 **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。 **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。

プロパティー	デフォルト	説明
truncate.handling.mode	bytes	**TRUNCATE** イベントを伝播すべきかどうかを指定します (Postgr 11 以降で **pgoutput** プラグインを使用する場合のみ利用可能)。 **skip** を指定すると、これらのイベントが省略されます (デフォルト)。 **include** を指定すると、これらのイベントが含まれます。 **切り捨て (truncate)** イベントの構造とそれらの順序付けセマンティクスについては、**切り捨て (truncate) イベント** を参照してください。

以下の **高度な** 設定プロパティーには、ほとんどの状況で機能する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7.24 高度なコネクター設定プロパティー

プロパティー	デフォルト	説明

プロパティー	デフォルト	説明
snapshot.mode	**Initial**	コネクターの起動時にスナップショットを実行する基準を指定します。 **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。 **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。 **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** snapshot モードを使用すると、関心のあるすべてのデータはまだ WAL に反映されている知っているときにのみ有効です。 **initial_only**: コネクターは初期スナップショットを実行してから停止しますが、その後の変更は処理されません。 **exported**: 非推奨 スナップショットモード設定のリファレンステーブルに詳細があります。
snapshot.include.collection.list	**table.include.list**に指定したすべてのテーブル	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** プロパティーで名前を付ける必要があります。このプロパティーは、コネクターの **snapshot.mode** プロパティーが **never** 以外の値に設定されている場合にのみ有効になります。 このプロパティーは増分スナップショットの動作には影響しません。

プロパティー	デフォルト	説明
snapshot.lock.timeout.ms	10000	スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する正の整数値。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。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。

プロパティー	デフォルト	説明
snapshot.select.statement.overrides	デフォルトなし	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、プロパティーを使用します。このプロパティー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。 プロパティーには、**<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、 **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** 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ーを追加して、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ーの名前 (**snapshot.select.statement.overrides.<schemaName>.<tableName>**) を指定します。例: **snapshot.select.statement.overrides.customers.orders**. 例: 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、以下のプロパティーを追加します。 "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC"

プロパティ	デフォルト	説明
event.processing.failure.handling.mode	fail	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る処理中にコネクターが例外に反応する方法を指定します。 **fail** は例外を伝播し、問題のあるイベントのオフセットを示し、コネクターを停止させます。 **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。 **skip** は問題のあるイベントを省略し、処理を継続します。
max.queue.size	20240	ブロッキングキューの最大サイズの正の整数値。コネクターは、Kafka に書き込む前にストリーミングレプリケーションから受信される変更イベントをブロッキングキューに配置します。このキューは、たとえば Kafka へのレコードの書き込みが遅い場合や Kafka が利用できない場合などにバックプレシャーを提供できます。
max.batch.size	10240	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。
max.queue.size.in.bytes	0	ブロッキングキューの最大サイズ (バイト単位) の long 値。この機能はデフォルトで無効になっています。正の long 値が設定されると有効になります。
poll.interval.ms	1000	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 1000 ミリ秒 (1 秒) です。

プロパティー	デフォルト	説明
include.unknown.datatypes	**false**	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。 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、このプロパティーを **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** プロパティーを設定すると、正確な表現を制御できます。 **注記** **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になります。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になります。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。

プロパティー	デフォルト	説明
database.initial.statements	デフォルトなし	データベースへの JDBC 接続を確立するときにコネクターが実行する 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。 コネクターは 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、このプロパティー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。 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、コネクターはこれらのステートメントを実行しません。
+status.update.interval.ms	**10000**	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、このプロパティーは、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。

プロパティー	デフォルト	説明
heartbeat.interval.ms	0	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。 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、このプロパティーを、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。 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。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。
heartbeat.topics.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次のようになります。 <heartbeat.topics.prefix>.<server.name> たとえば、データベースサーバー名が **fulfillment** の場合、デフォルトのトピック名は **__debezium-heartbeat.fulfillment** になります。

プロパティー	デフォルト	説明
heartbeat.action.query	デフォルトなし	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。 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ーすることで、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にハートビートテーブルを作成し、そのテーブルにレコードを挿入するステートメントにこのプロパティーを設定するなどの方法があります。 **INSERT INTO test_heartbeat_table (text) VALUES ('test_heartbeat')** これにより、コネクターはトラフィックの少ないデータベースから変更を受け取り、その LSN を確認することができ、データベースホスト上での WAL の無制限な増加を防ぐことができます。
schema.refresh.mode	**columns_diff**	テーブルのインメモリースキーマの更新をトリガーする条件を指定します。 **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ーまと常に同期されるようにします。 **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ーに指示します。 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、コネクター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。

プロパティー	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、このプロパティーは、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。このプロパティーは、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** プロパティーを 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)。

プロパティー	デフォルト	説明
toasted.value.placeholder	**__debezium_unavailable_value**	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。**toasted.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。 **このオプションは非推奨です。代わりにunavailable.value.placeholderを使用してください。**
unavailable.value.placeholder	**__debezium_unavailable_value**	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。
provide.transaction.metadata	**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、Transaction metadata を参照してください。
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	デフォルトなし	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。操作には、**c** (挿入/作成)、**u** (更新)、および **d** (削除) が含まれます。デフォルトでは、操作はスキップされません。
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。 コレクション名の指定には次の形式を使用します。 **<schemaName>.<tableName>** シグナル機能はテクノロジープレビュー機能です。

プロパティー	デフォルト	説明
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。 増分スナップショットはテクノロジープレビュー機能です。

パススルーコネクター設定プロパティー

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

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

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

Debezium PostgreSQL コネクターは、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ときにコネクター操作に関する情報を提供します。

Debezium モニター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。

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	型	説明

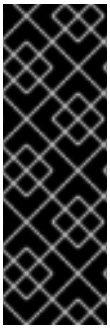
属性	型	説明
LastEvent	string	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除される予定ですので、代わりに 'CapturedTables' メトリクスを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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	型	説明
SnapshotAborted	boolean	スナップショットが中断されたかどうか。
SnapshotCompleted	boolean	スナップショットが完了したかどうか。
SnapshotDurationInSeconds	long	スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。
RowsScanned	Map<String, Long>	スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

コネクターは、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	型	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。

属性	型	説明
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。



重要

増分スナップショットの使用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 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。

7.7.2. Debezium PostgreSQL コネクター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	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルター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	型	説明
MonitoredTables **非推奨、今後のリリースで削除される予定ですので、代わりに 'CapturedTables' メトリクスを使用してください。**	**string[]**	コネクターによって監視されるテーブルの一覧。
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。
MaxQueueSizeInBytes	**long**	キューの最大バッファー (バイト単位)。
CurrentQueueSizeInBytes	**long**	キュー内のレコードの現在のデータ (バイト単位)。

7.8. DEBEZIUM POSTGRESQL 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
- PostgreSQL が使用不可能になる
- クラスターの障害
- 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。

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で、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ーのバックアップを、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ー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。新しいコネクタータスクは、以前のタスクが停止した場所でプロセスを開始します。コネクタータスクが正常に停止され、新しいプロセスで再起動されるまでの間、プロセスに短い遅延が発生します。

Kafka Connect プロセスのクラッシュ

Kafka Connector プロセスが予期せず停止した場合、最後に処理されたオフセットを記録せずに、実行中のコネクタータスクが終了します。Kafka Connect が分散モードで実行されている場合は、Kafka Connect は他のプロセスでこれらのコネクタータスクを再起動します。ただし、PostgreSQL コネクターは、以前のプロセスで最後に**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 コネクターは、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ーによって生成されるため、Kafka Connect フレームワークは、Kafka プロデューサー API を使用してこれらのイベントを記録します。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、コネクターを実行している 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、コネクタータスクは接続が再確立されるまで一時停止します。接続が再確立されると、コネクターは停止した場所から再開します。

コネクターの一定期間の停止

コネクターが正常に停止された場合、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。コネクターが再起動すると、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、コネクター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 コネクター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 コネクターがしばらく停止した後に再起動すると、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 コネクター

Debezium の SQL Server コネクターは、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。

このコネクターと互換性のある 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 コネクター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 コネクターの概要」
- 「Debezium SQL Server コネクターの仕組み」
- 「Debezium SQL Server コネクターのデータ変更イベントの説明」
- 「Debezium SQL Server コネクターによるデータ型のマッピング方法」
- 「Debezium コネクターを実行するための SQL Server のセットアップ」
- 「Debezium SQL Server コネクターのデプロイ」
- 「スキーマ変更後のキャプチャーテーブルの更新」
- 「Debezium SQL Server コネクターのパフォーマンスの監視」

Debezium SQL Server コネクターが 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。コネクターは、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。コネクターは、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。コネクターは、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 コネクターの概要

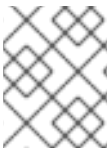
Debezium SQL Server コネクターは、SQL Server 2016 Service Pack 1 (SP1) およびそれ以降 の Standard エディションまたは Enterprise エディションで利用可能な 変更データキャプチャー (CDC) 機能に基づいています。SQL Server のキャプチャープロセスでは、指定のデータベースおよびテーブルを監視し、その変更をストアドプロシージャーファサードのある特別に作成された **変更テーブル** に格納します。

Debezium SQL Server コネクター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、コネクター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。コネクターは、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。

コネクターが 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、コネクター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、コネクター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 コネクターはフォールトトラレントです。コネクターは変更を読み取り、イベントを生成するため、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**)。コネクターが何らかの理由で停止した場合 (通信障害、ネットワークの問題、クラッシュなど)、コネクター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ーが停止した場合、コネクター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 コネクターの仕組み

Debezium SQL Server コネクターを最適に設定および実行するには、コネクター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。

コネクター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 コネクター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 コネクター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 コネクターによるスキーマ変更トピックの使用方法」
- 「Debezium SQL Server コネクター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 コネクターによって生成されたイベント」

8.2.1. Debezium SQL Sever 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

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 コネクター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。

コネクター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、コネクター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、コネクターが最初に起動すると、データベースの最初の **整合性スナップショット** が実行されます。この初期スナップショットは、コネクター用に設定された**include** プロパティーおよび **exclude** プロパティー (**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。

コネクター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** 設定プロパティーによって決定されます。

3. サーバーのトランザクションログ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。

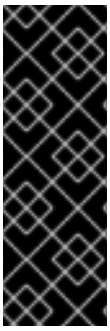
5. 必要な場合は、ステップ 2 で取得したロックを解放します。ほとんどの場合、ロックは短期間のみ保持されます。

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. コネクターオフセットにスナップショットの正常な完了を記録します。

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、コネクター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



重要

アドホックスナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。通常の状況では、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。

既存のテーフルのアドホックスナップショットを開始すると、コネクターはテーフ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。

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**	スナップショットを作成するテーブルの完全修飾名が含まれる配列。 名前の形式は **signal.data.collection** 設定オプションと同じです。

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

重要

増分スナップショットの使用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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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、ストリーム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ーに提供します。スナップショットウィンドウ中に、バッファー内の **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、バッファーされた **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、バッファーに含まれるのは、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。

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。SQL **INSERT** クエリーとしてテーブルにシグナルを送信します。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、テーブルを一覧表示する **data-collections**アレイを指定します (例: **{"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。

前提条件

- シグナルが有効になっている。
  - シグナルデータコレクションがソースのデータベースに存在し、コネクターはこれをキャプチャーするように設定されています。
  - 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>_"}');
   ```

   以下に例を示します。

   -

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

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

以下の表では、これら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

値	説明
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。
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。
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。
data-collections	スナップショットに含めるテーブル名の配列を指定するシグナルの **data**フィールドの必須コンポーネント。 配列は、**signal.data.collection** 設定プロパティーにコネクターのシグナルテーブルの名前を指定するときに使用する形式で、完全修飾名別にテーブルを一覧表示します。
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。 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。

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。

以下に例を示します。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。
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。

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。

8.2.2. Debezium SQL Server コネクターによる変更データテーブルの読み取り方法

コネクター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履歴トピックに永続化します。その後、コネクターは各ソーステーブルの変更テーブルを特定し、以下の手順を完了します。

1. コネクターは、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. コネクターは、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します。この並べ替えの順序により、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. コネクターは、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. コネクターは最大 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、コネクター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。

コネクターは、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. 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 コネクターは、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。**<serverName>.<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。

serverName
: **database.server.name** コネクター設定プロパティーで指定したコネクターの論理名です。

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、コネクター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.dbo.products**
- **fulfillment.dbo.products_on_hand**
- **fulfillment.dbo.customers**
- **fulfillment.dbo.orders**

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。

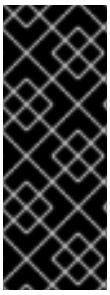
8.2.4. Debezium SQL Server コネクターによるスキーマ変更トピックの使用方法

CDC が有効になっている各テーブルについて、Debezium SQL Server コネクターは、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。コネクターは、スキーマ変更イベントをすべて **<serverName>** という名前の Kafka トピックに書き込みます。**serverName** は **database.server.name** 設定プロパティーに指定された論理サーバー名になります。

コネクター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ー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

コネクター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、内部データベース履歴トピックにも格納されます。内部データベース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。

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。

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。

以下に例を示します。SQL Server コネクター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": "1.7.2.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1588252618953,
      "snapshot": "true",
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": null,
      "commit_lsn": "00000025:00000d98:00a2",
      "event_serial_no": null
    },
    "databaseName": "testDB", (1)
    "schemaName": "dbo",
    "ddl": null, (2)
    "tableChanges": [ (3)
      {
```

```
"type": "CREATE", 4
"id": "\"testDB\".\"dbo\".\"customers\"", 5
"table": { 6
  "defaultCharsetName": null,
  "primaryKeyColumnNames": [ 7
    "id"
  ],
  "columns": [ 8
    {
      "name": "id",
      "jdbcType": 4,
      "nativeType": null,
      "typeName": "int identity",
      "typeExpression": "int identity",
      "charsetName": null,
      "length": 10,
      "scale": 0,
      "position": 1,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "first_name",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
      "typeExpression": "varchar",
      "charsetName": null,
      "length": 255,
      "scale": null,
      "position": 2,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "last_name",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
      "typeExpression": "varchar",
      "charsetName": null,
      "length": 255,
      "scale": null,
      "position": 3,
      "optional": false,
      "autoIncremented": false,
      "generated": false
    },
    {
      "name": "email",
      "jdbcType": 12,
      "nativeType": null,
      "typeName": "varchar",
```

```
            "typeExpression": "varchar",
            "charsetName": null,
            "length": 255,
            "scale": null,
            "position": 4,
            "optional": false,
            "autoIncremented": false,
            "generated": false
          }
        ]
      }
    }
  ]
}
}
```

表8.3 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	**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。
2	**ddl**	SQL Server コネクターの場合は常に **null** です。その他のコネクターでは、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 コネクターでは使用できません。
3	**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。
4	**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
5	**id**	作成、変更、または破棄されたテーブルの完全な識別子。
6	**table**	適用された変更後のテーブルメタデータを表します。
7	**primaryKeyColumnNames**	テーブルのプライマリーキーを設定する列のリスト。
8	**列**	変更されたテーブルの各列のメタデータ。

コネクター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"
  }
}
```

8.2.5. Debezium SQL Server コネクターのデータ変更イベントの説明

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

```
    ...
  },
}
```

表8.4 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ー 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ー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

SQL Server 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。

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」

8.2.5.1. 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.dbo.customers.Key" 4
   },
   "payload": { 5
      "id": 1004
   }
}
```

表8.5 変更イベントキーの説明

項目	フィールド名	説明

項目	フィールド名	説明
1	**schema**	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。
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.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。 • **server1** はこのイベントを生成したコネクターの名前です。 • **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。

8.2.5.2. 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": "server1.dbo.customers.Value", 2
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
```

```
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "server1.dbo.customers.Value",
        "field": "after"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": false,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
            "field": "connector"
          },
          {
            "type": "string",
            "optional": false,
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_ms"
          },
          {
            "type": "boolean",
            "optional": true,
            "default": false,
            "field": "snapshot"
          },
          {
            "type": "string",
            "optional": false,
            "field": "db"
          },
          {
            "type": "string",
            "optional": false,
            "field": "schema"
          },
          {
            "type": "string",
            "optional": false,
            "field": "table"
          },
          {
            "type": "string",
            "optional": true,
```

```
            "field": "change_lsn"
          },
          {
            "type": "string",
            "optional": true,
            "field": "commit_lsn"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "event_serial_no"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.sqlserver.Source", (3)
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "server1.dbo.customers.Envelope" (4)
  },
  "payload": { (5)
    "before": null, (6)
    "after": { (7)
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "source": { (8)
      "version": "1.7.2.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729468470,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000758:0003",
      "commit_lsn": "00000027:00000758:0005",
      "event_serial_no": "1"
    },
    "op": "c", (9)
```

```
    "ts_ms": 1559729471739 10
  }
}
```

表8.6 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のスキーマを指定します。 **server1.dbo.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.databaseSchemaName.tableName.Value** の形式を取るので、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 スキーマには独自の進化と履歴があります。
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**	**server1.dbo.customers.Envelope** は、ペイロードの全体的な構造のスキーマで、**server1** はコネクター名、**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。
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**	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。 ● 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
9	**op**	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。 ● **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**	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。 **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。

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。

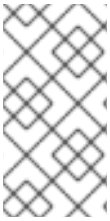
```
{
  "schema": { ... },
  "payload": {
    "before": { 1
```

```
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "john.doe@example.org"
    },
    "after": { (2)
      "id": 1005,
      "first_name": "john",
      "last_name": "doe",
      "email": "noreply@example.org"
    },
    "source": { (3)
      "version": "1.7.2.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559729995937,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000ac0:0002",
      "commit_lsn": "00000027:00000ac0:0007",
      "event_serial_no": "2"
    },
    "op": "u", (4)
    "ts_ms": 1559729998706 (5)
  }
}
```

表8.7 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**	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。 ● 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 **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。 ● 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**	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。
5	**ts_ms**	コネクターがイベントを処理した時間を表示する任意のフィールド。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。 **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。



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、**削除** イベント、行の古いキーを持つ 廃棄 (tombstone) イベント、および行の新しいキーを持つ **作成** 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 **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": "1.7.2.Final",
      "connector": "sqlserver",
      "name": "server1",
      "ts_ms": 1559730445243,
      "snapshot": false,
      "db": "testDB",
      "schema": "dbo",
      "table": "customers",
      "change_lsn": "00000027:00000db0:0005",
      "commit_lsn": "00000027:00000db0:0007",
      "event_serial_no": "1"
    },
    "op": "d", (3)
    "ts_ms": 1559730450205 (4)
  }
}
```

表8.8 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**	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。 ● 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
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**	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。 **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。

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。

8.2.6. 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。

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** および **event_count** のペアの配列。

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。

以下に例を示します。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",
  "event_count": null,
  "data_collections": null
}

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

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

トランザクションイベントは、**<database.server.name>.transaction** という名前のトピックに書き込まれます。

8.2.6.1. 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"
  }
}
```

8.2.7. Debezium SQL Server コネクターによるデータ型のマッピング方法

Debezium SQL Server コネクターは、行が存在するテーブルのように構造化されたイベントを生成して、テーブル行データへの変更を表します。各イベントには、行のコラム値を表すフィールドが含まれます。イベントが操作のコラム値を表す方法は、列の SQL データ型によって異なります。このイベントで、コネクターは各 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。

コネクターは SQL Sever 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、コネクターによる基本的な SQL Server データ型のマッピング方法を示しています。

表8.9 SQL Server コネクター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
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

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

SQL Server 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。
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。

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
DATETIME	**INT64**	**io.debezium.time.Timestamp** エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。
SMALLDATETIME	**INT64**	**io.debezium.time.Timestamp** エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。
DATETIME2(0), **DATETIME2(1)**, **DATETIME2(2)**, **DATETIME2(3)**	**INT64**	**io.debezium.time.Timestamp** エポックからの経過時間をミリ秒で表し、タイムゾーン情報は含まれません。
DATETIME2(4), **DATETIME2(5)**, **DATETIME2(6)**	**INT64**	**io.debezium.time.MicroTimestamp** エポックからの経過時間をマイクロ秒で表し、タイムゾーン情報は含まれません。
DATETIME2(7)	**INT64**	**io.debezium.time.NanoTimestamp** エポックからの経過時間をナノ秒で表し、タイムゾーン情報は含まれません。

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

SQL Server 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。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。

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
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ー の設定にしたがって 10 進数を処理します。

decimal.handling.mode=precise

表8.10 decimal.handing.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)
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。

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)
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8.11 decimal.handing.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8.12 decimal.handing.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**
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ー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。

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 キャプチャー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 のキャプチャージョブエージェントの設定パラメーター」

CDC の適用後、CDD 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、システムストアドプロシージャーを実行して CDC を有効にします。システムストアドプロシージャーは、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ー **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。

   以下に例を示します。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ー **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ーは、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** 固定データベー出力ルのメンバーである。

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ー **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。

   以下に例を示します。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', (1)
```

```
@role_name     = N'MyRole', (2)
@filegroup_name = N'MyDB_CT',(3)
@supports_net_changes = 0
GO
```

(1)(1)(1)(1)(1)(1)(1)キャプチャーするテーブルの名前を指定します。

(2)(2)(2)(2)(2)(2)(1)(2)ソーステーブルのキャプチャされた列に対する **SELECT** 権限を付与したいユーザーを追加できるロール **My Role** 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。**sysadmin** または **db_owner** のメンバーだけがキャプチャされた情報に完全にアクセスできるようにするには、**@role_name** の値を **NULL** に設定します。

(3)(3)(3)(3)(3)(2)(3)キャプチャしたテーブルの変更テーブルを SQL Server が配置する **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ーは、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**My DB**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。

   以下に例を示します。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 ではテストされていません。

8.3.5. SQL Server キャプチャー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、キャプチャー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。同期を即時に行うとパフォーマンスに影響します。キャプチャー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ー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 のキャプチャージョブエージェントの設定パラメーター

SQL Server では、キャプチャー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。キャプチャー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ーを実行し、新しい値を指定することで、キャプチャージョブ設定を調整し、CPU の負荷を軽減します。

注記

SQL Server のキャプチャー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。キャプチャー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

- キャプチャージョブが、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 コンテナー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ーの説明」

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 コンテナーイメージをビルドすることです。

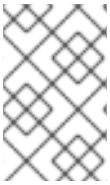
デプロイメントプロセス中に、以下のカスタムリソース (CR) を作成し、使用します。

- 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**コンテナーイメージを格納する場所を指定します。コンテナーイメージは 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

- AMQ Streams on OpenShift の使用の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ー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 にデプロイするの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 コンテナー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 コンテナー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ーは **.spec.build.output** に指定されるコンテナーレジストリーにプッシュされ、Kafka Connect Pod 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必要があります。
- AMQ Streams Operator が稼働している必要があります。
- Kafka クラスターは、Apache Open Shift での AMQ ストリームのデプロイとアップグレード に記載されているようにデプロイされます。
- Red Hat Integration ライセンスがある。

- Kafka Connect is deployed on AMQ Streams 。
- 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が必要であるか、ImageStream リソースを作成する必要があります。

  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には、以下を実行します。

  - 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 として保存します。

  - 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。

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、以下の例のように **metadata.annotations** および **spec.build** プロパティーを指定する **KafkaConnect** CR を作成します。**dbz-connect.yaml** などの名前でファイルを保存します。

例8.1 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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: debezium-kafka-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (1)
spec:
  version: 3.00
  build: (2)
    output: (3)
      type: imagestream (4)
      image: debezium-streams-connect:latest
    plugins: (5)
      - name: debezium-connector-sqlserver
        artifacts:
          - type: zip (6)
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-sqlserver/1.7.2.Final-redhat-<build_number>/debezium-connector-sqlserver-1.7.2.Final-redhat-<build_number>-plugin.zip (7)
          - type: zip
            url: https://maven.repository.redhat.com/ga/io/apicurio/apicurio-registry-distro-connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-2.0-redhat-<build-number>.zip
          - type: zip
```

```
            url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

表8.13 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、クラスターオペレーターが **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ー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 などのコンテナー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** の指定に関する詳細は、AMQ Streams Build スキーマ参照 のドキュメント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。

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

```
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ー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、**sqlserver-inventory-connector.yaml** として保存します。

例8.2 Debezium コネクターの 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 (1)
spec:
  class: io.debezium.connector.sqlserver.SqlServerConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: sqlserver.debezium-sqlserver.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_sqlserver (10)
    database.include.list: public.inventory (11)
```

表8.14 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 がデータベースに接続するユーザーアカウントの名前。

項目	説明
8	データベースユーザーアカウントのパスワード
9	変更をキャプチャーするデータベースの名前。
10	データベースインスタンスまたはクラスターの論理名。 指定の名前は英数字またはアンダースコアからのみ形成する必要があります。 論理名は、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、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 {context}-inventory-connector.yaml
```

コネクターは 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。コネクター Pod の準備ができると、Debezium が実行されます。

これで、Debezium SQL ディ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 コンテナーイメージをビルドして Debezium SQL Server コネクターのデプロイ

Debezium SQL Server コネクターをデプロイするには、Debezium コネクターアーカイブが含まれるカスタム Kafka Connect コンテナーイメージをビルドし、このコンテナーイメージをコンテナー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。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 コネクターを定義する **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 コネクターと連携するように 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 コネクターを実行するコンテナーを追加する予定のコンテナーレジストリー (**quay.io** や **docker.io**など) でコンテナー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 コンテナーを作成します。
   a. Debezium SQL Server コネクターアーカイブ をダウンロードします。
   b. Debezium SQL Server コネクターアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。以下に例を示します。

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

   c. **registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0** 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。たとえば、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。**my-plugins** はプラグインディレクトリーの名前に置き換えます。

```
cat <<EOF >debezium-container-for-sqlserver.yaml (1)
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./<my-plugins>/ /opt/kafka/plugins/ (2)
USER 1001
EOF
```

(1) (1) (1) 任意のファイル名を指定できます。

(2) (2) (2) **my-plugins** は、プラグインディレクトリーの名前に置き換えます。

   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。

   d. 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ー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ーイメージを構築します。

   e. カスタムイメージを quay.io などのコンテナーレジストリーまたは内部のコンテナーレジストリーにプッシュします。コンテナーレジストリーは、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
```

f. 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、以下の例のように **annotations** と **image** プロパティーを指定する **dbz-connect.yaml** という名前の KafkaConnect CR を作成します。

```
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)
```

(1) KafkaConnector リソースはこの Kafka Connect クラスター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、このプロパティーによって Cluster Operator の **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** 変数がオーバーライドされます。

g. 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 SQL Server 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、コネクターに使用できる設定プロパティーを使用して、**.yaml** ファイルに Debezium SQL Server コネクターを設定します。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コラムで 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ー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 コネクターを設定します。このホストには、**testDB** という名前のデータベース、名前が **customers** というテーブルがあり、**fulfillment** がサーバーの論理名です。

SQL Server fulfillment-connector.yaml

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
```

```
  name: fulfillment-connector 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.dbname: testDB 7
    database.server.name: fullfullment 8
    database.include.list: dbo.customers 9
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 10
    database.history.kafka.topic: dbhistory.fullfillment 11
```

表8.15 コネクター設定の説明

項目	説明
1	Kafka Connect サービスに登録する場合のコネクターの名前。
2	この SQL Server コネクタークラスの名前。
3	SQL Server インスタンスのアドレス。
4	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	namespace を形成する SQL Server インスタンス/クラスターの論理名で、コネクターが書き込む Kafka トピックの名前、Kafka Connect スキーマ名、および 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 namespace のすべてに使用されます。
9	Debezium が変更をキャプチャーする必要があるすべてのテーブルのリスト。
10	DDL ステートメントをデータベース履歴トピックに書き込み、復元するためにコネクターによって使用される Kafka ブローカーのリスト。
11	コネクターが DDL ステートメントを書き、復元するデータベース履歴トピックの名前。このトピックは内部使用のみを目的としており、コンシューマーが使用しないようにしてください。

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 **fulfillment-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 fulfillment-connector.yaml
```

上記のコマンドは **fulfillment-connector** を登録し、コネクターは **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 コネクターは 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ー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_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

  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。

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---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   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。

- 端末画面で、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_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_conne
ctor_sqlserver.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
_sqlserver.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
_sqlserver.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.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":"transacti
on"}],"optional":false,"name":"inventory_connector_sqlserver.inventory.products_on_hand.E
nvelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.7.2.Final-redhat-
00001","connector":"sqlserver","name":"inventory_connector_sqlserver","ts_ms":163898524
```

7805,"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,"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。

Debezium SQL Server コネクターに設定できる設定プロパティーの完全リストは SQL Server コネクタープロパティー 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 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。

8.4.4. Debezium SQL Server コネクター設定プロパティーの説明

Debezium SQL Server 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ーが多数あります。多くのプロパティーにはデフォルト値があります。

プロパティーに関する情報は、以下のように設定されています。

- 必要なコネクター設定プロパティー
- 高度なコネクター設定プロパティー
- 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設定プロパティー。
  - パススルーデータベースの履歴プロパティー
- 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ー

Debezium SQL Server コネクター設定プロパティー (必須)

以下の設定プロパティーは、デフォルト値がない場合は**必須**です。

プロパティー	デフォルト	説明
name	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。(このプロパティーはすべての 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。SQL Server コネクターには、常に **io.debezium.connector.sqlserver.SqlServerConnector** の値を使用してください。

プロパティー	デフォルト	説明
tasks.max	**1**	このコネクターのために作成する必要のあるタスクの最大数。SQL Server コネクターは常に単一のタスクを使用するため、この値を使用しません。そのため、デフォルト値は常に許容されます。
database.hostname	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。
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.dbname	デフォルトなし	変更内容をストリームする SQL Server データベースの名前 **database.names** と一緒に使ってはいけません。
database.names	デフォルトなし	変更をストリームする SQL Server データベースの名前をコンマで区切って指定します。現在は、1 つのデータベース名のみサポートしています。**database.dbname** と併用しないでください。 このオプションは**実験的な**もので、本番では使用しないでください。これを使用すると、コネクターの動作がデフォルトの設定と互換性がなく、アップグレードやダウングレードのパスがありません。 ● コネクターは、コミットされたオフセットメッセージに異なるキーを使用します。 ● **snapshot.select.statement.overrides** で使用する SQL 文では、完全修飾テーブル名の一部としてデータベース名を使用する必要があります。

プロパティー	デフォルト	説明
database.server.name	デフォルトなし	Debezium がキャプチャーする SQL Server データベースサーバーの namespace を識別および提供する論理名。論理名は、他のコネクター全体で一意となる必要があります。これは、このコネクターから生成されるすべての Kafka トピック名の接頭辞として使用されるためです。使用できる文字は、英数字、ハイフン、ドット、アンダースコアのみです。
table.include.list	デフォルトなし	Debezium が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)。**table.include.list** に含まれていないテーブルはキャプチャーから除外されます。各識別子の形式は schemaName.tableName です。デフォルトでは、コネクターは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。**table.exclude.list** と併用しないでください。
table.exclude.list	デフォルトなし	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と一致する正規表現のコンマ区切りリスト (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 schemaName.tableName です。**table.include.list** と併用しないでください。
column.include.list	**空の文字列**	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 schemaName.tableName.columnName 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。また、**column.exclude.list** プロパティーも設定しないでください。
column.exclude.list	**空の文字列**	変更イベントメッセージの値から除外される必要がある列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。列の完全修飾名の形式は schemaName.tableName.columnName です。プライマリーキー列は、値から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。また、**column.include.list** プロパティーも設定しないでください。

プロパティー	デフォルト	説明
column.mask.hash.hashAlgorithm.with.salt.salt	該当なし	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 `<schemaName>.<tableName>._<columnName>` です。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。 仮名は、指定された **hashAlgorithm** と **salt** 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、参照整合性は維持され、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 section に説明されています。 以下の例では、**CzQMA0cB5K** が無作為に選択された salt になります。 `column.mask.hash.SHA-256.with.salt.CzQMA0cB5K = inventory.orders.customerName, inventory.shipment.customerName` 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。 使用される **hashAlgorithm**、選択された **salt**、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。
time.precision.mode	**adaptive**	時間、日付、およびタイムスタンプは、異なる精度の種類で表すことができます。**adaptive** (デフォルト) は、データベース列の型を基にして、ミリ秒、マイクロ秒、またはナノ秒の精度値のいずれかを使用して、データベースの値と全く同じように時間とタイムスタンプをキャプチャーします。**connect** は、Kafka Connect の Time、Date、および Timestamp の組み込み表現を使用して、常に時間とタイムスタンプ値を表し、データベース列の精度に関わらず、ミリ秒の精度を使用します。時間的価値 を参照します。

プロパティー	デフォルト	説明
decimal.handling.mode	**precise**	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。 **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。
include.schema.changes	**true**	コネクターがデータベーススキーマの変更を、データベースサーバー ID と同じ名前の Kafka トピックに公開するかどうかを指定するブール値。各スキーマの変更は、データベース名が含まれるキーと、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。
tombstones.on.delete	**true**	廃棄 (tombstone) イベントの後に**削除**イベントが続くかどうかを制御します。 **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。 **false** - delete イベントのみが出力されます。 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。
column.truncate.to._length_.chars	**該当なし**	フィールド値が指定された文字数より長い場合に、変更イベントメッセージ値で値を省略する必要がある文字ベースの列の完全修飾名と一致する正規表現のコンマ区切りリスト (任意)。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数である必要があります。列の完全修飾名の形式は schemaName.tableName.columnName です。

プロパティー	デフォルト	説明
column.mask.with._length_.chars	該当なし	文字ベースの列の完全修飾名にマッチする正規表現のコンマ区切りリスト (オプション) で、変更イベントメッセージの値を、指定された数のアスタリスク (*) 文字で設定されるフィールド値に置き換える必要があります。長さが異なる複数のプロパティーを単一の設定で使用できますが、それぞれの長さは正の整数またはゼロである必要があります。列の完全修飾名の形式は schemaName.tableName.columnName です。
column.propagate.source.type	該当なし	出力された変更メッセージの該当するフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列の完全修飾名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。列の完全修飾名の形式は schemaName.tableName.columnName です。
datatype.propagate.source.type+	該当なし	出力された変更メッセージフィールドスキーマに元の型および長さをパラメーターとして追加する必要がある列のデータベース固有のデータ型名と一致する、正規表現のコンマ区切りリスト (任意)。スキーマパラメーター (**__debezium.source.column.type**、**__debezium.source.column.length**、および **__debezium.source.column.scale**) は、それぞれ元の型名と長さ (可変幅型) を伝播するために使用されます。シンクデータベースの対応する列を適切にサイズ調整するのに便利です。完全修飾データ型名の形式は schemaName.tableName.typeName です。SQL Server 固有のデータ型のリストは SQL Server データ型 を参照してください。

プロパティー	デフォルト	説明
message.key.columns	該当なし	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。 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。 テーブルのカスタムメッセージキーを作成するには、テーブルとメッセージキーとして使用する列をリストします。各リストエントリーは以下の形式をとります。 **<fully-qualified_tableName>:_<keyColumn>_,<keyColumn>** 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。 **<schemaName>.<tableName>** プロパティーには複数のテーブルのエントリー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ーを区切ります。 以下の例では、テーブル **inventory.customers** と **purchase.orders** にメッセージキーを設定しています。 **inventory.customers:pk1,pk2; (.*).purchaseorders:pk3,pk4** テーブル **inventory.customer** では、列 **pk1** と **pk2** がメッセージキーとして指定されています。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。 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。

プロパティー	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。**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。

高度な SQL Server コネクター設定プロパティー

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

プロパティー	デフォルト	説明
snapshot.mode	Initial	キャプチャーされたテーブルの構造 (および必要に応じてデータ) の最初のスナップショットを作成するモード。スナップショットが完了すると、コネクターはデータベースのやり直し (redo) ログから変更イベントの読み取りを続行します。以下の値がサポートされます。 • **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。
snapshot.include.collection.list	**table.include.list**に指定したすべてのテーブル	スナップショットに含めるテーブルの完全修飾名 (**<dbName>.<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 プロパティーで名前を付ける必要があります。このプロパティーは、コネクターの **snapshot.mode** プロパティーが **never** 以外の値に設定されている場合にのみ有効になります。 このプロパティーは増分スナップショットの動作には影響しません。

プロパティー	デフォルト	説明
snapshot.isolation.mode	repeatable_read	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。以下の値がサポートされます。 • **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。) **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。 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す。**read_uncommitted** の場合、データの整合性の保証はありません (一部のデータは損失または破損する可能性があります)。
event.processing.failure.handling.mode	**fail**	**fail** は、例外 (問題のあるイベントのオフセットを示す) を伝播させ、コネクターを停止させます。 **warn** は、問題のあるイベントをスキップし、問題のあるイベントのオフセットをログに記録します。 **skip** は、問題のあるイベントをスキップします。
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ーが待機する時間 (ミリ秒単位) を指定する正の整数値。デフォルトは 1000 ミリ秒 (1 秒) です。

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

プロパティー	デフォルト	説明
snapshot.fetch.size	2000	スナップショットの実行中に各テーブルから1度に読み取る必要がある行の最大数を指定します。コネクターは、このサイズの複数のバッチでテーブルの内容を読み取ります。デフォルトは 2000 です。
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。
snapshot.lock.timeout.ms	10000	スナップショットの実行時に、テーブルロックを取得するまで待つ最大時間 (ミリ秒単位) を指定する整数値。この時間間隔でテーブルロックを取得できないと、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)。 **0** に設定すると、コネクターがロックを取得できない場合、直ちに失敗します。値 **-1** は、無限に待つことを意味します。

プロパティー	デフォルト	説明
snapshot.select.statement.overrides	デフォルトなし	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、プロパティーを使用します。このプロパティー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。 プロパティーには、**<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、 **"snapshot.select.statement.overrides": "inventory.products,customers.orders"** 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ーを追加して、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ーの名前 (**snapshot.select.statement.overrides.<schemaName>.<tableName>**) を指定します。例: **snapshot.select.statement.overrides.customers.orders**. 例: 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、以下のプロパティーを追加します。 > "snapshot.select.statement.overrides": "customer.orders", "snapshot.select.statement.overrides.customer.orders": "SELECT * FROM [customers].[orders] WHERE delete_flag = 0 ORDER BY id DESC" 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。

プロパティー	デフォルト	説明
sanitize.field.names	コネクター設定が、Avro を使用するように **key.converter** または **value.converter** パラメーターを明示的に指定する場合は **true** です。それ以外の場合のデフォルトは **false** です。	Avro の命名要件に準拠するためにフィールド名がサニタイズされるかどうか。
provide.transaction.metadata	**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。
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	デフォルトなし	ストリーミング中にスキップされる操作タイプのコンマ区切りリスト。操作には、**c** (挿入/作成)、**u** (更新)、および **d** (削除) が含まれます。デフォルトでは、操作はスキップされません。
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次のフォーマットを使用します。 **<databaseName>.<schemaName>.<tableName>** シグナル機能はテクノロジープレビュー機能です。
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。

プロパティー	デフォルト	説明
max.iteration.transactions	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。

Debezium SQL Server コネクターデータベース履歴設定プロパティー

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 **database.history.*** プロパティー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 **database.history** プロパティーについて説明しています。

表8.16 コネクターデータベース履歴設定プロパティー

プロパティー	デフォルト	説明
database.history.kafka.topic		コネクターがデータベーススキーマの履歴を保存する Kafka トピックの完全名。
database.history.kafka.bootstrap.servers		Kafka クラスターへの最初の接続を確立するためにコネクターが使用するホストとポートのペアの一覧。このコネクションは、コネクターによって以前に保存されたデータベーススキーマ履歴の取得や、ソースデータベースから読み取られる各 DDL ステートメントの書き込みに使用されます。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。
database.history.kafka.recovery.poll.interval.ms	**100**	永続化されたデータのポーリングが行われている間にコネクターが起動/回復を待つ最大時間 (ミリ秒単位) を指定する整数値。デフォルトは 100 ミリ秒です。
database.history.kafka.recovery.attempts	**4**	エラーでコネクターのリカバリーが失敗する前に、コネクターが永続化された履歴データの読み取りを試行する最大回数。データが受信されなかった場合に最大待機する時間は、**recovery.attempts** x **recovery.poll.interval.ms** です。

プロパティー	デフォルト	説明
database.history.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。
database.history.store.only.monitored.tables.ddl **今後のリリースで非推奨になり、削除される予定です。代わりに database.history.store.only.captured.tables.ddl を使用してください。**	**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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 安全なデフォルトは **false** です。
database.history.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る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、は、不足しているデータが必要になるため、注意して **true** に設定してください。 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ー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履歴プロパティー

Debezium は、Kafka プロデューサーを使用して、データベース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。**database.history.producer.*** および **database.history.consumer.*** 接頭辞で始まるパススルー設定プロパティー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ー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ーデータベース履歴プロパティーは、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。

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

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

Debezium は、プロパティーを Kafka クライアントに渡す前に、プロパティー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ー および Kafka コンシューマー設定プロパティーの詳細は、Kafka のドキュメントを参照してください。

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

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ーは、接頭辞 **database.*** で始まります。たとえば、コネクターは **database.foobar=false** などのプロパティーを JDBC URL に渡します。

データベース履歴クライアントのパススループロパティー の場合のように、Debezium はプロパティー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。

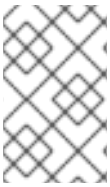
以下の方法のいずれかを使用すると、スキーマの変更後にキャプチャーテーブルを更新できます。

- 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カラムが、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。

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. 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 コネクターを起動します。

8. Debezium コネクター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** でストアドプロシージャー **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ーを実行し、更新元テーブルに新しいキャプチャ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ーを実行することで、古いキャプチャーテーブルを削除することができます。

以下に例を示します。データベーススキーマ変更後のオンラインスキーマ更新の実行

次の例は、**customers** ソーステーブルにカラム **phone_number** 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ーを実行して、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ー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]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
connect_1    | 2019-01-17 10:11:14,924 INFO   ||  Schema will be changed for ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
...
connect_1    | 2019-01-17 10:11:33,719 INFO   ||  Migrating schema to ChangeTable
[captureInstance=dbo_customers_v2, sourceTableId=testDB.dbo.customers,
changeTableId=testDB.cdc.dbo_customers_v2_CT, startLsn=00000025:00000ef8:0048,
changeTableObjectId=1749581271, stopLsn=NULL]
[io.debezium.connector.sqlserver.SqlServerStreamingChangeEventSource]
```

最終的には、**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 コネクターのパフォーマンスの監視

Debezium SQL Server コネクターは、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。コネクター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ーを監視するための、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 コネクターのスナップショットメトリクス

MBean は、**debezium.sql_server:type=connector-metrics,context=snapshot,server=<sqlserver.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	型	説明
LastEvent	**string**	コネクターが読み取りした最後のスナップショットイベント。
MilliSecondsSinceLastEvent	**long**	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	**long**	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	**long**	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	**string[]**	コネクターによって監視されるテーブルの一覧。
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	型	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。
SnapshotAborted	boolean	スナップショットが中断されたかどうか。
SnapshotCompleted	boolean	スナップショットが完了したかどうか。
SnapshotDurationInSeconds	long	スナップショットが完了したかどうかに関わらず、これまでスナップショットにかかった時間 (秒単位)。
RowsScanned	Map<String, Long>	スナップショットの各テーブルに対してスキャンされる行数が含まれるマップ。テーブルは、処理中に増分がマップに追加されます。スキャンされた 10,000 行ごとに、テーブルの完成時に更新されます。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。**max.queue.size.in.bytes** が正の long 値で渡された場合に有効になります。

属性	型	説明
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

コネクターは、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	型	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。

重要

増分スナップショットの使用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 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。

8.6.2. Debezium SQL Server コネクターのストリーミングメトリクス

MBeanは、**debezium.sql_server:type=connector-metrics,context=streaming,server=<sqlserver.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	型	説明

属性	型	説明
LastEvent	string	コネクターが読み取られた最後のストリーミングイベント。
MilliSecondsSinceLastEvent	long	コネクターが最新のイベントを読み取りおよび処理してからの経過時間 (ミリ秒単位)。
TotalNumberOfEventsSeen	long	前回の開始またはリセット以降にコネクターで確認されたイベントの合計数。
NumberOfEventsFiltered	long	コネクターに設定された include/exclude リストのフィルターリングルールによってフィルターされたイベントの数。
MonitoredTables **非推奨、今後のリリースで削除予定ですので、代わりに CapturedTables メトリクスを使用してください。**	string[]	コネクターによって監視されるテーブルの一覧。
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	型	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。
MaxQueueSizeInBytes	long	キューの最大バッファー (バイト単位)。
CurrentQueueSizeInBytes	long	キュー内のレコードの現在のデータ (バイト単位)。

8.6.3. Debezium SQL Server コネクター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。

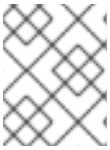
以下の表は、利用可能なスキーマ履歴メトリクスの一覧です。

属性	型	説明
Status	string	データベース履歴の状態を示す **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。

属性	型	説明
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 コネクター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ー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 コネクターメトリクス
- MongoDB コネクターメトリクス
- Postgre SQL コネクターのメトリクス
- SQL Server コネクターメトリクス
- Db2 コネクターメトリクス

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. OPENSHIFT 上での DEBEZIUM の監視

OpenShift 上で Debezium を使用している場合、JMX ポートを **9999** 番で開放することで JMX メトリクスを取得することができます。詳細は、Using AMQ Streams on OpenShift の JMX Options 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 のコネクターには、さまざまなログ機能が組み込まれています。ログの設定を変更して、ログに表示するメッセージやログの送信先を制御することができます。(Kafka、Kafka Connect、および Zookeeper と同様に) Debezium は Java の Log4j ログフレームワークを使用します。

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

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ー (通常は非常に大量のデータを扱う)

アペンダー

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

connect-log4j.properties

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

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

(1) (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) **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) **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。

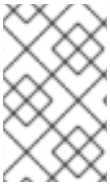
(4) (4) **stdout** アペンダーのパターン (詳しくは、Log4j ドキュメント を参照してください)。

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。

手順

1. **log4j.properties** ファイルを開きます。

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

   log4j.properties

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

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

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

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

   (3) (4) **additivity** を無効にします。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.relational.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、実行時にコネクター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、または 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 コネクター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 コネクター (および Kafka Connect ワーカー) は、複数のスレッドを使用してさまざまな動作を実行します。そのために、ログファイルを探し、特定の論理動作のログメッセージだけを識別するのが困難な場合があります。容易にログメッセージを探すことができるように、Debezium にはそれぞれのスレッドの追加情報を提供するさまざまな **マッピングされた診断コンテキスト** (MDC) が用意されています。

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

dbz.connectorType

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

dbz.connectorName

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

dbz.connectorContext

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

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

手順

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

   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'
   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,696 INFO   MySQL|dbserver1|snapshot  Snapshot is using user
   'debezium' with these MySQL grants:   [io.debezium.connector.mysql.SnapshotReader]
   2017-02-07 20:49:37,697 INFO   MySQL|dbserver1|snapshot   GRANT SELECT, RELOAD,
   SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO
   'debezium'@'%'   [io.debezium.connector.mysql.SnapshotReader]
   ...
   ```

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

10.4. OPENSHIFT での DEBEZIUM ログ

OpenShift で Debezium を使用している場合、Kafka Connect ロガーを使用して Debezium ロガーおよびログレベルを設定することができます。Kafka Connect スキーマでのロギングプロパティー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 コネクターの設定

Debezium コネクター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 コネクターが 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 コネクターが 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 コネクターへのシグナル送信

コネクター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** プロパティー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** プロパティーで、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、プロパティー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 ソースコネクター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 コネクター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。コネクター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ー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ー」
- 「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ー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** プロパティーによって制御されます。次の例に示すように、プロパティーのデフォルト値は **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 プロパティーの設定は、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ー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ーを指定し、続いてそれぞれのグループに適用するプロパティーを指定します。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。設定プロパティー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ー 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** プロパティーと **partitions** プロパティーを少なくとも **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ーの値を取得することができます。

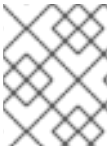
11.1.3. 自動的に作成されたトピックの設定

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ーに関するソースコネクターからの情報が必要です。それぞれの Debezium コネクターの設定で、トピックの作成を制御するプロパティーを定義します。Kafka Connect がコネクター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、そのコネクター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ーのセットが、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ー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ー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** プロパティーには、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** プロパティーの値を **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** プロパティー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ー

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ー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ー 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ー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、Installing Debezium on OpenShift の Kafka topic creation recommendations 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す。

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ー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ーを定義するには、以下の例に示すように、コネクターのカスタムリソースの **spec.config** にプロパティーを追加します。

```
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ー 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ー設定

項目	説明

項目	説明
1	**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。 **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。
2	**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。 **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。
3	**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** プロパティーにマッピングされ、ログの保存ポリシーを定義します。
4	**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** プロパティー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。

注記

カスタムグループは、必要な**replication.factor** および **partitions** プロパティーのみに対して、**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ー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、コネクターのカスタムリソースの **spec.config** に **topic.creation.<group_name>.include** プロパティーを追加し、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ーを定義します。
  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** プロパティーはオプションです。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** プロパティーはオプションです。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** プロパティー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** プロパティー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** プロパティー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、コネクターのカスタムリソースに **topic.creation.groups** プロパティーを追加し、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ー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。

以下に例を示します。デフォルトのトピック作成グループと 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 コネクターの設定

Debezium コネクターは 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 コネクター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ライズする。
3. レコードを正しい Kafka トピックに書き込む。

個々の Debezium コネクター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ライズする 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。Getting Started with Debezium guide 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、以下のコネクター設定プロパティーを **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ーを **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ライズ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 コネクターに不可欠です。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 シリアラ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容易に対応することができます。

Apache Avro シリアラ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は、Red Hat Integration - Service Registryのドキュメントを参照してください。

11.2.1. Service Registry の概要

Red Hat Integration - Service Registry は、Avro と連携する以下のコンポーネントを提供します。

- Debezium コネクター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ライズします。
- 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

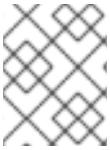
ループットが向上することを意味します。

- 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 コネクターを実行するのに使用している Kafka Connect コンテナーイメージに 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 コネクターのデプロイの概要

Avro シリアライゼーションを使用する Debezium コネクター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. Installing and deploying Service Registry on Open Shift の手順に従って、Red Hat Integration - Service Registry 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 コネクター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ーを設定して、Avro シリアライゼーションを使用するように Debezium コネクター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
```

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 コンテナーで Avro を使用するコネクターのデプロイ

ご使用の環境で、提供された Debezium コンテナーを使用して、Avro シリアライゼーションを使用する Debezium コネクター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 コンテナー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 コネクターを設定するには、以下の手順を完了します。

前提条件

- コンテナー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 コネクター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 コネクターのアーカイブを展開して、コネクタープラグインのディレクトリー構造を作成します。複数の Debezium コネクター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ー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 コネクターが含まれるディレクトリーに 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 コネクターディレクトリーにアーカイブを展開します。

   複数のタイプの Debezium コネクターを 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ータイプのディレクトリーにアーカイブを展開します。それぞれのディレクトリー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 コネクター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-30-rhel8:2.0.0** 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ーの名前に置き換えてください。

```
FROM registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Kafka Connect は、コネクター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 コンテナー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ー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. 新しいコンテナー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** プロパティーを編集します。このプロパティー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、新しいコンテナー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 コネクターをデプロイします。それぞれの Debezium コネクターについて、以下の設定を行います。

a. Debezium コネクター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 My SQL コネクター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
    database.server.name: dbserver1
    database.include.list: inventory
```

```
    database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092
    database.history.kafka.topic: schema-changes.inventory
    key.converter: io.apicurio.registry.utils.converter.AvroConverter
    key.converter.apicurio.registry.url: http://apicurio:8080/api
    key.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
    value.converter: io.apicurio.registry.utils.converter.AvroConverter
    value.converter.apicurio.registry.url: http://apicurio:8080/api
    value.converter.apicurio.registry.global-id:
io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy
```

b. コネクター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、コネクターが **inventory** データベースに対して実行されるようになります。

6. コネクターが作成され、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、コネクター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、シリアライズ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 コネクターの設定プロパティー **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、適合しないフィールドをシリアライズ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 コネクターを設定して、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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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 コネクターを設定するために、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 コネクターから出力される CloudEvents 変更イベントレコードを示しています。この例では、PostgreSQL コネクターは 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" : "1.7.2.Final",                    8
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578923739738",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "565",
  "iodebeziumlsn" : "29274832",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "565",                                9
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : {                                              10
    "before" : null,
    "after" : {
      "pk" : 1,
      "name" : "Bob"
    }
  }
}
```

1 1 1 変更イベントの内容に基づいてコネクター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、コネクター設定の **database.server.name** プロパティー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ータイプ。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 コネクターで有効にすると、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ーによって、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 コネクターから出力される CloudEvents 変更イベントレコードを示していま

す。この例でも、PostgreSQL コネクターは 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、ここではコネクターは **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;txId:578",
  "source" : "/debezium/postgresql/test_server",
  "specversion" : "1.0",
  "type" : "io.debezium.postgresql.datachangeevent",
  "time" : "2020-01-13T14:04:18.597Z",
  "datacontenttype" : "application/avro",          (1)
  "dataschema" : "http://my-registry/schemas/ids/1", (2)
  "iodebeziumop" : "r",
  "iodebeziumversion" : "1.7.2.Final",
  "iodebeziumconnector" : "postgresql",
  "iodebeziumname" : "test_server",
  "iodebeziumtsms" : "1578924258597",
  "iodebeziumsnapshot" : "true",
  "iodebeziumdb" : "postgres",
  "iodebeziumschema" : "s1",
  "iodebeziumtable" : "a",
  "iodebeziumtxId" : "578",
  "iodebeziumlsn" : "33227720",
  "iodebeziumxmin" : null,
  "iodebeziumtxid": "578",
  "iodebeziumtxtotalorder": "1",
  "iodebeziumtxdatacollectionorder": "1",
  "data" : "AAAAAAEAAgICAg=="                      (3)
}
```

(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 コネクター設定で **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",      (1)
```

```
"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、同じコネクター設定で **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。

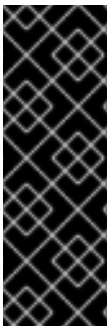
11.3.3. Debezium CloudEvents コンバーター設定オプション

CloudEvent コンバーターを使用するように Debezium コネクターを設定する場合、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	Arvo 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。

11.4. DEBEZIUM コネクターへのシグナル送信



重要

シグナル機能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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Debezium のシグナルメカニズムを使用すると、コネクター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。コネクター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上に作成したシグナリングテーブルは、Debezium との通信専用に指定されています。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 コネクターで使用可能です。

- Db2
- 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ーごとに、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、コネクター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 コネクター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。
   コネクター設定で、プロパティー **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、コネクターによって異なります。
   次の例では、各コネクターに使用する名前のフォーマットを示しています。

   Db2

   <schemaName>.<tableName>

   MySQL

   <databaseName>.<tableName>

   PostgreSQL

   <schemaName>.<tableName>

   SQL Server

   <databaseName>.<schemaName>.<tableName>

   signal.data.collection プロパティーの設定の詳細については、お使いのコネクターの設定プロパティーの表を参照してください。
4. モニターリングするテーブルのリストに、シグナリングテーブルを追加します。
   Debezium コネクターの設定で、手順 1 で作成したデータコレクションの名前を

table.include.list プロパティーに追加します。

table.include.listプロパティーの詳細については、お使いのコネクターの設定プロパティーの表を参照してください。

11.4.1.1. デベージア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、コネクター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ーで使用できますが、他の信号の種類は特定のコネクター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。

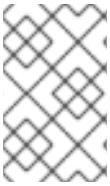
前提条件

- 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

- ソースデータベースに SQL クエリーを送信して、次の例のように必要な構造と一致するテーブルを作成します。

  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、コネクター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. デベージアムシグナルアクションの種類

シグナリングを使って、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

一部の信号はすべてのコネクター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ーを作成することで、ログにエントリーを追加するようコネクターに要求できます。シグナルの処理後、コネクター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

カラム	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。
data	**{"message":"Signal message at offset {}"}**	**message** パラメーターは、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 (**{}**) 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ーを作成することで、アドホックスナップショットの開始をコネクターに要求することができます。信号を処理した後、コネクター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。

コネクター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、コネクター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 コネクターで利用可能です。

- Db2
- MySQL
- 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

カラム	値
id	**d139b9b7-7777-4547-917d-e1775ea61d41**
type	**execute-snapshot**
data	**{"data-collections": ["public.MyFirstTable", "public.MySecondTable"]}**

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 コネクターのアドホックスナップショット
- My SQL コネクターのアドホックスナップショット
- Postgre SQL コネクター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

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 コネクターの増分スナップショット
- My SQL コネクターのインクリメンタルスナップショット
- Postgre SQL コネクターのインクリメンタルスナップショット
- SQL Server コネクターの増分スナップショット

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ーを設定することができます。また、Debezium SMT をシンクコネクターに適用して、コネクターが 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します。

メッセージフィルターリング 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、コネクター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ー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

コネクター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、コネクター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ーまたはシンクコネクター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 コネクターにこれらの SMT のいずれかを設定する場合、Kafka Connect はコネクター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、一部の変換を適用する場合があります。

たとえば、Debezium コネクターでは、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 コネクター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、シンクコネクター SMT に定義する述語に類似しています。コネクターは 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ーに渡します。

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ー (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ーはありません。
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 コネクターがソースデータベースで削除操作を処理すると、コネクター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、行の削除マーカーを表します。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ーが含まれる場合は、**RecordIsTombstone** 述語で SMT を使用して廃棄レコードをフィルターリングするようにシンクコネクターを設定できます。

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ーレコードの場合、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:フィルターと条件付き SMT
- 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 コネクター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ー設定で変換に 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 コネクター用に設定されている。

手順

1. Debezium コネクターの設定で、**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. コネクター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
      - RecordIsTombstone

- TopicNameMatches
  以下に例を示します。

```
predicates.IsOutboxTable.type=org.apache.kafka.connect.predicates.TopicNameMatch
```

b. 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.*
```

4. 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
```

前述のプロパティーは、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。

以下に例を示します。outbox イベントルーター変換のための Topic Name Match 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.predicates.TopicNameMatch
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** プロパティー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** プロパティー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** プロパティーが **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。このプロパティーを **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 **RecordIsTombstone** 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 つにまとまり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 コネクター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す。

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 コネクターの 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。

たとえば、**.properties** ファイルの設定は以下のようになります。

```
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。

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** プロパティーを **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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを 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ーを設定できます。

- 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
key.enforce.uniqueness	**true**	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。 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。
key.field.name	**__dbz__physicalTableIdentifier**	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。
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。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。
key.field.replacement		**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。

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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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。

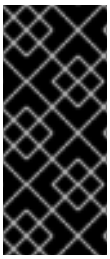
- 変換を事前にコンパイルし、それを Kafka Connect にデプロイする必要がある。
- 変更が生じるたびにコードの再コンパイルおよび再デプロイが必要になり、運用の柔軟性が失われる。

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の実装をダウンロードし、言語実装で使用されるその他の JAR ファイルと共に Debezium コネクタープラグインのディレクトリーに追加する必要があります。たとえば、Groovy 3 の場合は、https://groovy-lang.org/ からその JSR 223 実装をダウンロードすることができます。GraalVM JavaScript の JSR 223 実装は、https://github.com/graalvm/graaljs から入手することができます。

12.3.1. Debezium コンテンツベースのルーティング SMT の設定

セキュリティー上の理由から、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-1.7.2.Final.tar.gz** で提供されます。Debezium コネクタープラグインでコンテンツベースのルーティング 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。



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ー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。

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-1.7.2.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. 以下に例を示します。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 コネクターを設定するには、コネクターの 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もあります。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、コネクター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。

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	型
key	メッセージのキー。	**org.apache.kafka.connect.data .Struct**

名前	説明	型
value	メッセージの値。	**org.apache.kafka.connect.data .Struct**
keySchema	メッセージのキーのスキーマ。	**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
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、以下のプロパティーを公開します。 • **value** (タイプ: **Object**) • **schema** (タイプ: **org.apache.kafka .connect.data.Schema**)	**java.util.Map<String, io.debezium .transforms.scripting .RecordHeader>**

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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを 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ーを設定できます。

- 変換用の SMT 述語を設定する。
- SMT の 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 コネクターは、**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 コネクターでフィルター ContentBasedRouting SMT を使用するには、まず **ExtractNewDocumentState** SMT を適用してフィールドを解放する必要があります。

式の中で JSON パーサを使用する方法もあります。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

プロパティー	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**	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** メッセージに条件ロジックを適用します。

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機能は、最新の技術をいち早く提供し、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

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の実装をダウンロードし、言語実装で使用されるその他の JAR ファイルと共に Debezium コネクタープラグインのディレクトリーに追加する必要があります。たとえ

ば、Groovy 3 の場合は、https://groovy-lang.org/ からその JSR 223 実装をダウンロードすることができます。GraalVM JavaScript の JSR223 実装は、https://github.com/graalvm/graaljs から入手することができます。

12.4.1. Debezium フィルター SMT の設定

セキュリティー上の理由から、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-1.7.2.Final.tar.gz** で提供されます。Debezium コネクタープラグインでフィルター 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。

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ー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。

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-1.7.2.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. 以下に例を示します。Debezium ベーシックフィルターの SMT 設定

Debezium コネクターの 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 コネクターを設定するには、Debezium コネクターの 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、コネクター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。

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	型
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	メッセージのキーのスキーマ。	**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
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、以下のプロパティーを公開します。 • **value** (タイプ: **Object**) • **schema** (タイプ: **org.apache.kafka.connect.data.Schema**)	**java.util.Map<String, io.debezium.transforms.scripting.RecordHeader>**

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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを 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ーを設定できます。

- 変換用の SMT 述語を設定する。
- SMT の 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 コネクターは、**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 コネクターでフィルター SMT を使用するには、まず **ExtractNewDocumentState** SMT を適用してフィールドを解放する必要があります。

式の中で JSON パーサを使用する方法もあります。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

プロパティー	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ず、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 のデータ変更イベントは、さまざまな情報を提供する複雑な構造を持ちます。Debezium の変更イベントを伝える Kafka レコードには、このすべての情報が含まれています。ただし、Kafka エコシステムの一部では、フィールド名と値のフラットな構造の Kafka レコードが要求されます。この種の

レコードを提供するために、Debezium ではイベントフラット化単一メッセージ変換 (SMT) を利用することができます。Debezium の変更イベントが含まれる Kafka レコードよりも単純なフォーマットの Kafka レコードをコンシューマーが要求する場合に、この変換を設定します。

イベントフラット化変換は Kafka Connect SMT です。

この変換は、SQL データベースコネクターでのみ利用することができます。

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)
  - 変更を加えた操作
  - データベースや変更が加えられたテーブルの名前などのソース情報
  - 変更が加えられた時刻のタイムスタンプ
  - (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

例えば、**UPDATE** 変更イベントの構造の一部は次のようになります。

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

この複雑なフォーマットは、システムで発生する変更に関するほとんどの情報を提供します。しかし、その他のコネクターや Kafka エコシステムの他の要素では、通常、以下のような単純なフォーマットのデータが要求されます。

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

コンシューマーが必要とする Kafka レコードのフォーマットを提供するには、イベントフラット化 SMT を設定します。

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 コネクターまたは Debezium コネクターから出力されるメッセージを使用するシンクコネクターに、イベントフラット化 SMT を設定することができます。シンクコネクター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 を元のコネクターまたはシンクコネクター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、トゥームストーンレコードをドロップする代わりに、イベントフラット化 SMT を設定してトゥームストーン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

コネクターの設定に SMT 設定の詳細を追加して、Kafka Connect ソースコネクターまたはシンクコネクターに Debezium イベントフラット化 SMT を設定します。デフォルトの動作を得るためには、**.properties** ファイルで、以下のように指定します。

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

他の Kafka Connect のコネクター設定と同様に、**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ニング 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。

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 では、元の変更イベントメタデータを簡素化した Kafka レコードに追加することができます。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ー固有のフィールド

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にアンダースコアを挿入します。

DELETE 操作用のシンプルな 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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを SMT を選んで適用することが推奨されます。

SMT の選択的な適用方法の詳細は、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
drop.tombstones	**true**	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。

オプション	デフォルト	説明
delete.handling.mode	**drop**	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**none** または **rewrite** に設定します。 ストリームに変更イベントの記録を残す場合は、**none** を指定します。レコードには **"value": "null"** のみが含まれています。 **rewrite** を指定して変更イベントのレコードをストリームに残し、レコードを編集して、**before**フィールドにあったキーと値のペアを含む **value** フィールドを持ち、さらに **__deleted: true** を **value** 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。 **rewrite** 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することができます。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。
route.by.field		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。 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 コネクター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。 シンクコネクター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。

オプション	デフォルト	説明
add.fields.prefix	__ (ダブルアンダースコア)	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。
add.fields		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**)。 オプションとして、**<field name>:<new field name>** でフィールド名を上書きすることができます。例えば、以下のように、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。 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。 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。
add.headers.prefix	__ (ダブルアンダースコア)	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
add.headers		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: **source.ts_ms**)。 オプションとして、**<field name>:<new field name>** でフィールド名を上書きすることができます。例えば、以下のように、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。 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。 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。

12.6. 送信トレイパターンを使用する DEBEZIUM コネクターの設定

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。

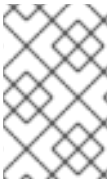
Debezium アプリケーションに送信トレイパターンを実装するには、Debezium コネクターを以下のように設定します。

- 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 コネクターは、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。

コネクター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ます。

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 コネクターと互換性がありません。

MongoDB ユーザーは、MongoDB outbox event router SMT を実行できます。

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」

12.6.1. 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 コネクターは、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": "1.7.2.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。

12.6.2. 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	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。 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、コネクター設定で **table.field.event.id** SMT オプションを設定します。

列	結果
aggregatetype	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 **${routedByValue}** 変数を置き換えます。 たとえば、デフォルト設定では、**route.by.field** SMT オプションは **aggregatetype** に設定され、**route.topic.replacement** SMT オプションは **outbox.event.${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。コネクターは、最初のレコードを **outbox.event.customers** トピックに出力します。コネクターは、2 番目のレコードを **outbox.event.orders** トピックに出力します。 別の送信トレイテーブル列からこの値を取得するには、コネクター設定で **route.by.field** SMT オプションを設定します。
aggregateid	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。 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、コネクター設定で**table.field.event.key** SMT オプションを設定します。
payload	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プトが含まれ、各フィールドは個別に表されます。詳細は、Emitting messages with additional fieldsを参照してください。 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、コネクター設定で **table.field.event.payload** SMT オプションを設定します。
追加のカスタム列	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。 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** などがあります。

12.6.3. 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 コネクターを設定するには、**outbox.EventRouter** SMT を設定します。例えば、**.properties**ファイルの基本的な設定は次のようになります。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
```

12.6.4. 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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを 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ーを設定できます。

- 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。

12.6.5. 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** カラムの値は、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ーに適用します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.transforms.outbox.EventRouter
value.converter=io.debezium.converters.ByteBufferConverter
```

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。**ByteBufferConverter** 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 コネクターを設定することができます (サポートはコネクターによって異なります)。これらのイベントは **ByteBufferConverter** でシリアライズすることができないため、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するように、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.ByteBufferConverter
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。

12.6.6. 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。**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=type:header:eventType
```

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=type:envelope:eventType
```

12.6.7. 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 メッセージは以下のようになります。

```
# 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、以下のように **table.expand.json.payload** を true に設定し、**StringConverter** 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.storage.StringConverter
```

12.6.8. 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	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。 ● **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、コネクターは処理を停止します。 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。
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。
table.field.event.timestamp		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。
table.field.event.payload	**payload**	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。

オプション	デフォルト	グループ	説明
table.field.event.payload.id	**aggregateid**	テーブル	ペイロード 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キーとして使用されます。 このオプションは非推奨になっています。代わりに table.field.event.key を使用してください。
table.expand.json.payload	**false**	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。 詳細については、expanding escaped json セクションを参照してください。
table.fields.additional.placement		テーブル、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。 **id:header,my-field:envelope** 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。 **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。
table.field.event.schema.version		テーブル、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	**aggregatetype**	ルーター	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テーブルの説明 に示します。
route.topic.regex	**(?<routedByValue>.*)**	ルーター	送信トレイ SMT が RegexRouter 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。 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 **${routedByValue}** 変数を **route.by.field** 送信トレイ SMT オプションの設定に置き換えることです。
route.topic.replacement	**outbox.event.${routedByValue}**	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、**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 **顧客** の場合には、トピック名は **outbox.event.customers** 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。 ● **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	**false**	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。

12.7. 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トレイイベントルーター SMT を使用する方法については、Outbox event router 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に設定します。

- 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。

コネクター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ます。

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」

12.7.1. 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 コネクターは、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\": {\"$oid\": \"b2730779e1f596e275826f08\"}, \"aggregatetype\": \"Order\", \"type\": \"OrderCreated\", \"payload\": {\"_id\": {\"$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}}",
  "source": {
    "version": "1.7.2.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。

12.7.2. 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",
  "aggregatetype": "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	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。 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、コネクター設定で **collection.field.event.id** SMT オプションを設定します。
aggregatetype	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 **${routedByValue}** 変数を置き換えます。 たとえば、デフォルト設定では、**route.by.field** SMT オプションは **aggregatetype** に設定され、**route.topic.replacement** SMT オプションは **outbox.event.${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。コネクターは、最初のドキュメントを **outbox.event.customers** トピックに送信します。コネクターは、2 番目のドキュメントを **outbox.event.orders** トピックに送信します。 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、コネクター設定で **route.by.field** SMT オプションを設定します。
aggregateid	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。 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、コネクター設定で **collection.field.event.key** SMT オプションを設定します。

フィールド	結果
payload	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プトが含まれ、各フィールドは個別に表されます。詳細は、Emitting messages with additional fieldsを参照してください。 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、コネクター設定で **collection.field.event.payload** SMT オプションを設定します。
追加のカスタムフィールド	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。 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。

12.7.3. 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 コネクターを設定するには、**outbox.EventRouter** SMT を設定します。次の例は、**.properties** ファイル内の基本的な SMT の設定を示しています。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
```

12.7.4. 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 コネクターが出力する変更イベントメッセージの他に、コネクター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ーを 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ーを設定できます。

- 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 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ーに適用します。

```
transforms=outbox,...
transforms.outbox.type=io.debezium.connector.mongodb.transforms.outbox.MongoEventRouter
value.converter=io.debezium.converters.ByteBufferConverter
```

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。**ByteBufferConverter** を値のコンバーターとして設定すると、**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 コネクターを設定することができます (サポートはコネクターによって異なります)。これらのイベントは **ByteBufferConverter** でシリアライズすることができないため、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するように、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.ByteBufferConverter
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。

12.7.6. 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。**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.connector.mongodb.transforms.outbox.MongoEventRouter
transforms.outbox.collection.fields.additional.placement=type:header:eventType
```

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.connector.mongodb.transforms.outbox.MongoEventRouter
transforms.outbox.collection.fields.additional.placement=type:envelope:eventType
```

12.7.7. 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
```

12.7.8. 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12.9 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

オプション	デフォルト	グループ	説明
collection.op.invalid.behavior	warn	コレクション	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。 ● **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、コネクターは処理を停止します。 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)。
collection.field.event.id	_id	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 **id** キーに保存されます。
collection.field.event.key	aggregateid	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。
collection.field.event.timestamp		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
collection.field.event.payload	**payload**	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。
collection.expand.json.payload	**false**	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。 詳細については、expanding escaped json セクションを参照してください。
collection.fields.additional.placement		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。
collection.field.event.schema.version		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	**aggregatetype**	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	**(?<routedByValue>.*)**	ルーター	送信トレイ SMT が RegexRouter 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 **${routedByValue}** 変数を **route.by.field** 送信トレイ SMT オプションの設定に置き換えることです。
route.topic.replacement	**outbox.event.${routedByValue}**	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、**outbox.event.** の後に送信トレイコレクションドキュメントの **aggregatetype** フィールドの値が続きます。たとえば、**aggregatetype** の値が **顧客** の場合には、トピック名は **outbox.event.customers** 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。 ● **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	**false**	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。

改訂日時：2022-10-22 22:54:10 +1000