PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#3

2016年度WG3活動報告書
異種DB間連携 連携方式調査編

目次

1. ライセンス

2. はじめに

2.1. PostgreSQLエンタープライズコンソーシアムとWG3について

PostgreSQLエンタープライズコンソーシアムは、PostgreSQL本体および各種ツールの情報収集と提供、整備などの活動を通じて、ミッションクリティカル性の高いエンタープライズ領域へのPostgreSQLの普及を推進することを目的として設立された団体です。

PostgreSQLエンタープライズコンソーシアム 技術部会ではPostgreSQLの普及に対する課題を活動テーマとし3つのワーキンググループで具体的な活動を行っています。

- WG1（性能ワーキンググループ）
- WG2（移行ワーキンググループ）
- WG3（課題検討ワーキンググループ）

WG3ではPostgreSQLの設計運用で想定される様々な課題に対し調査検証を行い、PostgreSQLが広く活用される事を推進しています。

2.2. 本資料の概要と目的

本資料はWG3の2016年度の活動としてPostgreSQLと他のRDBMS間でデータ連携を検討するにあたって様々な連携方式と周辺ツールを評価、整理するとともに一部のツールについて簡単な動作確認を行ったものです。

2.3. 本資料の構成

2. はじめに
3. 異種 DBMS と PostgreSQL の連携について
    - 異種 DBMS と PostgreSQL 間でデータ連携する目的や課題について記載しています。
4. DB間連携の各方式
    - データベース間のデータ連携方式について整理し、代表的なものを評価した比較表を掲載しています。
5. ETLツールを用いた連携方式
    - ETLツール Talendを用いた連携方式について概要、簡単な動作手順を記載しています。
6. SymmetricDS
    - DB同期ツール SymmetricDSを用いた連携方式について概要、簡単な動作手順や動作検証の結果を記載しています。
7. Logical Decoding
    - PostgreSQLのロジカルデコーディングを用いた連携方式について概要、簡単な動作手順を記載しています。
8. 外部データラッパ(FDW)を用いた連携方式
    - PostgreSQLのFDWを用いた連携方式について概要、簡単な動作手順を記載しています。

2.4. 想定読者

本資料の読者は以下のような知識を有していることを想定しています。

- DBMSを操作してデータベースの構築、保守、運用を行うDBAの知識
- PostgreSQLを利用する上での基礎的な知識

3. 異種 DBMSと PostgreSQLの連携について

3.1. 異種 DBMS と PostgreSQL を連携する目的

ICTシステムは往々にして当初のシステム設計と異なる要件が発生し、システムの拡張やシステム間連携などが 新たに必要となる場合があります。例えばビジネス状況の変化や事業の拡大に伴う新サービスの設立であったり、事業部門間または企業間の連携/統合、部門や地域をまたがった横断的なデータ分析への要求などが ユースケースとして挙げられます。

しかし、ICTシステム全体を統合/拡張するにはアプリケーションの改修や、データベースをはじめとする各システムの統合、運用面の検討が必要で多額のコストも必要となり、なかなか現実的ではありません。このため、既存システムを活用したままサブシステムを新たに拡張するケースが多くあります。

新たに拡張するシステムのデータベースとしてコストを抑えるためにPostgreSQLの利用シーンが増えており、異種DBMSとの連携を行うニーズが高まっています。

異種DBMSとPostgreSQLを連携させる目的や機会は様々ですが、例を挙げると以下のようなケースがあります。

1. マスタデータを活用するサブシステムへのデータ連携
    - 基幹システムからサブシステムに最新情報のデータ配信を実施する。マスタデータを地方の支店や工場にデータを連携して、最新の状態を現地のシステムで利用するなど。
2. 負荷分散による連携
    - 参照負荷の高いシステム向けにPostgreSQLを複数台立て、マスタ管理のデータベースの負荷を軽減させる。
3. DR バックアップサイトの構築
    - 災害対策に備えて遠隔地にマスター DB の複製を確保する。
4. データベース移行による期間限定の連携
    - 事業統合などに伴うデータの統合や、開発サーバへのデータ移行や同期の為に一時利用する。

3.2. 異種 DBMS と PostgreSQL の連携を実現する際の課題

異種DBMSとPostgreSQL間でデータの連携を取る手段として様々な連携方式や各種ツール/ソフトウェアが存在していますが、各方式を選定するにあたっては以下のような多くの要素を考量し、選定する必要があります。

- データ変換の有無
- データ同期のリアルタイム性
- 実行性能(反映にかかる時間や、反映元、反映先のDBに与える性能影響)
- 連携する単位の粒度(DB単位、スキーマ単位、テーブル単位など)
- データ連携の仕組みに向けた設計開発の難易度
- かけられるコスト/開発期間

どの方式が最適であるか判断する為には、各連携方式の特徴を網羅的に理解し、比較判断するための観点を持っておくことが課題となります。

本書ではそのような課題を解決すべく、PostgreSQLでデータ連携を実現する代表的な方式に対して 検討すべき観点を列挙し、各観点への評価を記載しています。

4. DB間連携の各方式

4.1. 今回の調査にあたって

データベースで利用可能なデータ連携方式については、体系的に網羅していくと以下のような形で分類し、列挙していくことができます。(分類の仕方や挙げているものは一例となります)

- ハードウェアレイヤ/ストレージレイヤ
    - ストレージコピー機能
    - 物理的なディレクトリのバックアップ＆リストア(PGDATAディレクトリなど)
    - データベースのdump出力機能(pg_dumpなど)
    - テキストファイル出力(csvなど)
- データベースレイヤ
    - ログシッピングレプリケーション(ストリーミングレプリケーションなど)
    - ロジカルデコーディング
    - 外部データラッパ
- ソフトウェアレイヤ
    - DB同期ツール:ログ解析またはトリガベースのレプリケーション(SymmetricDSなど)
    - ETLツール(Talend、PowerCenterなど)
    - 仮想データ統合(JBoss Data Virtualizationなど)

本書では以下の4つに絞り、調査対象としています。

- ETLツール(Talendをピックアップ)
- DB同期ツール(SymmetricDSをピックアップ)
- ロジカルデコーディング
- 外部データラッパ(FDW)

上記4つの方式を対象に、調査/評価の観点として以下の要素を設定し調査しました。また、各方式の実施例についても以降の章で記載しています。

調査の観点	評価するポイント
想定用途	どのようなユースケースで活用できることを想定されているか
異種DBとの接続性	連携可能なDB種類の豊富さと、汎用性の高さについて
データ同期:A⇒Bで同期をとることは可能か	A⇒Bで連携データの同期はとれるか。差分反映は可能か
データ同期:リアルタイム性	バッチやジョブレベルでの定期的な反映なのか。クエリが発行されたらすぐ反映されるか、準同期か、それともクエリ単位で同期されるか
データ転送:データ転送/NW部分について	データ転送性能やNWに関わる部分について。データ圧縮機能はあるか。情報の付加など、生データより多いデータ量の転送となるか
データ転送:データ連携処理コンポーネント本体の処理性能	データ抽出およびデータ格納処理自体の性能について。性能面での劣化、追加コストなどはあるか
データ転送:連携処理時の、ソース側DBへの性能影響について	A⇒B(異種DB⇒PostgreSQL)連携時、ソース側であるAの処理に影響が発生するか
データ転送:連携処理時の、反映側DBへの性能影響について	A⇒B(異種DB⇒PostgreSQL)連携時、反映側であるBの処理に影響が発生するか
データ連携単位の粒度(インスタンス・DB・スキーマ・テーブル・列・行)	どの単位で抽出、またどの単位で反映ができるか
セキュリティ	データベースの特権の分離が可能か。ユーザ管理が同一の基盤上で動くか、それとも独立管理が必要か。転送時の暗号化が可能か
可用性:障害時の継続性	ソース側Aが落ちた場合、連携側Bで処理が継続できるか
可用性:フェイルバック、リカバリ	ソース側Aが落ちて復旧した後、フェイルバックやリカバリは可能か
データ連携の仕組みに向けた設計開発の難易度	データ連携機構を構築するにあたっての難易度(全体的なSIの難易度)
アプリケーション開発者に向けた難易度	連携用のAPを開発する際の難易度(テーブルの透過性、異種DB間でのストアド・関数開発など)
データ変換(文字コードからコード体系統一、MDM的な観点など)	文字コードが違う場合など自動で変換、対応してくれるか。その他の変換もカスタマイズができるか

上記観点を元に一覧比較表を作成し、以下の分類にて評価しました。

評価については調査担当者による絶対評価として記載しています。一例として参考にして下さい。

記号	意味
◎	非常に得意、売りの機能。
○	可能、問題なく使える。
△	使えるが制約がある、問題になるケースがある。
×	対応していない。事実上使えない。
ー	評価観点として適切ではない

4.2. 一覧比較表

比較表を別途Excelのpdf形式で添付いたします。一覧比較表（pdf形式）

5. ETLツールを用いた連携方式

5.1. ETLツールとTalendについて

5.1.1. ETLツールの活用

ETLとは異なるデータベース間でデータを転送する際の3つの工程、Extract(抽出)、Transform(変換・加工)、Load(ロード)の一連の処理を指す用語です。

ETL処理の作成/実施を支援する為に、ITベンダ各社より様々なツールが展開されています。

異種DBとPostgreSQL間でデータ連携を実施するにあたって ETLツールが必要とされるケースは以下のような場合があります。

1. 連携元DBから抽出したデータが、そのままPostgreSQLに取り込めない形式の場合
2. データの抽出にあたりAPの作成が必要となる場合
3. データを反映する前にデータの加工/変換など中間処理が必要となる場合

1.のケースとしては、通常のRDBMSであればCSVに出力すればそのまま取り込める場合が多いです。しかし、Excelファイルや連携元DB独自のデータ形式など PostgreSQLが取り込める形式ではない場合、PostgreSQLに合せてデータの加工が必要となります。

2.のケースでは特定のスキーマ/特定の範囲でデータを抽出したい場合などAPの作成が必要となり、連携元のデータベースに対するプログラミングの知識も必要となります。

3.のケースではPostgreSQLへの反映にあたって文字コードやデータ型の変換、データのフィルタリング/ソートなどを 実施したい場合には何らかの変換ツールの活用であったり、APの作成が必要となります。

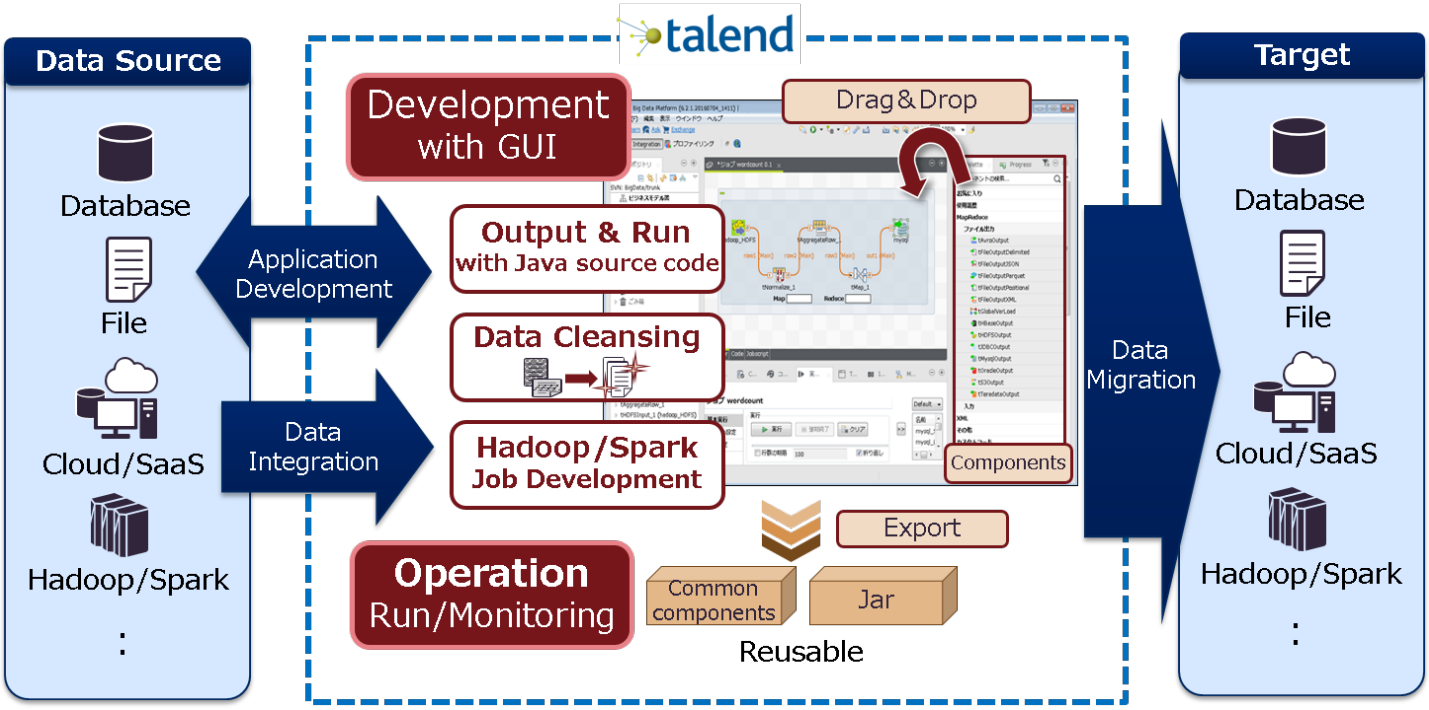
ETLツールはこれらの課題に対し大きな効果を期待することができます。GUIと予め用意された部品(コンポーネント)の組み合わせでプログラミングなしにAPを作成できるため、連携先のデータベースに対するプログラミングの知識がなくとも連携用のAPを構築できます。また、連携における中間処理も部品とし豊富に用意されており、複雑な業務も容易かつ柔軟に組むことができます。

5.1.2. ETLツール Talendについて

ETLツールは様々なベンダにより数多く展開されていますが、本章では無償版が提供されているTalendをピックアップし、紹介いたします。

Talendとは、Talend社が提供するデータ統合・連携製品です。

機能追加された有償版と無償のOSS版（Talend Open Studio）があり、それぞれTalend社やベンダー各社によるサポートが提供されています。



DBとDBを連携するAPの開発・運用・維持管理をサポートするデータ統合連携基盤であり、GUIを利用したジョブの作成と、JARファイルによる実行ファイルの出力まで実施できます。

Talendの特徴としてOSS版でも多くのETL機能、データファイルやDBへの接続コンポーネントを利用することができます。

OSS版と有償版の違いは以下のような点が挙げられますが、メインとなるETL開発に使えるコンポーネント(DBへの接続、データの抽出、加工、反映)については 無償版でもほぼ差はありません。

項目	OSS版	有償版
サポートの有無	ベンダによる提供有り	Talend社と、ベンダによる提供有り
作成したAPの管理と実行	手動で管理し、手動で配置/実行する	Talendによる構成管理、デプロイ、スケジュール実行が可能
開発形態	スタンドアロン環境で単独開発	クライアント・サーバ環境によるチーム開発が可能

5.2. Talendを利用した異種DB連携環境の構築手順

実際にTalend Open Studioを利用して異種DBと連携する手順を紹介いたします。

5.2.1. 実行環境と連携するデータのイメージ

本節では簡単な例として、Oracle Database 11gのデータを PostgreSQLに取り込むケースを解説いたします。

実行環境

項目	利用SW	OS	想定用途
反映元DBMS	Oracle Database 11g Express Edition	RedHat Enterprise Linux 7.2	携帯電話のコンテンツ決済情報を格納(マスタ)
反映先DBMS	PostgreSQL 9.6.2	RedHat Enterprise Linux 7.2	コンテンツ決済情報の分析用(参照用レプリカ)
ETLツール	Talend Open Studio for Data Integration V6.3.1	Windows7	連携用AP開発

テーブル定義(テーブル名 : mobile_payment)

項番	内容	列名	データ型(Oracle)	データ型(PostgreSQL)	連携時の型変換の有無
1	決済者のID(キャリアと電話番号)	user_id	char(16)	char(16)	ー
2	決済額	payment	number	int	○
3	利用サービスのID	service_id	char(8)	char(8)	ー
4	利用コンテンツのID	content_id	char(8)	char(8)	ー
5	決済時刻	payment_date	date	timestamp	○

主キーはuser_idとpayment_dateの複合キーとします。

リスト 5.1 投入データ例

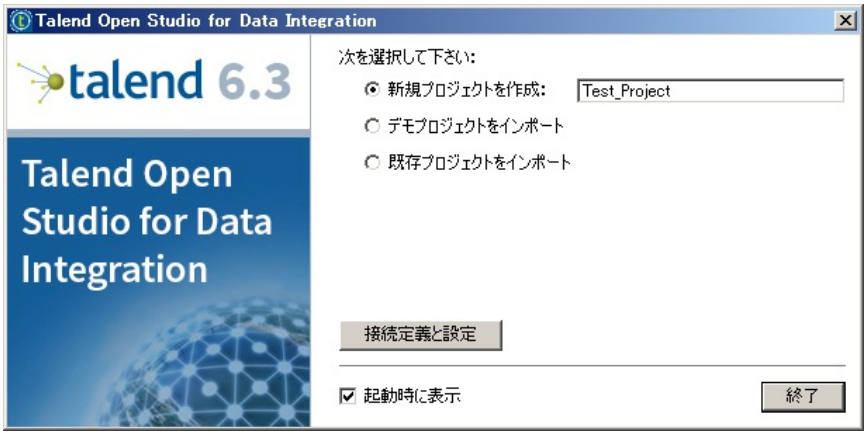
```
SQL> select * from mobile_payment;

USER_ID             PAYMENT SERVICE_ CONTENT_ PAYMENT_DATE
--------------- ---------- -------- -------- -------------------
0001-09012345678        500 00000021 00000036 2017/04/01 12:34:56
0001-08043218765       1000 00000015 00000021 2017/04/01 14:12:09
0002-09011223344        600 00000021 00000008 2017/04/01 14:18:05
```

5.2.2. Talend Open Studioのインストールと起動

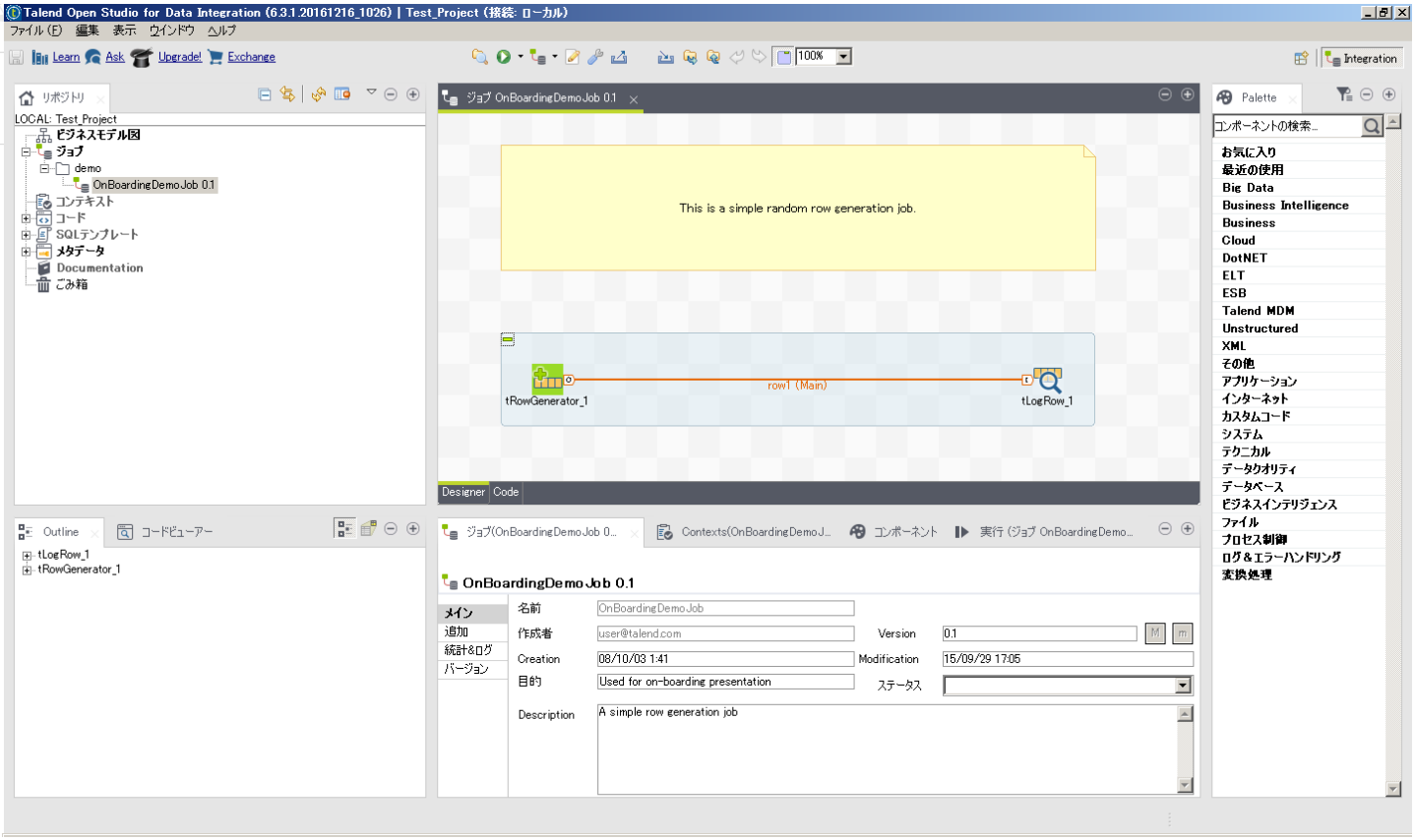
Talend社のHPからTalend Open Studio for Data Integrationをダウンロードし、手元の環境に展開、実行環境に合せて『TOS_DI-win32-x86.exe』または『TOS_DI-win-x86_64.exe』を実行します。

実行には事前にJavaのインストールが必要となります。Talend Open Studio for Data Integration V6.3.1の推奨Verは1.7です。その他、実行環境の要件はTalend社のHPから確認できます。

起動時にプロジェクトの選択を促されるため、ここでは新規にTest_Projectを作成します。

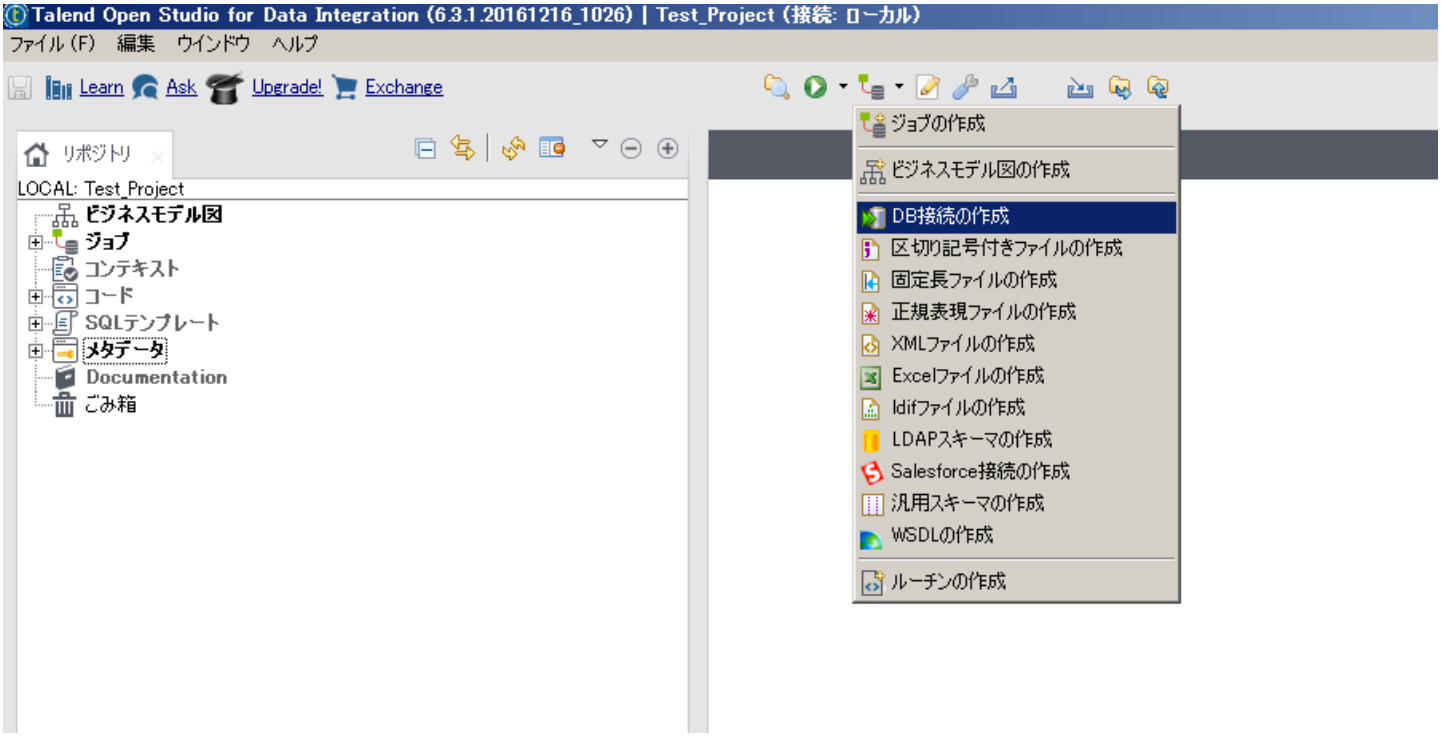


起動して最初の画面です。サンプルのジョブが開かれているので閉じておきます。



5.2.3. DBMSとの接続

ジョブ関連のアイコンから、『DB接続の作成』を選択します。

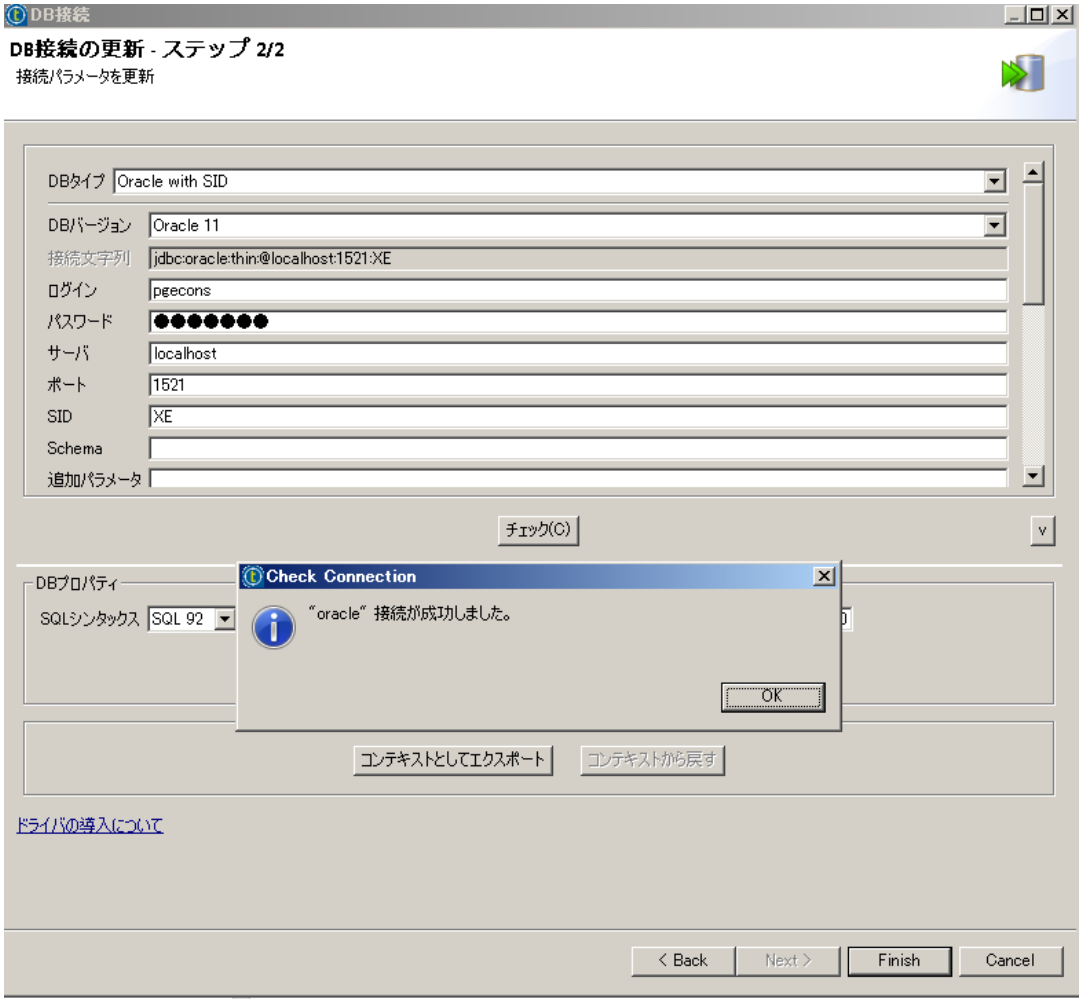


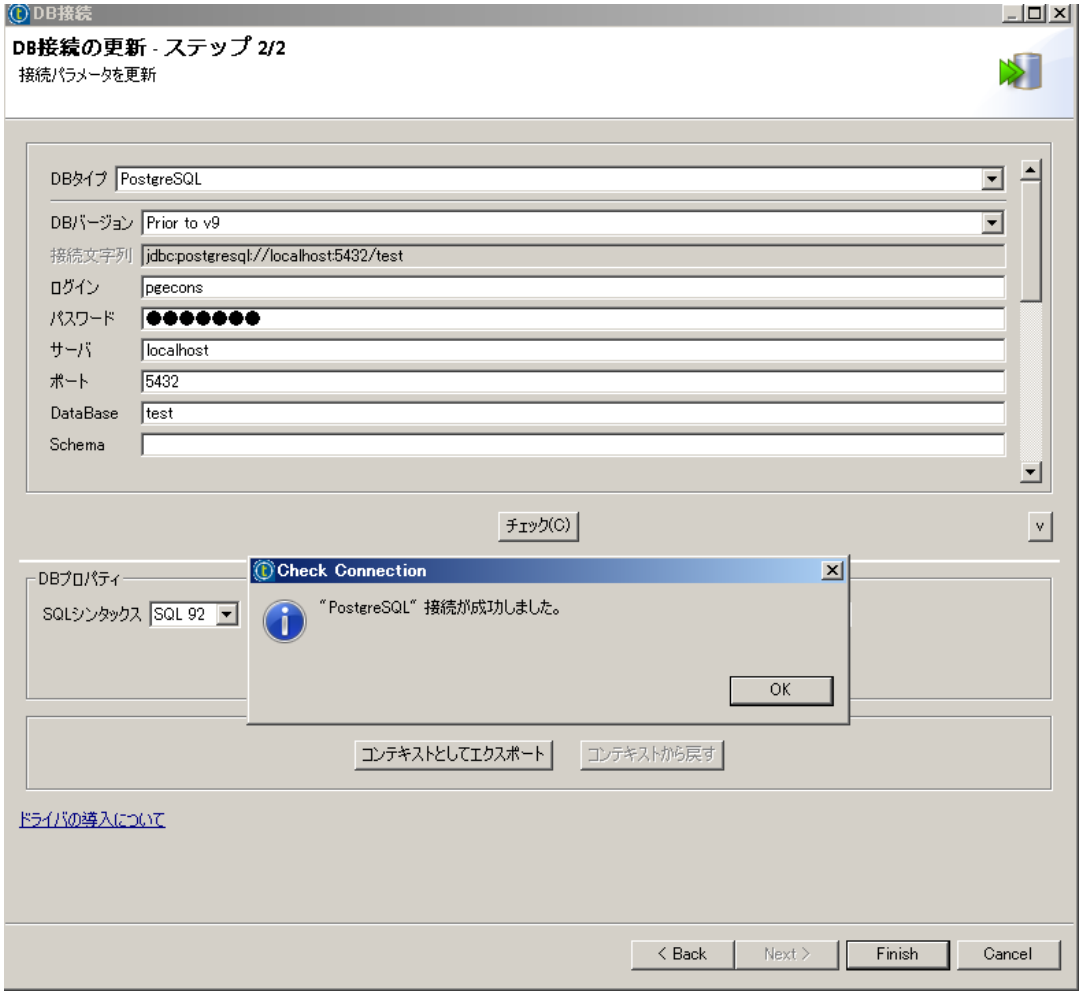
接続するDBMSに合せてDBタイプを選択し、必要な情報を入力します。チェックボタンで接続の確認が可能です。

この時、必要なJDBCドライバをTalendが自動でダウンロードします。

実行環境が外部ネットワークに接続されていない場合は、別途予め必要なJDBCを用意し、手動で配置する必要があります。

ログイン欄に指定するユーザー名/ロールには予め適切な権限を設定しておいて下さい。反映元DBへ接続するユーザーには参照権限、反映先DBへ接続するユーザーには各種更新権限が必要となります。



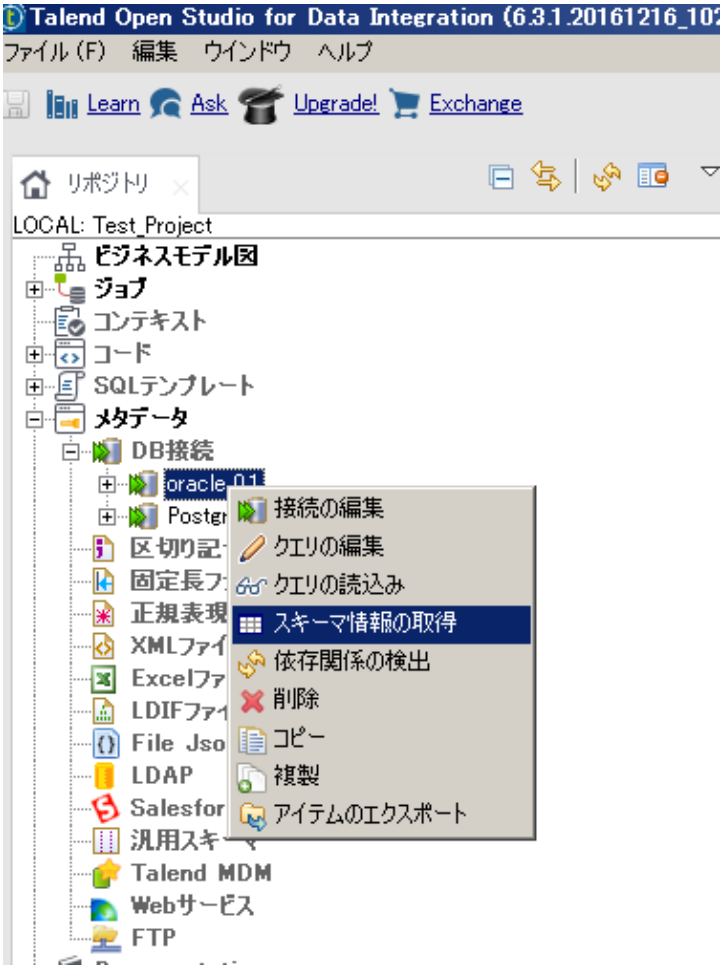


5.2.4. スキーマ情報の取得

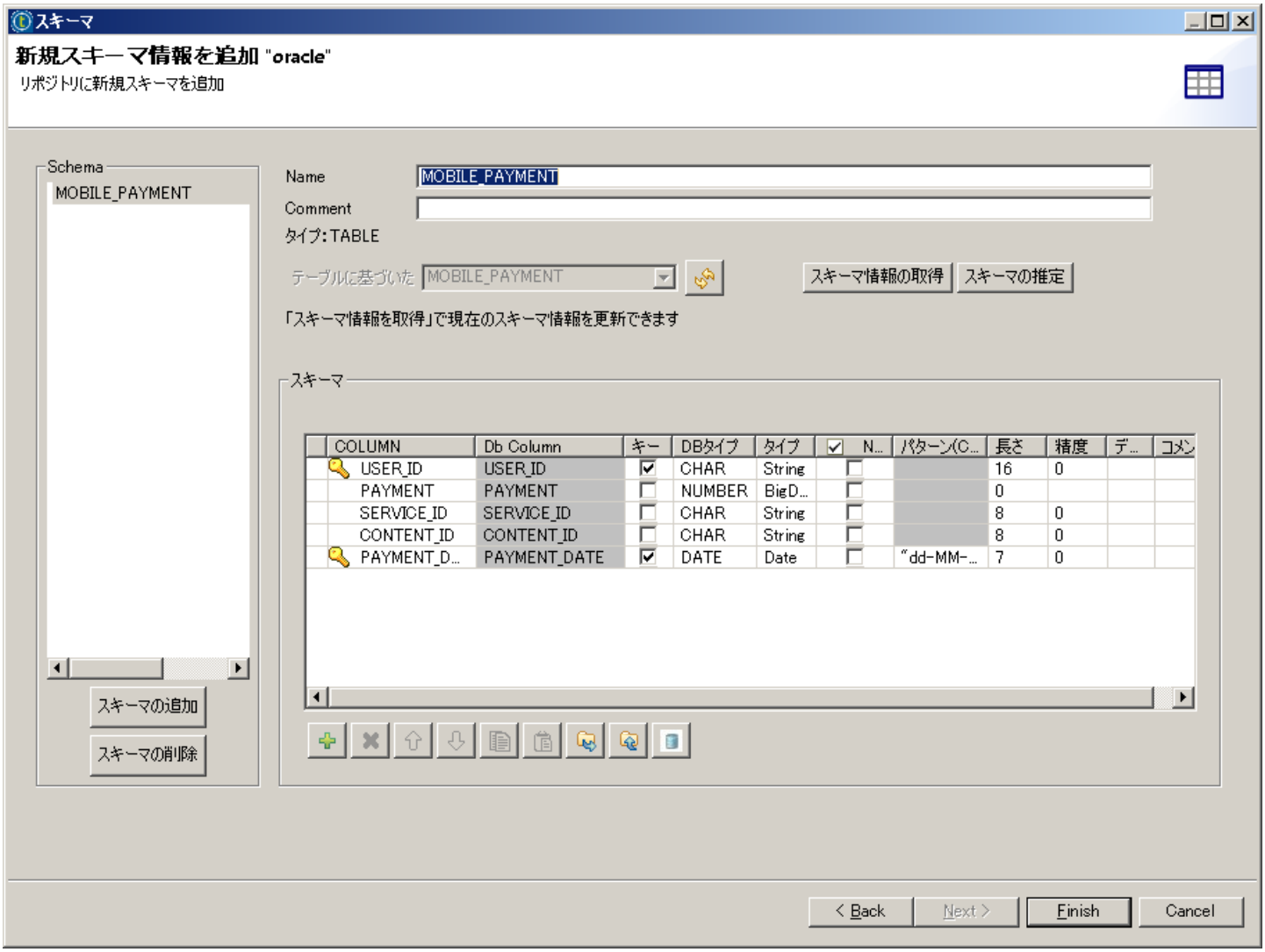
DBMSとの接続が設定できたら、データ連携の対象となるスキーマを選択し、読み込みます。

そうして取得したテーブルやビューがTalendの操作対象となります。

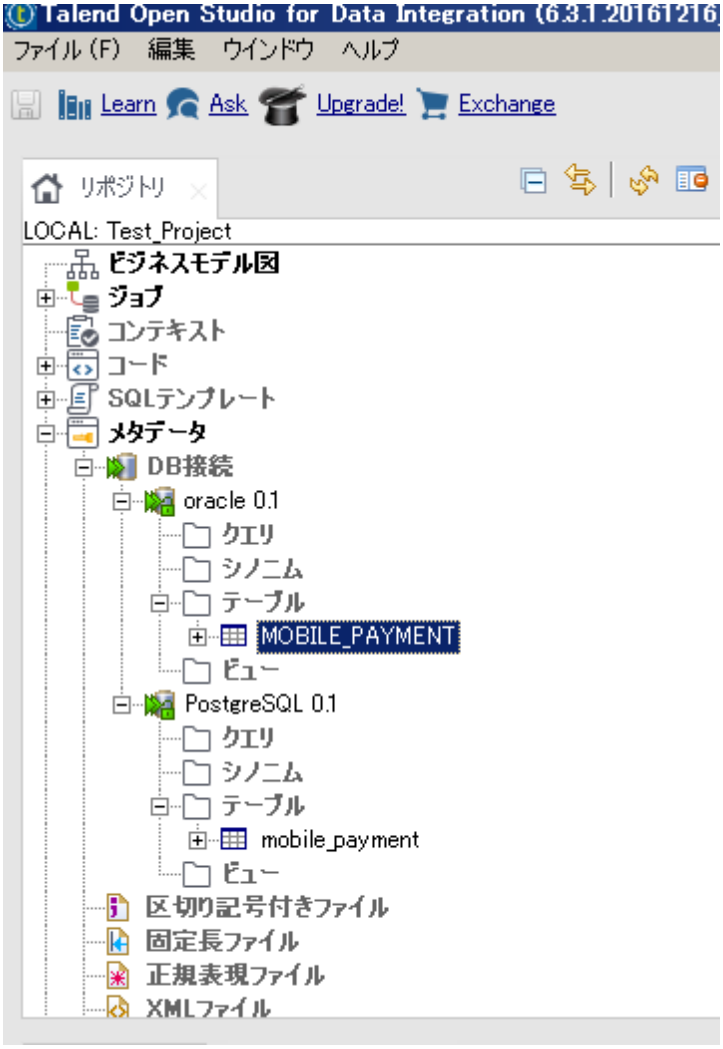
作成したDBの接続を右クリックし、『スキーマ情報の取得』を選択します。



データ連携の対象とするテーブルを選択します。



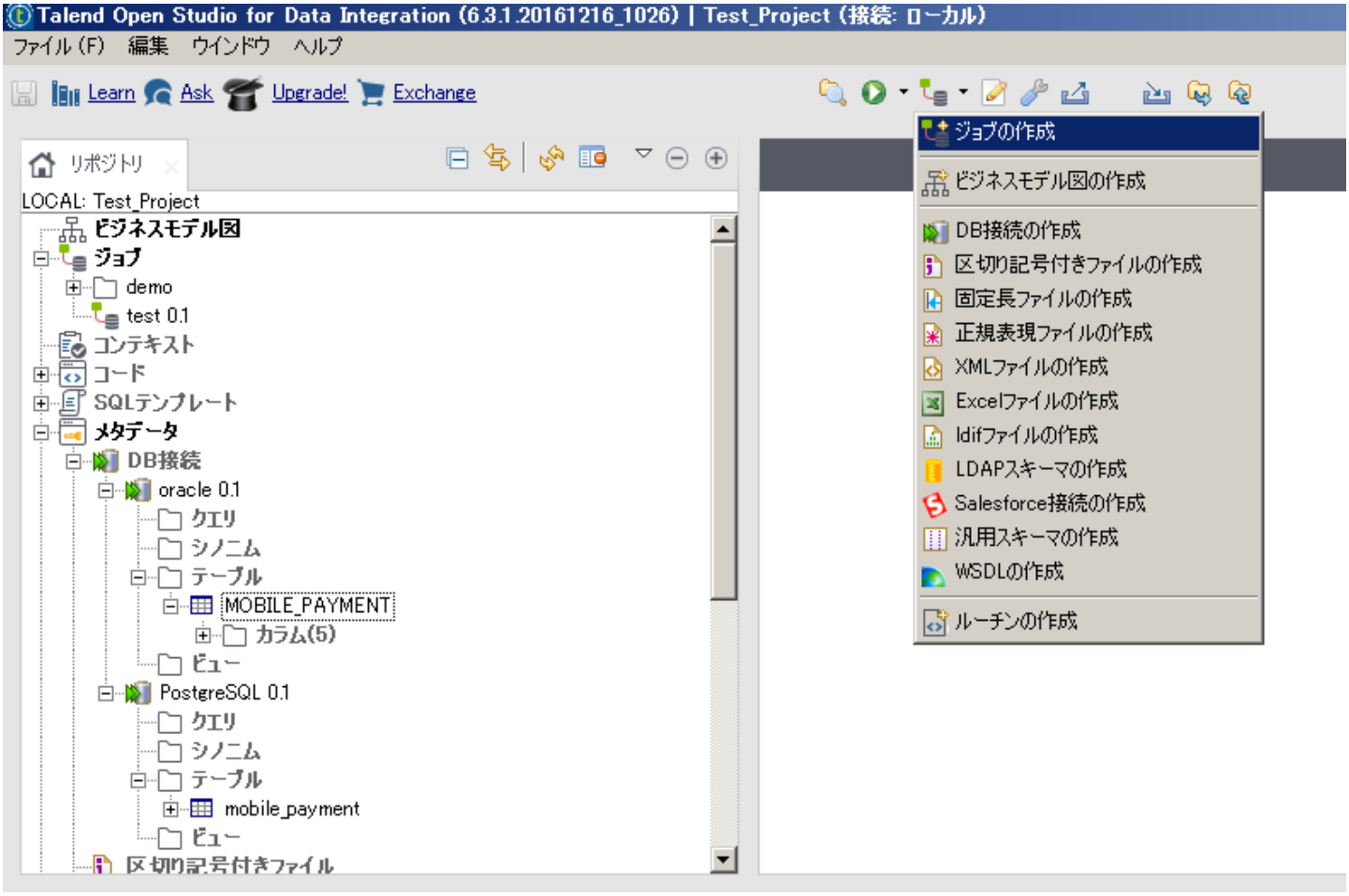
読み込んだスキーマは各DB接続のプルダウンメニューに表示されるようになります。



5.2.5. ジョブの作成

連携するDBMSへの接続と、スキーマ情報の取得を実施したらGUI上でコンポーネントを配置しながら連携の処理内容を作成していきます。

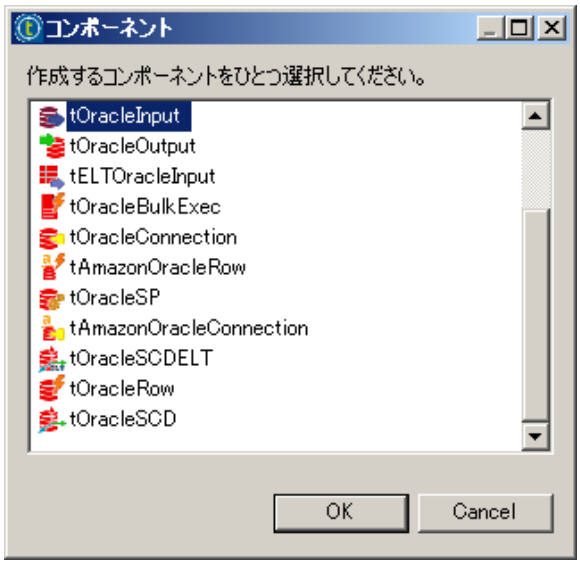
ジョブ関連のアイコンから、『ジョブの作成』を選択します。



中央のジョブ作成エリアに登録したスキーマ情報、テーブル『mobile_payment』をドラック＆ドロップすると コンポーネントとして配置することができます。

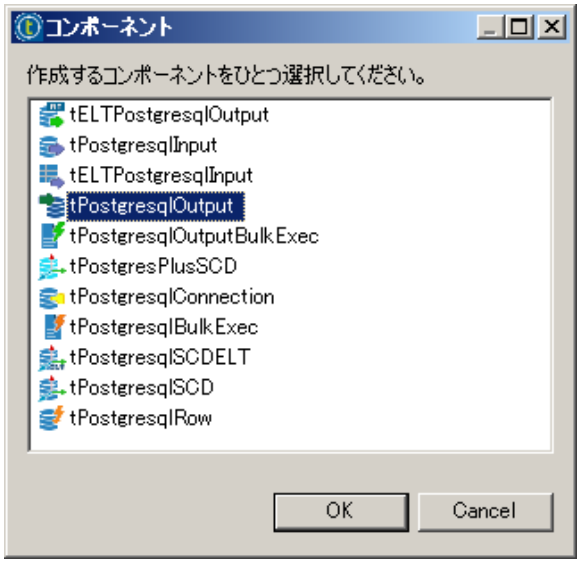
最初に反映元であるOracleのテーブルをコンポーネント配置します。

Oracleがデータの抽出元である場合は、『tOracleInput』を選択します。

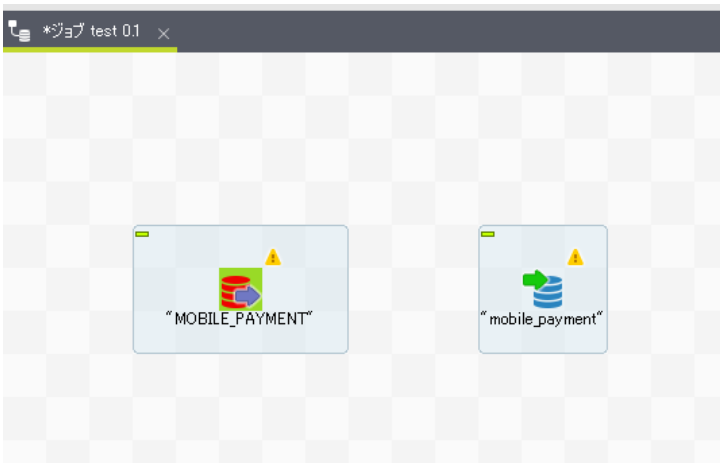


次に反映先であるPostgreSQLのテーブルも同様にドラッグ＆ドロップで配置します。

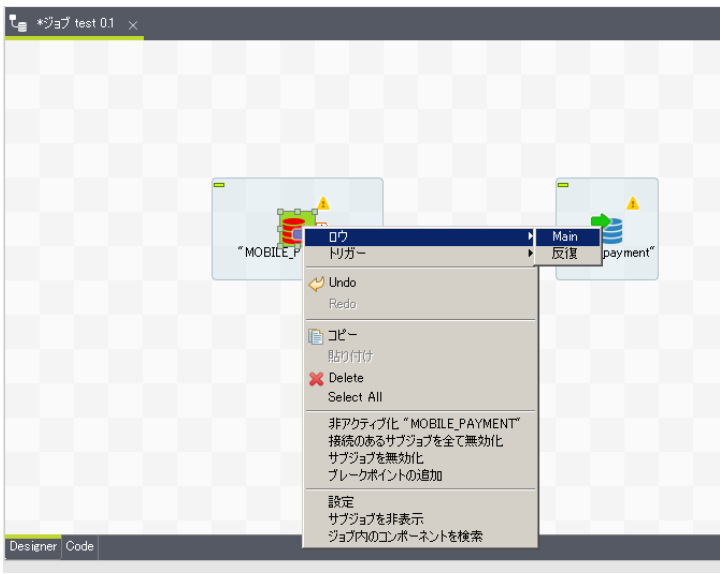
PostgreSQLが出力先である為、『tPostgresqlOutput』を選択します。



Oracleからの出力とPostgreSQLへの反映がそれぞれ配置されました。データの入/出力についてはコンポーネントのアイコンからも判別することができます。



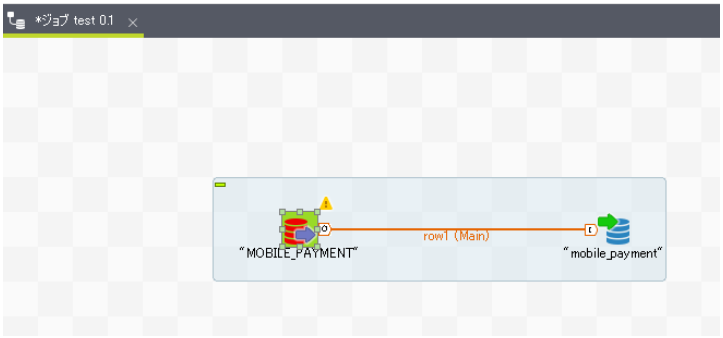
tOracleInputのアイコンを選択し、右クリック、『ロウ(Main)』を選択し、tPostgresqlOutputまでrowを繋げます。



Oracleからの抽出と、PostgreSQLへの反映が繋がりました。

これで単純な全件抽出、反映のジョブが完成となります。

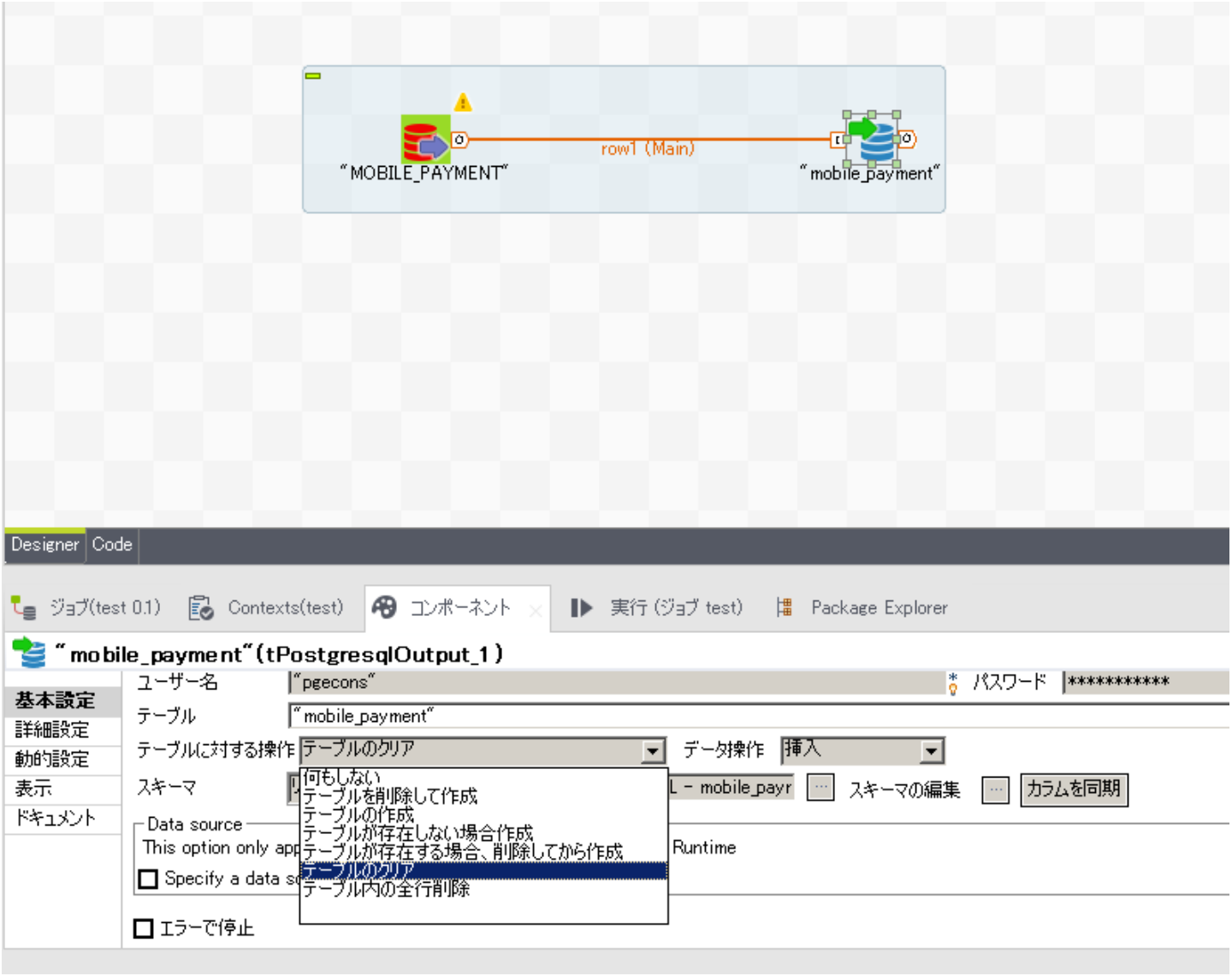
データ型について暗黙的に変換が可能なものは、Talendが自動で対応します。(NUMBER⇒IntやDATE⇒TimeStampなど)



OSS版であるTalend Open StudioではCDC(Change Data Capture)の機能が使えない為、差分反映が設定できません。

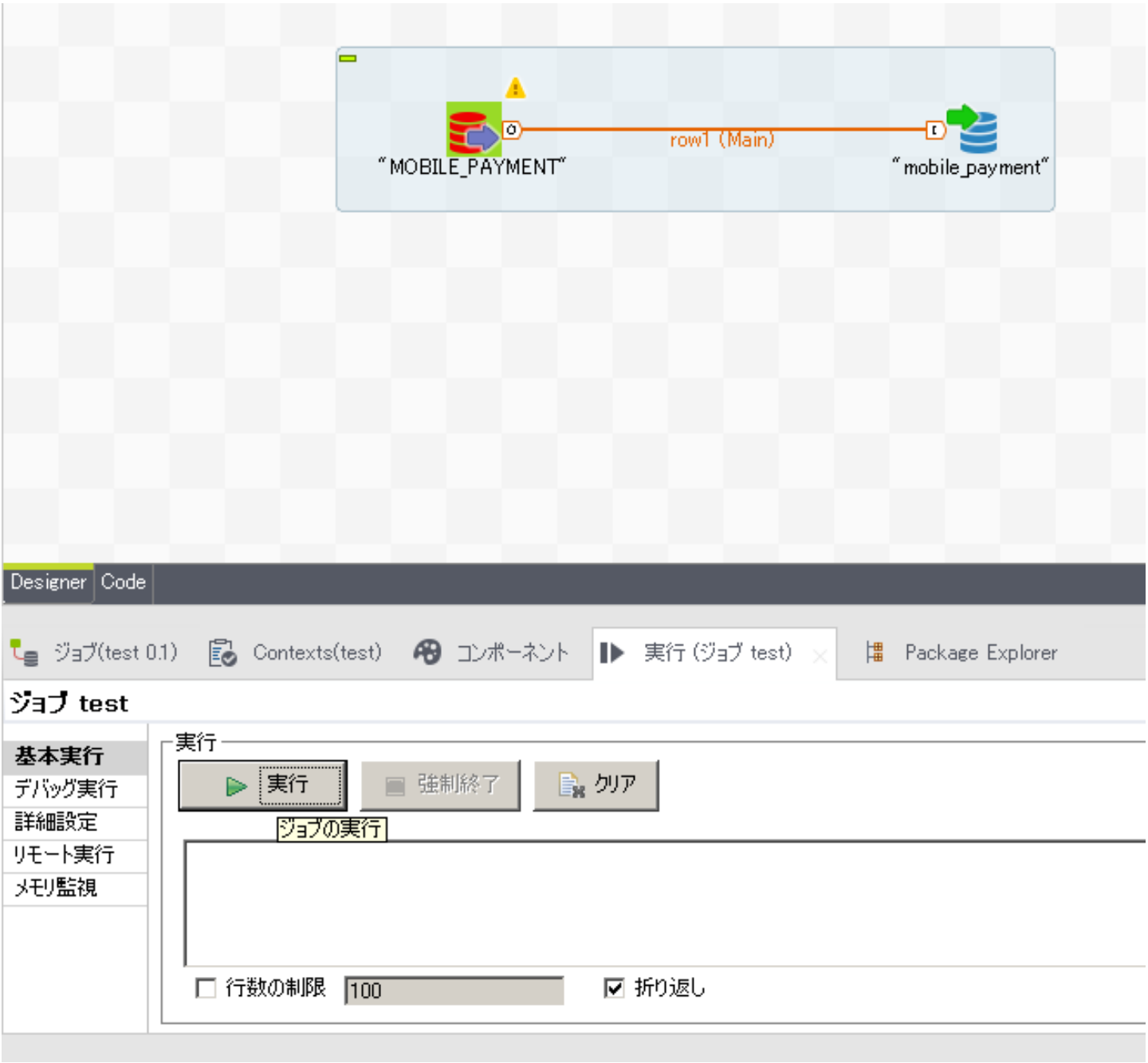
全件抽出、全件反映が基本となるため、今回の実施例ではPostgreSQLへの反映時、事前にテーブルをtruncateする処理を設定しておきます。

tPostgresqlOutputのアイコンを選択し、Talendの下部ウィンドウにある『コンポーネント』のタブを選択すると 選択したコンポーネントの詳細な設定が可能となります。

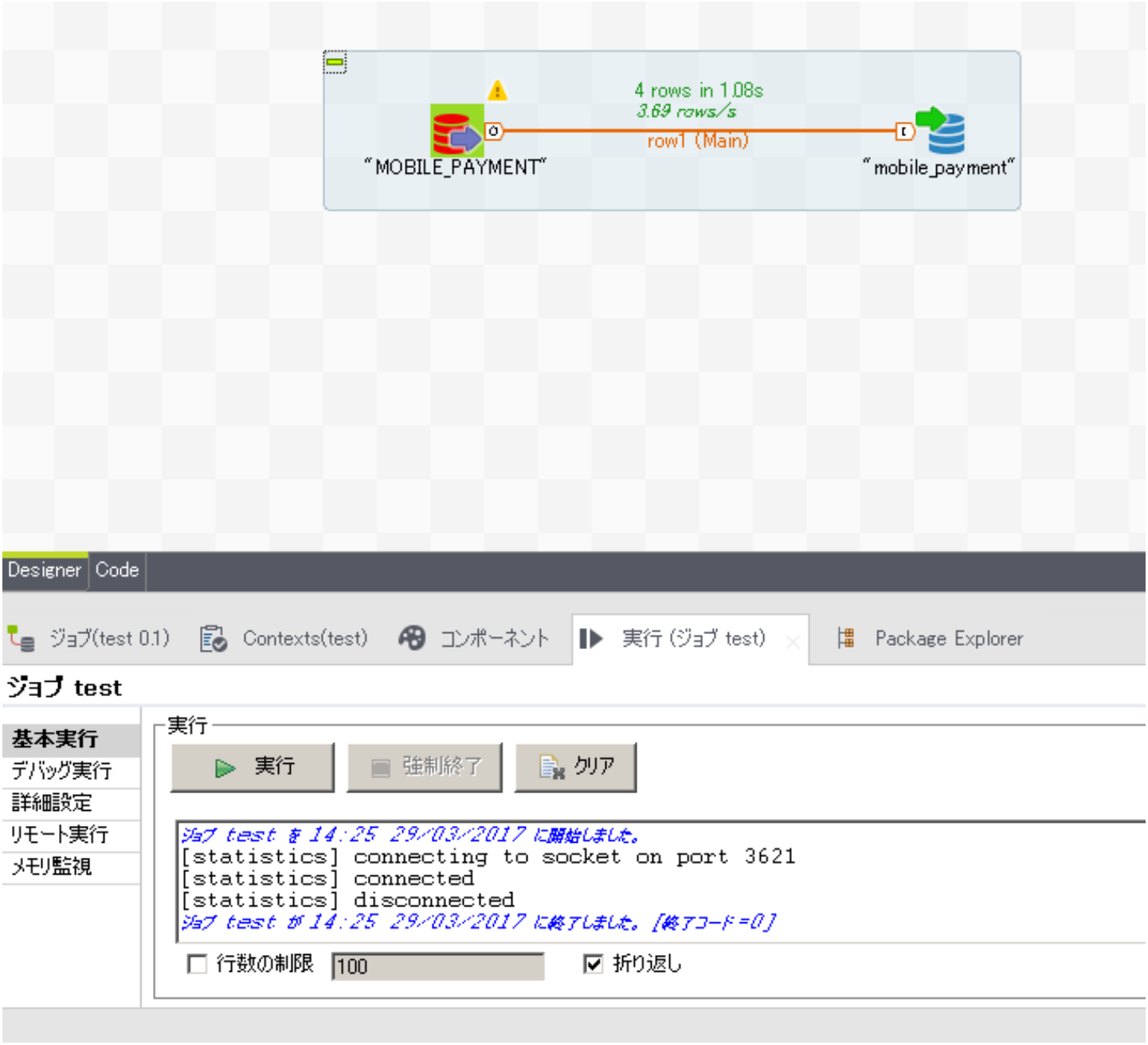


コンポーネント配置によるジョブの動作設定が完了したら、下部ウィンドウの『実行』タブを選択し、ジョブの実行をテストします

※実際に接続するDBMSへの操作が行われます。



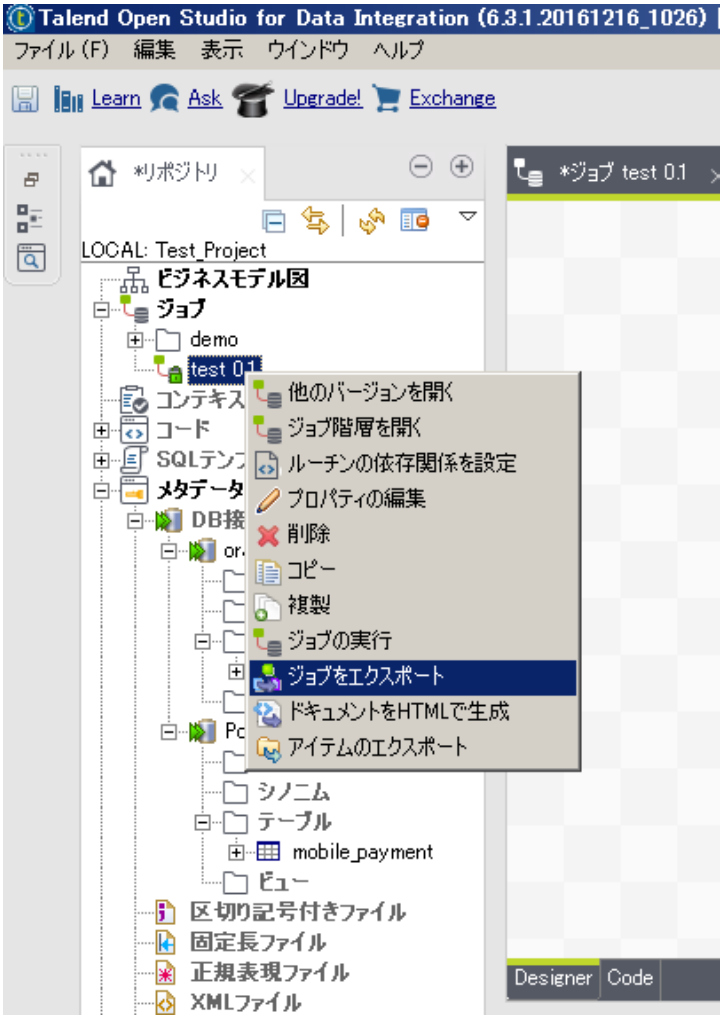
ジョブの実行が正常に終了できれば、ジョブの完成となります。ジョブ作成エリアには実行時間なども表示されます。



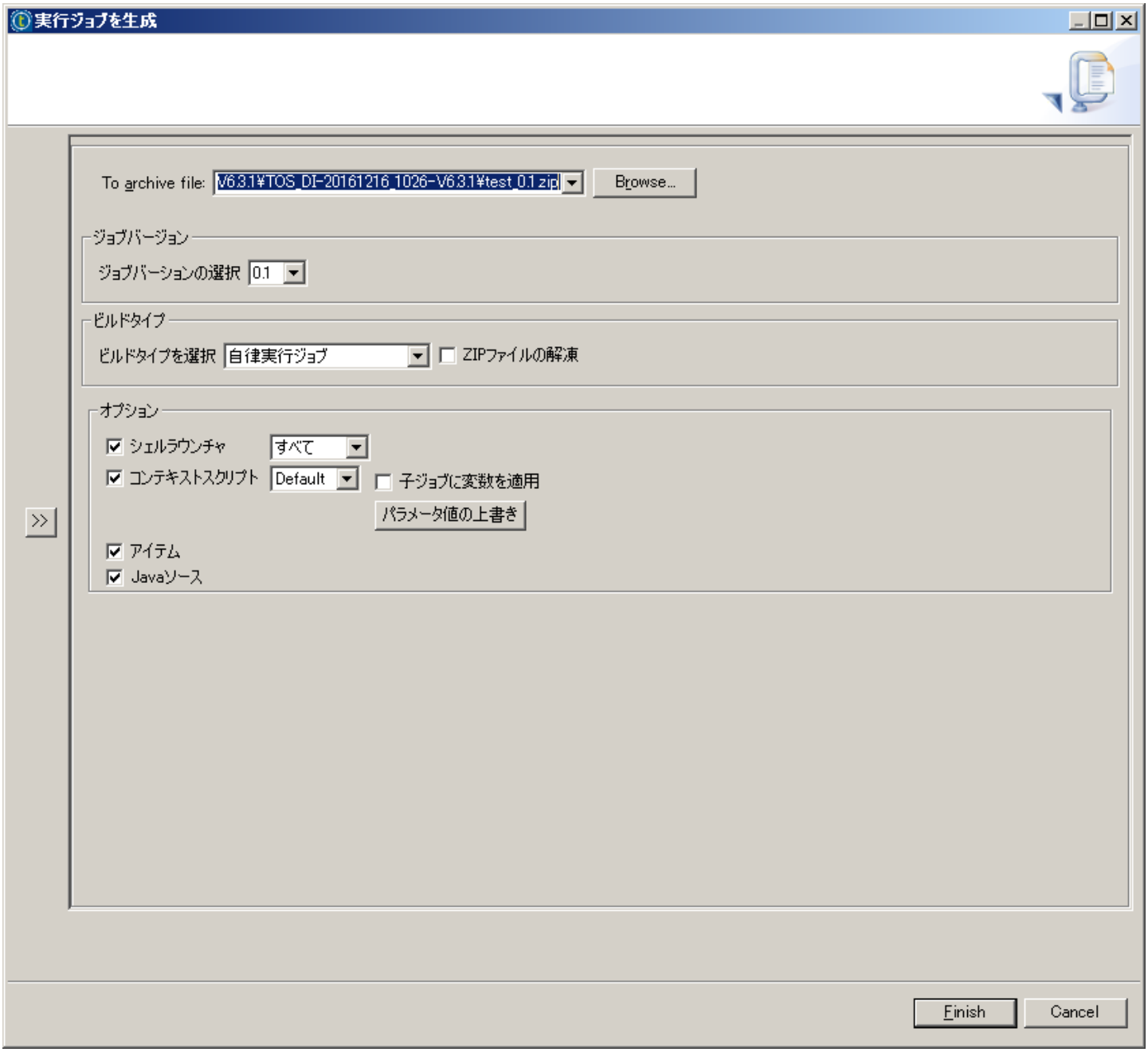
5.2.6. 実行ファイルの生成

作成したジョブはJavaのAPとして実行ファイルに出力することが可能です。

Talend左側のエクスプローラーから、出力するジョブを選択し、右クリックで『ジョブをエクスポート』を選択します。



ビルドタイプに『自律実行ジョブ』を選択し、出力先を指定して実行すると ジョブがビルドされ、jarファイルと関連ファイルをまとめたzipファイルが出力されます。



zipファイルには、作成したジョブの実行ファイル(jar形式)と、DBMSへの接続に必要なJDBC、jarを実行するbatファイルとスクリプトに加え、Javaのソースコードも含まれています。

5.2.7. 実行ファイルの配置と実行

出力したjarファイルとbatファイルまたは実行用スクリプトを任意のサーバに配置すれば、データ連携APを実施、OracleのデータをPostgreSQLに反映することが可能となります。

リスト 5.2 実行例

```
1.Oracle上のデータを確認

SQL> select * from mobile_payment;

USER_ID             PAYMENT SERVICE_ CONTENT_ PAYMENT_DATE
---------------- ---------- -------- -------- -------------------
0001-09012345678        500 00000021 00000036 2017/04/01 12:34:56
0001-08043218765       1000 00000015 00000021 2017/04/01 14:12:09
0002-09011223344        600 00000021 00000008 2017/04/01 14:18:05
```

```
2.PostgreSQLのサーバで作成したjarファイルを実行

[pgecons@host_postgres01 test]# ls
items  src  test_0_1.jar  test_project  test_run.bat  test_run.sh
[pgecons@host_postgres01]# ./test_run.sh
[pgecons@host_postgres01]#


3.PostgreSQLへの反映を確認

test=# select * from mobile_payment;
     user_id      | payment | service_id | content_id |    payment_date
------------------+---------+------------+------------+---------------------
 0001-09012345678 |     500 | 00000021   | 00000036   | 2017-04-01 12:34:56
 0001-08043218765 |    1000 | 00000015   | 00000021   | 2017-04-01 14:12:09
 0002-09011223344 |     600 | 00000021   | 00000008   | 2017-04-01 14:18:05
(3 行)


4.Oracle上でレコードを一件追加


SQL> insert into mobile_payment values ('0001-
09012345678',500,'00000021','00000038',to_date('2017-4-1 14:25:04','yyyy/mm/dd
hh24:mi:ss'));

1行が作成されました。

SQL> select * from mobile_payment;

USER_ID             PAYMENT SERVICE_ CONTENT_ PAYMENT_DATE
---------------- ---------- -------- -------- -------------------
0001-09012345678        500 00000021 00000036 2017/04/01 12:34:56
0001-08043218765       1000 00000015 00000021 2017/04/01 14:12:09
0002-09011223344        600 00000021 00000008 2017/04/01 14:18:05
0001-09012345678        500 00000021 00000038 2017/04/01 14:25:04

SQL> commit;

コミットが完了しました。

SQL>


5.PostgreSQLのサーバでjarファイルを再度実行

[pgecons@host_postgres01]# ./test_run.sh
[pgecons@host_postgres01]#


6.PostgreSQLへの反映を確認

test=# select * from mobile_payment;
     user_id      | payment | service_id | content_id |    payment_date
------------------+---------+------------+------------+---------------------
 0001-09012345678 |     500 | 00000021   | 00000036   | 2017-04-01 12:34:56
 0001-08043218765 |    1000 | 00000015   | 00000021   | 2017-04-01 14:12:09
 0002-09011223344 |     600 | 00000021   | 00000008   | 2017-04-01 14:18:05
 0001-09012345678 |     500 | 00000021   | 00000038   | 2017-04-01 14:25:04
(4 行)

test=#
```

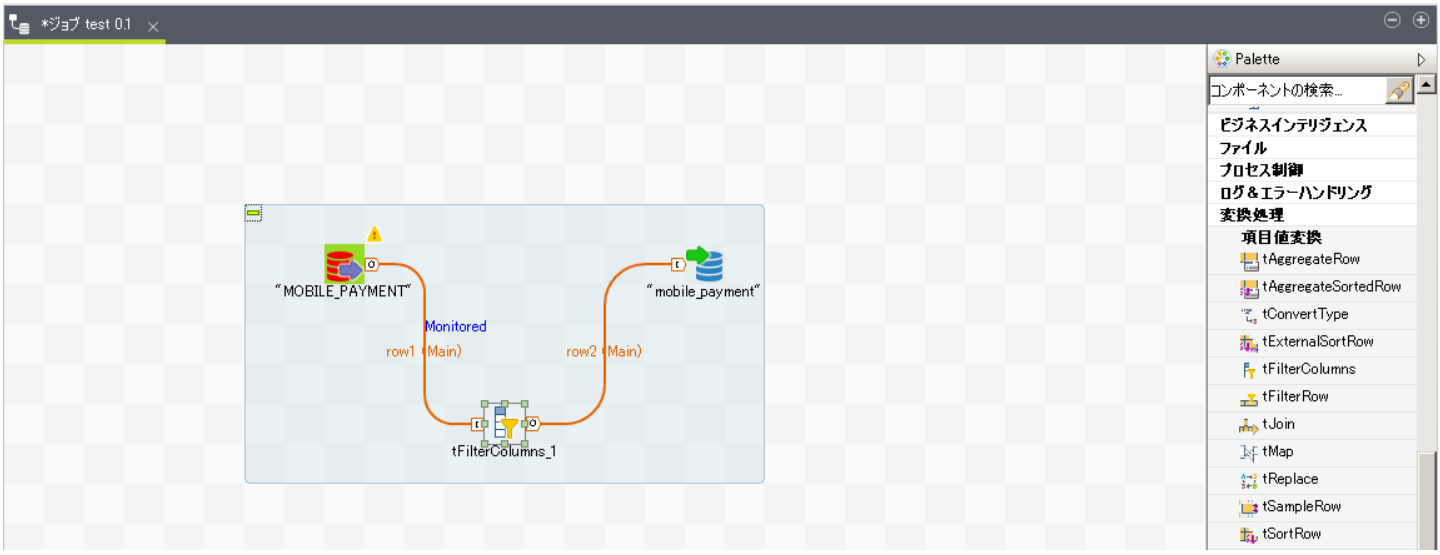
5.2.8. 補足事項

本節で紹介した構築手順では単純な全件コピーを実施するジョブを作成しましたが、Talend Open Studioには豊富な処理がコンポーネントとして部品で用意されてお

り、複雑な処理もGUIで直感的に作成することが可能です。

例えば、データの反映にあたりマスタデータから特定の列(個人情報など)を削除したい場合など フィルタのコンポーネントを間に挟み込むことで設定できます。

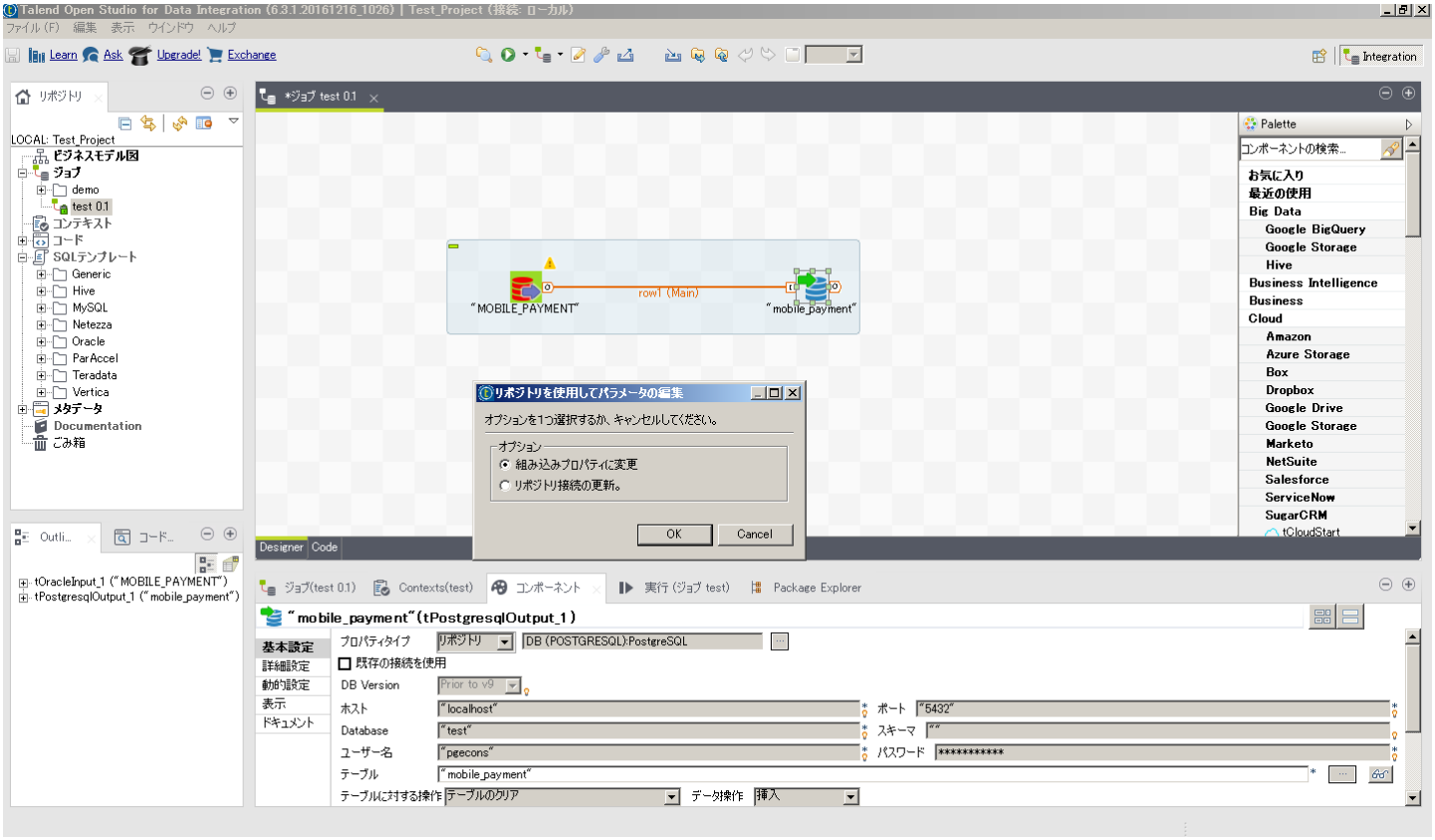
様々なコンポーネントは右側のパレットから選択できます。



また、Talendの実行環境と、出力する実行ファイルの配置先で ネットワークの設定などが異なる場合はDBへの接続情報を個別に変更することも可能です。

tOracleInputのアイコンを選択し、Talendの下部ウィンドウにある『コンポーネント』タブから 接続の際に指定しているホスト名などを変更可能です。

その際、オプションで組み込みプロパティの変更を選択すれば、本ジョブに関してのみ設定が変更されます。



5.3. ETLツール Talendの使い時と有償版の付加機能

Talend(OSS版)の使い時としては、

- 対象DBでのシステム開発の経験が少ない
- 開発期間が短い、開発工数が少ない
- APの継続的な改修があり、処理の可視化などのメンテナンス性が必要

などが想定されます。

Talendの利点は無償でも使えるETLツールであることと、数多くのDBに対応した コンポーネントが標準で用意されていることです。ビッグデータ/NoSQLに分類されるDB(mongoDBなど)やクラウド(AWSなど)にも対応しています。

留意点としては、OSS版ではCDCの機能が使えないため、差分反映を設定することができません。時系列によってテーブルを分ける、反映先は都度truncateするなど、運用による工夫が必要です。

有償版ではCDCの他にも様々な機能拡張がされているため、より高度な開発管理、運用を実施したい場合には有償版の採用も検討に値します。

有償版では

- 開発APのバージョン管理や共有リポジトリ
- 複数ジョブの並列実行
- Talendからの実行APのデプロイや、スケジュール実行
- ロードバランシングとフェールオーバー
- 連携処理エラー時のリカバリ機能

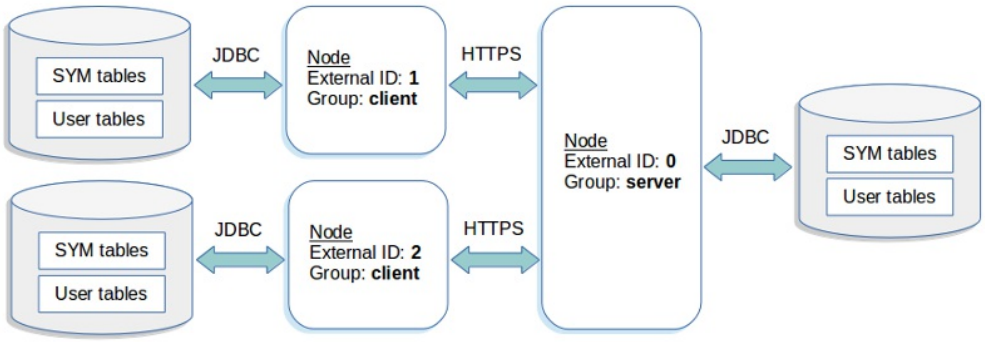
など、開発管理、運用管理に関わる高度な機能が利用可能です。

6. SymmetricDS

6.1. SymmetricDS について

SymmetricDSは、異種DB間で一方向レプリケーション、双方向マスターレプリケーションをサポートするツールで、レプリケーション時にフィルタリングや変換処理を行うことができます。レプリケーションは非同期でデータを複製します。

SymmetricDSは、Java上で動作するため、Javaをサポートする様々なデータベースに対応し、様々なプラットフォーム間で動作するように設計されています。



ライセンス体系は、OSSライセンスと商用ライセンス(SymmetricDS Pro)の2種類があります。OSSのライセンスは、GNU General Public License (GPL) version 3.0となっています。

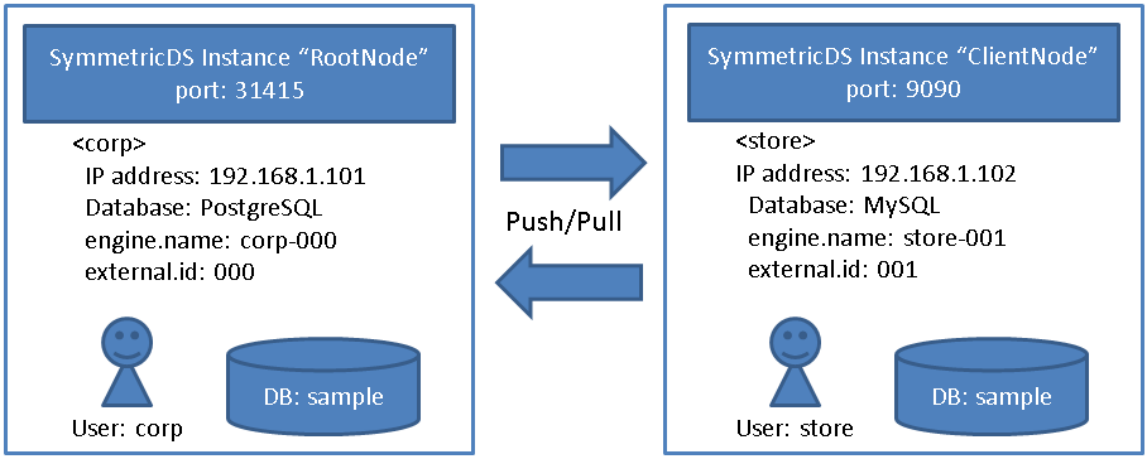
実際の挙動を確認するため、OSSライセンス版を使用して構築手順、挙動、性能影響について検証した結果をまとめます。検証したバージョンは、SymmetricDS 3.8.0 (OSSライセンス版) です。

なお、2012年度のPGEConsの成果物 の 異種DB間連携調査編(WG2) では、商用製品の紹介していますので、併せてご参照ください。

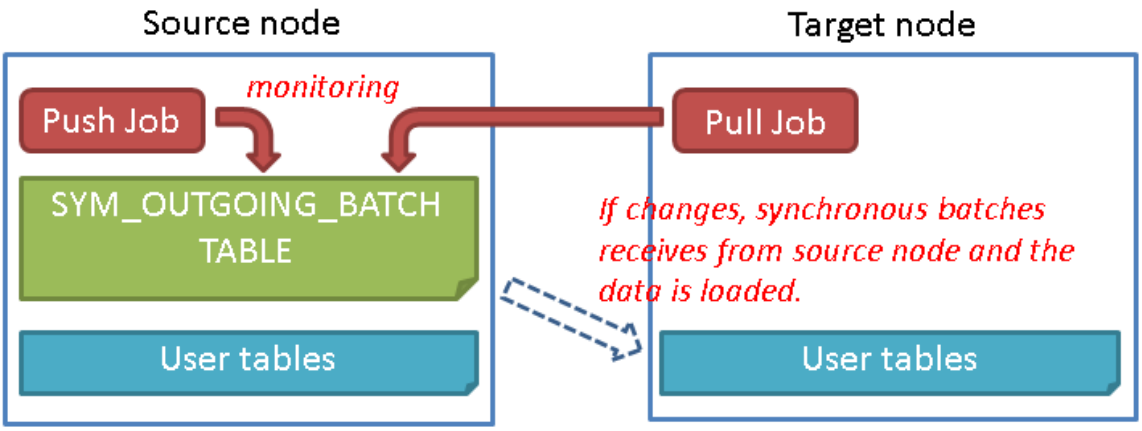
6.2. SymmetricDSを利用した異種DB連携環境の構築手順

SymmetricDSを利用して異種DB間の連携する手順は、以下の通りとなります。ここでは、SymmetricDS 3.8 Tutorials のデモのシナリオを使って説明します。

構築結果イメージは、下図になります。
ノードは、それぞれ親となるノードをRootNode、クライアントとなるノードをClientNodeと明記します。
PostgreSQLとMySQLの連携による環境構築を行っていますが、説明はPostgreSQLのみとします。



連携方式には、Push JobとPull Jobの2種類が存在します。違いは、Push Jobは更新データを連携元ノード(Source node)で監視し、Pull Jobは連携先ノード(Target node)で監視します。



6.2.1. 事前準備

事前準備は、RootNode、ClientNodeの両ノードに対して行います。

6.2.1.1. SELinuxとファイアウォールの設定

構築手順の検証で行ったSELinuxとファイアウォールの設定は、以下の通りです。

●SELinuxの設定

SELINUX=permissive

●ファイアウォールの設定

```
# firewall-cmd --permanent --add-port=9090/tcp
# firewall-cmd --reload
```

6.2.1.2. Javaのインストール

SymmetricDSは、JDK、または、JRE 7.0以降がシステム要件となっています。
インストールしていない場合には、インストールが必要です。

6.2.1.3. PL/pgSQLのインストール

必要に応じて、PL/pgSQLのインストールを行います。
なお、PostgreSQL 9.0以降はデフォルトでインストールされるため、改めてのインストールは不要です。

```
postgres=# CREATE FUNCTION plpgsql_call_handler() RETURNS language_handler AS '$libdir/plpgsql' LANGUAGE C;
postgres=# CREATE FUNCTION plpgsql_validator(oid) RETURNS void AS '$libdir/plpgsql' LANGUAGE C;
postgres=# CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql HANDLER plpgsql_call_handler VALIDATOR plpgsql_validator;
```

6.2.1.4. ユーザ作成と権限付与

SymmetricDSユーザ"corp"を作成します。

```
postgres=# create user corp;
```

必要に応じて、publicスキーマに対してSymmetricDSユーザ"corp"に権限(usage、create)を付与します。

```
postgres=# grant usage on schema public to corp;
postgres=# grant create on schema public to corp;
```

6.2.1.5. データベース作成

SymmetricDSでDB連携するデータベースを作成します。
例）sampleデータベース

```
postgres=# create database sample;
```

6.2.2. 環境構築

6.2.2.1. SymmetricDSの展開

SymmetricDSのダウンロードサイトからダウンロードした圧縮ファイルを展開します。
環境構築の検証では、SymmetricDSのホームディレクトリを/usr/local/symmetric-serverとして展開した例で説明します。

6.2.2.2. プロパティファイルを作成、配置

/usr/local/symmetric-server/engines下にプロパティファイル(.properties)を配置します。
いずれもPostgreSQLの場合の設定を記載しています。

●RootNode
corp-000.properties

リスト 6.1 RootNodeのプロパティ設定

```
engine.name=corp-000

# The class name for the JDBC Driver
db.driver=org.postgresql.Driver

# The JDBC URL used to connect to the database
db.url=jdbc:postgresql://localhost:54320/sample?protocolVersion=2&stringtype=unspecified

# The user to login as who can create and update tables
db.user=corp

# The password for the user to login as
db.password=corp

registration.url=
sync.url=http://localhost:31415/sync/corp-000

# Do not change these for running the demo
group.id=corp
external.id=000

# Don't muddy the waters with purge logging
job.purge.period.time.ms=7200000

# This is how often the routing job will be run in milliseconds
job.routing.period.time.ms=5000
# This is how often the push job will be run.
job.push.period.time.ms=10000
# This is how often the pull job will be run.
job.pull.period.time.ms=10000
# Kick off initial load
initial.load.create.first=true
```

●ClientNode
store-001.properties

リスト 6.2 ClientNodeのプロパティ設定

```
engine.name=store-001

# The class name for the JDBC Driver
db.driver=org.postgresql.Driver

# The JDBC URL used to connect to the database
db.url=jdbc:postgresql://localhost:54320/sample?protocolVersion=2&stringtype=unspecified

# The user to login as who can create and update tables
db.user=store

# The password for the user to login as
db.password=store

# The HTTP URL of the root node to contact for registration
registration.url=http://192.168.1.101:31415/sync/corp-000

# Do not change these for running the demo
group.id=store
external.id=001

# This is how often the routing job will be run in milliseconds
job.routing.period.time.ms=5000
# This is how often the push job will be run.
job.push.period.time.ms=10000
# This is how often the pull job will be run.
job.pull.period.time.ms=10000
```

6.2.2.3. SymmetricDS用のテーブルを作成

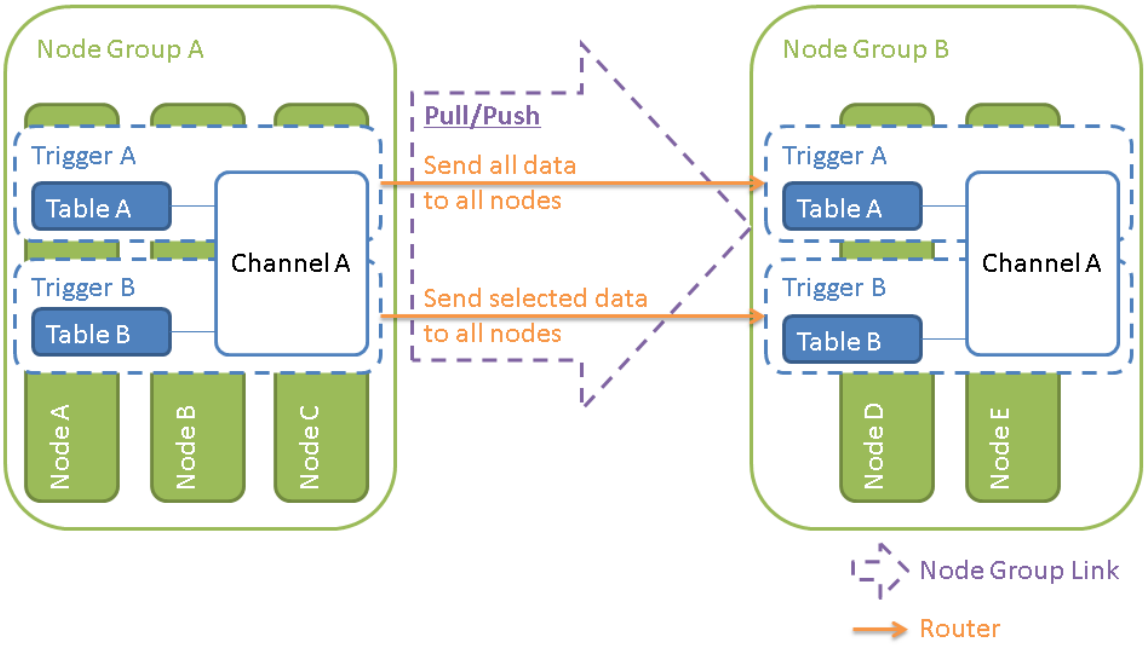
SymmetricDS用のテーブルを作成します。
RootNode側で作成しておけば、ClientNode側にも反映されます（通常のテーブルの場合は、反映されない点に注意）。

```
$ /usr/local/symmetric-server/bin/symadmin --engine corp-000 create-sym-tables
```

作成される主なテーブルは、以下の通りです。

テーブル名	役割
sym_channel	チャネルを管理するテーブル チャネルは、任意のテーブル更新を1つのトランザクションでまとめる単位を定義します。
sym_node	ノードを管理するテーブル 一意識別子となるノードIDを付与したノード情報や、どのノードグループに属するかを定義します。
sym_node_group	ノードグループを管理するテーブル ノードグループは、データを反映するノードのグルーピングを定義します。
sym_node_group_link	ノードグループ間のデータ反映方式（push or pull）を管理するテーブル
sym_trigger	トリガを管理するテーブル トリガは、監視対象のテーブルとチャネルの結びつけを定義します。
sym_router	ルータを管理するテーブル 監視対象のテーブルに変更があった場合のノードグループ間での反映の範囲 (default: テーブル全ての変更、column: 条件に従った変更のみ)を定義します。
sym_trigger_router	トリガとルータの結び付けを管理するテーブル 監視対象のテーブルに変更があった場合の反映の範囲を定義します。
sym_node_security	ノードのセキュリティを管理するテーブル ノードの認証に必要なパスワードを定義します。
sym_node_identity	RootNodeを定義するテーブル　※SymmetricDS起動時に必須。

ユーザテーブル(Table)と、Channel、Node、Node Group、Node Group Link、Trigger、Routerとの関係をイメージにした図を下図に示します。図では、左のテーブルが更新された場合の連携を示しています。



6.2.3. SymmetricDSの起動

6.2.3.1. RootNodeの起動

RootNode側のSymmetricDSを起動し、ClientNodeから待ち受けるモードにします。

```
$ /usr/local/symmetric-server/bin/sym --engine corp-000 --port 31415
$ /usr/local/symmetric-server/bin/symadmin --engine corp-000 open-registration store 001
```

6.2.3.2. ClientNode側の起動

```
$ /usr/local/symmetric-server/bin/sym --engine store-001 --port 9090
```

6.2.4. SymmetricDSによるデータ連携(動作確認)

SymmetricDSで準備されているサンプルデータを用いて、データ連携の動作確認を行います。

6.2.4.1. 連携するテーブルを作成

RootNode、ClientNodeの両ノードで、create_sample.xmlを用いて、itemテーブルなどのサンプルテーブルを作成します。

```
$ cd /usr/local/symmetric-server/samples/
$ ../bin/dbimport --engine corp-000 --format XML create_sample.xml
```

6.2.4.2. データロード

RootNodeで、insert_sample.sqlを用いて、サンプルデータと連携の設定情報をインサートします。

```
$ cd /usr/local/symmetric-server/samples/
$ ../bin/dbimport --engine corp-000 insert_sample.sql
```

実行後、ClientNode側にインサートしたデータが連携していることを確認してください。また、反対方向の連携(ClientNodeからRootNode)は転送方式が異なるた

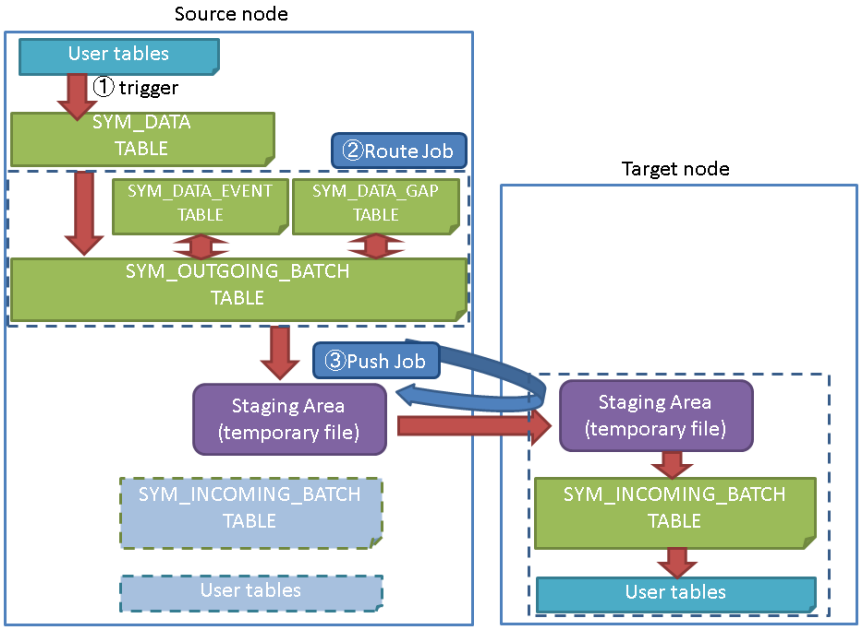
め、ClientNode側に別のデータをインサートしてみて、連携することを確認してください。

6.3. SymmetricDSの連携時の挙動

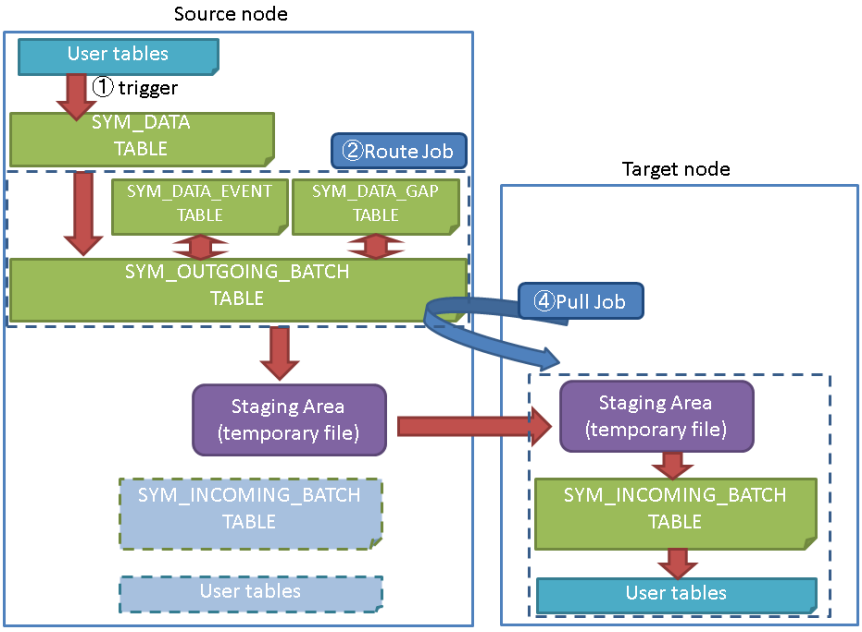
6.3.1. 挙動の概要

データ連携におけるデータの流れは、下記の通り元テーブルから宛先テーブルに伝搬します。
Push Jobの場合とPull Jobの場合をそれぞれ図に示します。

- Push Jobによる連携



- Pull Jobによる連携



6.3.1.1. データの連携機能

データの伝搬を行うにあたって、4つの主要な機能により連携を実現しています。

① トリガ
元テーブルが更新されると、トリガがSYM_DATAテーブルに更新内容を記録します。
更新内容は、CSV形式で格納します。

② Route Job
Route Jobが、SYM_DATAテーブルの更新を検知し、バッチを作成し、SYM_OUTGOING_BATCHテーブルに格納します。

その際、同一チャネル(バッチの宛先、連携方式)であれば処理をまとめられるため、その処理をSYM_DATA_EVENTテーブルで決定し、どこまで終わったかの連携管理をSYM_DATA_GAPテーブルで管理しています。

③ Push Job
SYM_OUTGOING_BATCHテーブルからStaging Areaの一時ファイルにデータを抽出し、HTTP PUTで当該ファイルを送信先ノードのStaging Areaに送ります。
当該ファイルからSYM_INCOMING_BATCHテーブルにデータをロードします。
ロードしたデータをもとに宛先テーブルを更新します。

④ Pull Job
SYM_OUTGOING_BATCHテーブルからStaging Areaの一時ファイルにデータを抽出し、HTTP GETで当該ファイルを送信先ノードのStaging Areaで受け取ります。
当該ファイルからSYM_INCOMING_BATCHテーブルにデータをロードします。
ロードしたデータをもとに宛先テーブルを更新します。
(Pull Jobは送信可能なバッチの有無に関わらずネットワークコネクションを貼る一方、Push Jobはバッチがあるときのみネットワークコネクションを貼ります。)

6.3.2. データの整合性

連携におけるデータの整合性は、送信元はSYM_OUTGOING_BATCHテーブル、送信先はSYM_INCOMING_BATCHテーブルのステータスを用いて保つようになっており、各テーブルで確認が可能です。
ステータスは以下の通りです。

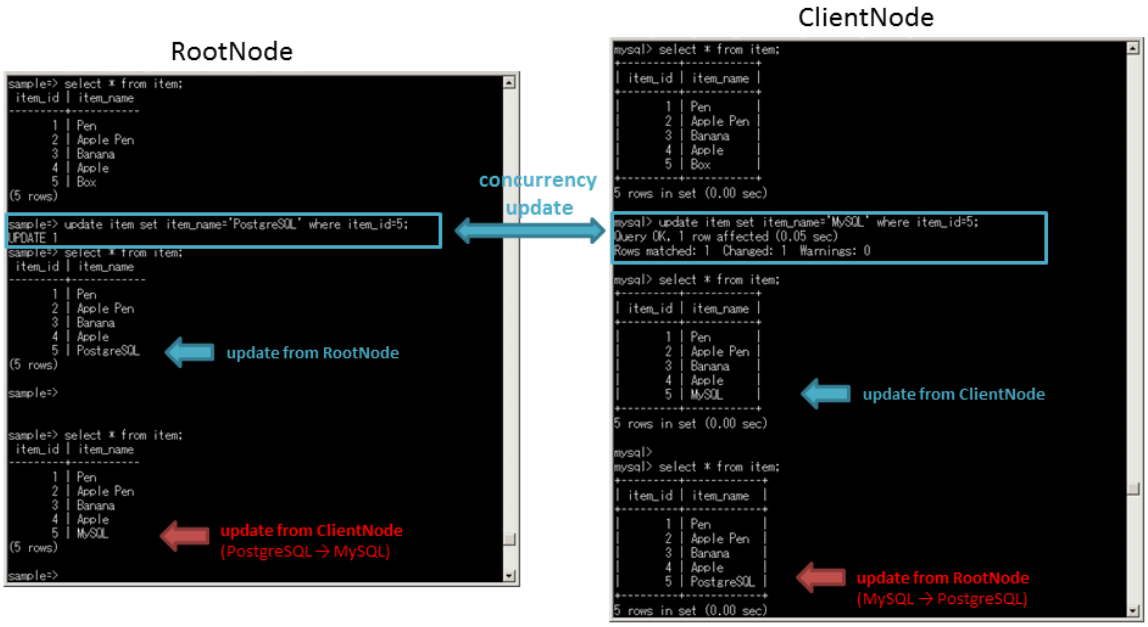
- SYM_OUTGOING_BATCHテーブルのステータス
RT ルーティング中
RQ バックグラウンドで抽出要求された
NE バッチが新しく作られてレプリケーションの準備ができた
QY データベースからクエリーされ中
SE ノードに送られた
LD ロード準備ができた
OK ロード成功のackがあった
IG 無視された
ER エラー

- SYM_INCOMING_BATCHテーブルのステータス
LD ロード中
OK ロードが成功した
ER エラー
SK スキップされた

6.3.3. データ連携における注意点

6.3.3.1. 同時更新時の注意点

データ連携の挙動を確認したところ、送信元ノードと送信先ノードで同時更新した場合には更新をそれぞれ反映されてしまったので、データの一貫性が担保できませんでした。
下図が検証結果になります。



このように、同時更新が発生すると更新が上書きされてしまうため、SymmetricDSをマルチマスタ用途として利用することは避けるべきであると考えられます。

6.4. SymmetricDSの障害対応について

6.4.1. 障害対応の概要

本節では、SymmetricDSの障害検知方法と障害回復方法を紹介します。

6.4.2. 障害検知方法

6.4.2.1. 検知対象

障害の検知対象は下表の8種類で、検知対象の閾値設定やノードグループ、ノードIDとの対応付けや有効化などの設定を行います。

検知対象	概要
cpu	CPU使用率
disk	ディスク使用率
memory	メモリ使用率
batchError	エラーになったバッチの数
batchUnsent	送信待ちのバッチ数
dataUnrouted	ルーティング待ちのバッチ数
dataGaps	ルーティング中に確認したデータギャップの数
nodesOffline	オフラインノードの数

6.4.2.2. 通知方法

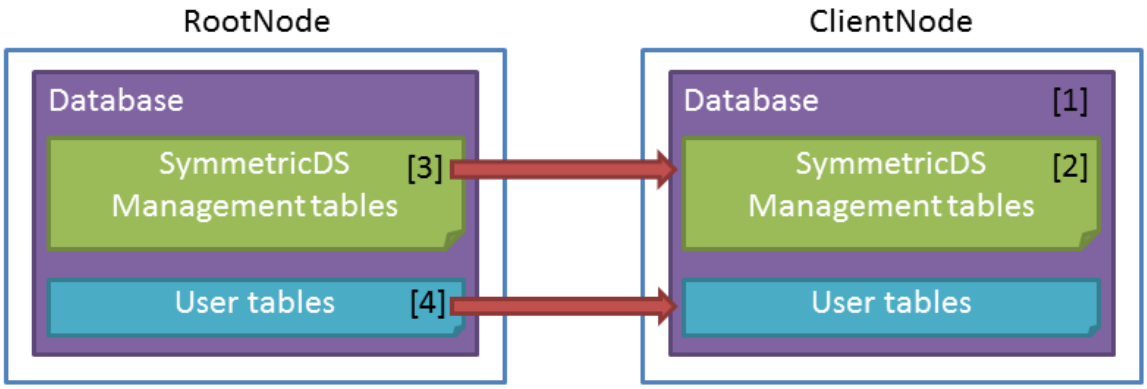
障害検知した場合の通知方法としては下記の2種類があります。こちらも、通知方法とノードグループやノードIDとの対応付けや有効化などの設定を行います。

通知方法	概要
log	ログに出力する
email	メール通知する

6.4.3. 障害回復方法

6.4.3.1. ClientNode障害時の手順

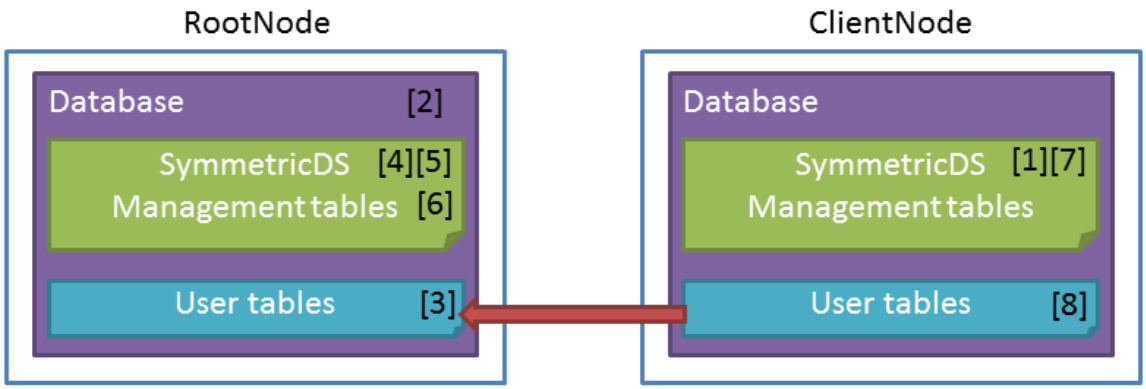
ClientNode障害時の回復手順とイメージを下記に示します。



1. ClientNode: データベースを作成
2. ClientNode: SymmetricDS起動
3. RootNode: RootNodeでClientNodeを登録
4. RootNode: スキーマの反映

6.4.3.2. RootNode障害時の手順

RootNode障害時の回復手順とイメージを下記に示します。



1. ClientNode: 反映側のPush/Pull JobをOfflineにする
2. RootNode: データベースを作成
3. RootNode: テーブルを作成
4. RootNode: SymmetricDS用テーブルの準備
5. RootNode: sym_channel, sym_node_group, sym_node_group_link, sym_trigger, sym_router, sym_trigger_router, sym_node, sym_node_security, sym_node_identityの初期値を設定する
6. RootNode: SymmetricDS起動
7. ClientNode: 1のOfflineモードを解除
8. ClientNode: データの反映

コマンド等、詳細は、SymmetricDS User Guide をご確認ください。

6.5. SymmetricDSの遅延度合の動作検証について

6.5.1. 動作検証の概要

SymmetricDSは、非同期のデータ連携であるため、どの程度の遅延が発生するか気になります。
実際に動作検証を行いました。

6.5.1.1. 検証構成

検証環境のハードウェアおよびソフトウェアの主なスペックと構成は以下の通りです。なお、検証を行うに当たって、その他の負荷はほとんど無い状態で実施しています。

ノード	項目	仕様
RootNode	CPU	Core i7-610E 2.6GHz (4 core)
	搭載メモリ	8GB
	内蔵ストレージ	160GB
	OS	RedHat Enterprise Linux 7.2
	DBMS	PostgreSQL 9.5.1
ClientNode	CPU	Core i7-610E 2.6GHz (4 core)
	搭載メモリ	8GB
	内蔵ストレージ	160GB
	OS	RedHat Enterprise Linux 7.2
	DBMS	MySQL 5.7.17

6.5.1.2. 検証方法

SymmetricDSによるデータ連携環境を構築し、1クエリ(insert文)を実行した際の遅延度合を計測しました。
計測方法は、SymmetricDSの実行ログと全クエリのレスポンス時間をログに出力・集計することで、SymmetricDSの内部処理にかかったタイムラグが確認できます。

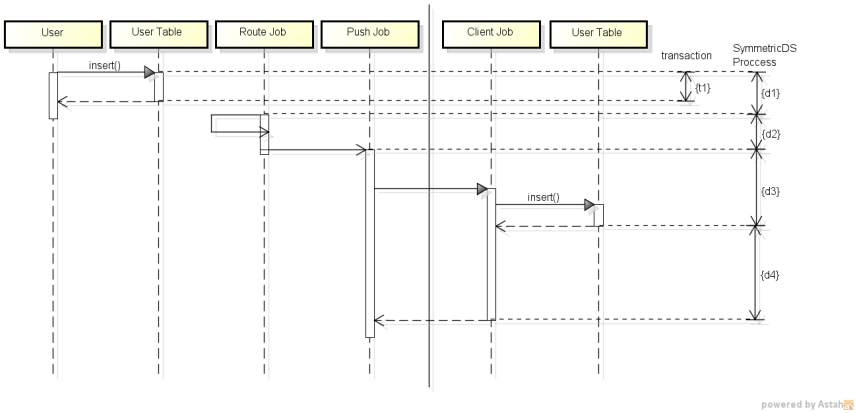
この検証に当たって、以下の設定を行いました。
●postgresql.confの設定
全クエリのレスポンス時間をログに出力する設定しました。
log_min_duration_statement = 0

●SymmetricDSのプロパティファイルの設定
JOBの処理間隔を最小値に設定する設定にしました。

```
job.routing.period.time.ms = 1
job.push.period.time.ms = 1
job.pull.period.time.ms = 1
```

6.5.1.3. 検証結果

検証結果の前に、処理の流れをシーケンス図でまとめました。
insert文のトランザクション結果をt1、SymmetricDSの処理結果がd1～d4で表しており、トランザクション時間(t1)と連携先のテーブルに書き込まれるまでの時間(d1+d2+d3)のギャップが、遅延度合ということになります。



5回検証を行った結果を下図にまとめました。

#	Phase	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)	ave. (ms)
t1	Insert Transaction	19.098	9.553	93.712	19.665	11.304	30.666
d1	Trigger Job	21.098	11.553	95.712	20.665	12.304	32.266
d2	Route Job	15.582	23.512	24.539	16.550	19.677	19.972
d3	Push (Insert) Job	91.716	48.312	76.559	69.034	325.288	122.182
d4	Push (After) Job	121.569	118.967	127.744	155.292	420.023	188.719
	async gap ((d1+d2+d3)-t1)	100.610	88.731	44.637	105.939	374.266	142.837

上記の検証環境下においては、遅延度合は1秒以内にデータ連携が完了していることがわかります。
ただし、連携後の処理(d4)に時間がかかっているため、バッチが多くなる場合には遅延度合が増える可能性があります。

6.6. SymmetricDSの利用シーンと留意点

●利用シーン
・他システムのデータを参照する必要があり、1秒程度の遅延があっても大丈夫な場合
・データ連携したいDBが、FDWに未対応だがJDBCには対応している場合

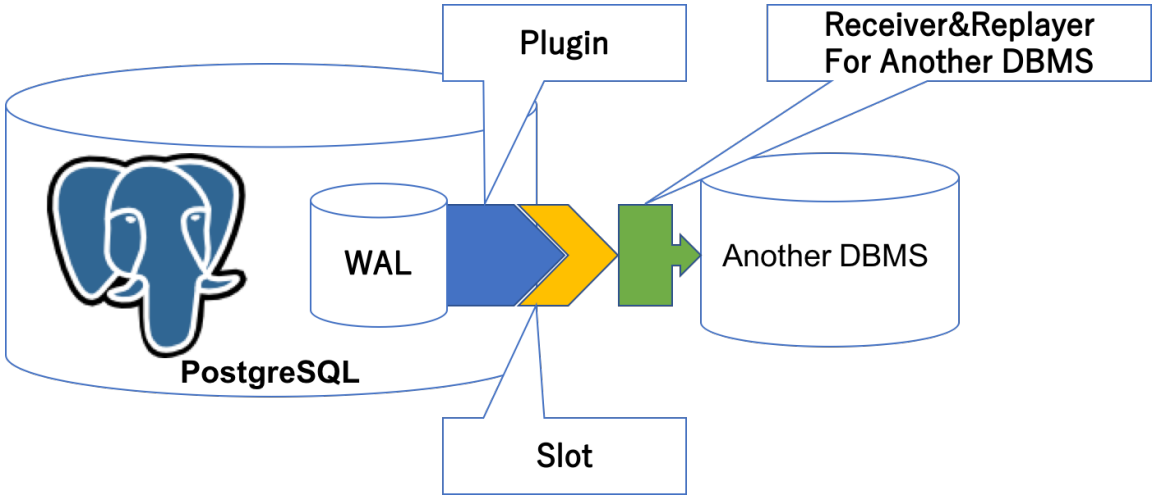
●留意点
・連携方式を選択する指標として、接続先が多く更新頻度も高いデータの連携を行いたい場合には、コネクションプール的な役割を持つPull Job方式を主方式として検討する。一方、更新頻度が高くない場合には、都度接続のPush Job方式を検討することでリソースを無駄なく使うようにする。
・連携できるのはDML(insert、update、delete)のみで、truncateやDDLはtriggerの対象外なので連携できません。
・NullPointerExceptionや強制終了など発生することが多く、品質にかなり難があると感じています。
・ユーザガイド通り設定してもできない箇所や、ユーザガイドのリンク先が誤っている箇所を10個ほど見つけました。ドキュメント品質を上げる必要があると感じています。
・コミュニティはあるがあまり積極的ではなく、重要な問題もなかなか解決しないので、使用するにあたり注意が必要です。

7. Logical Decoding

7.1. ロジカルデコーディングについて

ロジカルデコーディングは、PostgreSQL9.4から提供されている仕組みです。実行された更新処理を他のリソースへと伝播することができます。詳しくは PostgreSQL文書「ロジカルデコーディング」も参照してください。

ロジカルデコーディングによる異種DB連携は、PostgreSQLのトランザクションログ(WAL)を 独自のプラグインでデコードした更新処理をスロットから取得し、他のリソースへの反映することで実現します。



独自のプラグインなどを作成するので敷居は高いですが、他の連携方法よりも 自由度が高いことが特徴です。

たとえば、WALの内容を見て、特定のテーブルのデータのみを連携させたり、データの列名や値によるフィルタリングもプラグインの実装次第で可能です。

7.2. ロジカルデコーディングを利用した異種DB連携環境の構築手順

ロジカルデコーディングを用いる環境を構築するために必要な手順の流れを以下に示します。

(1)元となるPostgreSQLを用意する

ロジカルデコーディングの入力元となるのはWALの情報です。何はともあれ、PostgreSQLを用意する必要があります。

(2)プラグインを作成する

DB連携する異種DBにあわせて、プラグインを作成します。(プラグイン作成に必要なことを後述します)

(3)PostgreSQLの設定を変更する

デフォルトの設定のままでは、ロジカルデコーディングを利用することはできません。利用するための設定を施します。

(4)ロジカルデコーディングを実行する

SQL関数もしくはレプリケーション接続を介して、デコーディングを行います。

7.2.1. プラグインの実装例

前述の「(2)プラグインを作成する」を行うにあたって必要な情報を、contribに含まれるtest_decodingモジュールを例に、プラグインの実装を見ていきます。ロジカルデコーディングの仕組みを利用するためは、いくつかのコールバック関数を実装する必要があります。また、初期化処理でこれらのコールバック関数を登録します。

(1)初期化関数

_PG_output_plugin_initという名前で実装し、構造体に関数ポインタを登録します。構造体および初期化関数の定義は以下の通りです。

リスト 7.1 ロジカルデコーディングで利用する構造体

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit)(struct OutputPluginCallbacks *cb);
```

なお、test_decodingでの実装は、以下のようになっています。サンプルということもあり、一通りのコールバック関数を実装/登録していますが、必須なのはbegin_cb, change_cb, commit_cbです。

リスト 7.2 test_decodingの初期化処理

```
78 /* specify output plugin callbacks */
79 void
80 _PG_output_plugin_init(OutputPluginCallbacks *cb)
81 {
82     AssertVariableIsOfType(&_PG_output_plugin_init, LogicalOutputPluginInit);
83
84     cb->startup_cb = pg_decode_startup;
85     cb->begin_cb = pg_decode_begin_txn;
86     cb->change_cb = pg_decode_change;
87     cb->commit_cb = pg_decode_commit_txn;
88     cb->filter_by_origin_cb = pg_decode_filter;
89     cb->shutdown_cb = pg_decode_shutdown;
90     cb->message_cb = pg_decode_message;
91 }
```

(2)コールバック関数

初期化関数の中で登録されるコールバック関数の一覧を以下に示します。

(2-1) 開始コールバック

スロットの作成時及びデータ取得の要求の各処理があった際に呼び出されます。

リスト 7.3 開始コールバック関数のインターフェイス

```
 94 /* initialize this plugin */
 95 static void
 96 pg_decode_startup(LogicalDecodingContext *ctx, OutputPluginOptions *opt,
 97                   bool is_init)
```

データのやりとりに利用されるLogicalDecodingContext構造体(ctx)の変数にメモリを割り当てたり、OutputPluginOptions.output_typeの変更(OUTPUT_PLUGIN_TEXTUAL_OUTPUTかOUTPUT_PLUGIN_BINARY_OUTPUT)など各種オプションの設定を行います。

(2-2) 終了コールバック

各処理の最後に呼び出されます。

リスト 7.4 終了コールバック関数のインターフェイス

```
190 /* cleanup this plugin's resources */
191 static void
192 pg_decode_shutdown(LogicalDecodingContext *ctx)
```

開始コールバック関数で割り当てた領域の削除などのクリーンアップ処理を実施します。

(2-3) トランザクション開始コールバック

トランザクションの開始(BEGIN)を検出した際に呼び出されます。

リスト 7.5 トランザクション開始コールバック関数のインターフェイス

```
200 /* BEGIN callback */
201 static void
202 pg_decode_begin_txn(LogicalDecodingContext *ctx, ReorderBufferTXN *txn)
```

なお、トランザクション開始を検出した場合でも、そのトランザクションが破棄(ABORT)される 場合には、トランザクション開始コールバック関数は呼び出されません。

(2-4) トランザクション終了コールバック

トランザクションの終了(COMMIT)を検出した際に呼び出されます。

リスト 7.6 トランザクション終了コールバック関数のインターフェイス

```
224 /* COMMIT callback */
225 static void
226 pg_decode_commit_txn(LogicalDecodingContext *ctx, ReorderBufferTXN *txn,
227                      XLogRecPtr commit_lsn)
```

(2-5) 更新コールバック

データの更新(INSERT, UPDATE, DELETE)を検出した際に呼び出されます。

リスト 7.7 更新コールバック関数のインターフェイス

```
391 /*
392  * callback for individual changed tuples
393  */
394 static void
395 pg_decode_change(LogicalDecodingContext *ctx, ReorderBufferTXN *txn,
396                  Relation relation, ReorderBufferChange *change)
```

更新内容は、change->actionを見ることで判別できます。また、新旧データの内容はchange->data.tpを見ることで取得できます。

表 7.1 ロジカルデコーディングの更新処理判別

change->action	更新処理内容	新しいデータ	古いデータ
REORDER_BUFFER_CHANGE_INSERT	挿入処理(INSERT)	change->data.tp.newtuple	ー
REORDER_BUFFER_CHANGE_UPDATE	更新処理(UPDATE)	change->data.tp.newtuple	change->data.tp.oldtuple
REORDER_BUFFER_CHANGE_DELETE	削除処理(DELETE)	ー	change->data.tp.oldtuple

(2-6) オリジンフィルターコールバック

トランザクションの開始/終了及びデータの更新を検出した際、前述のコールバック関数に先駆けて 呼び出されます。

リスト 7.8 オリジンフィルターコールバック関数のインターフェイス

```
247 static bool
248 pg_decode_filter(LogicalDecodingContext *ctx,
249                  RepOriginId origin_id)
```

オリジンの設定を行っておくことで、デコードすべきか否かを判断することが可能で、双方向のレプリケーション時などにデータの伝播が循環し続けてしまうことを抑制できます。オリジンの設定は、後述のSQL関数で設定できます。

(2-7) 汎用メッセージコールバック

汎用メッセージを検出した際に呼び出されます。

リスト 7.9 汎用メッセージコールバック関数のインターフェイス

```
479 static void
480 pg_decode_message(LogicalDecodingContext *ctx,
481                   ReorderBufferTXN *txn, XLogRecPtr lsn, bool transactional,
482                   const char *prefix, Size sz, const char *message)
```

汎用メッセージは、後述のSQL関数で発行することができます。

7.2.2. 設定例

ロジカルデコーディングの仕組みを利用するため、最低限以下の設定を変更する 必要があります。

(1)wal_level

wal_levelパラメータでWALに含まれる情報の量を変更します。デフォルトの"minimal"から"logical"にします。

ex) wal_level = logical

なお、wal_levelに指定できる値は、上記"minimal", "logical"のほか、"replica"があり、"logical", "replica", "minimal"の順でWALに含まれる情報の量は多くなります。PostgreSQL9.5以前の"archive"/"hot_standby"は"replica"として扱われます。

(2)max_replication_slot

max_replication_slotパラメータでデータベースクラスタで作成できるスロットの最大数を変更します。デフォルトの0から1以上の値にします。

ex) max_replication_slot = 1

また、ロジカルデコーディングの実行をレプリケーション接続を介して行う場合は、上記に加えて、以下の設定を変更する必要があります。

(3)max_wal_senders

max_wal_sendersパラメータでデータベースクラスタで起動できるwal senderプロセスの最大数を変更します。デフォルトの0から1以上の値にします。

ex) max_wal_senders = 1

(4)レプリケーション接続の許可

レプリケーション接続を許可するため、pg_hba.confにreplicationのレコードを追加します。

ex) 認証方式の選択は適宜行ってください。

リスト 7.10 レプリケーション接続の許可

```
local   replication   postgres                trust
host    replication   postgres   127.0.0.1/32 trust
host    replication   postgres   ::1/128      trust
```

7.2.3. 実行例

ロジカルデコーディングを実行する方法としては、以下の2通りがあります。

(1)SQL関数を介して実行

SQL関数を実行することで、ロジカルデコーディングを実行します。

(2)レプリケーション接続を介して実行

同期レプリケーション機能とロジカルデコーディングを併用する場合には、レプリケーション接続を介した実装、設定、実行を行う必要があります。レプリケーションプロトコルを利用したり、連携先の状態を連携元に伝えるなど の処理を実装する必要があります。
http://www.postgresql.jp/document/9.6/html/protocol-replication.html

説明を簡単にするため、ここではSQL関数を用いた実行例を見ていきます。設定後に実施する大まかな手順の流れは以下の通りです。

(1)スロットの作成

まずpg_create_logical_replication_slot関数を実行して、スロットを作成します。

pg_create_logical_replication_slot(slot_name name, plugin name)

関数の引数として、スロット名(slot_name)とプラグイン名(plugin)を指定します。戻り値は、スロット名(slot_name)とトランザクション位置(xlog_position)です。

リスト 7.11 スロットの作成

```
testdb=# select pg_create_logical_replication_slot('test_slot',
'test_decoding');
LOG:  logical decoding found consistent point at 0/152E108
DETAIL:  There are no running transactions.
STATEMENT:  select pg_create_logical_replication_slot('test_slot',
'test_decoding');
 pg_create_logical_replication_slot
------------------------------------
 (test_slot,0/152E140)
(1 row)
```

なお、作成したスロットはpg_replication_slotsビューで確認できます。また、スロットの 削除はpg_drop_replication_slot関数で行えます。

リスト 7.12 スロットの確認と削除

```
testdb=# select * from pg_replication_slots ;
-[ RECORD 1 ]-------+--------------
slot_name           | test_slot
plugin              | test_decoding
slot_type           | logical
datoid              | 16384
database            | testdb
active              | f
active_pid          |
xmin                |
catalog_xmin        | 880
restart_lsn         | 0/152E108
confirmed_flush_lsn | 0/152E140


testdb=# select pg_drop_replication_slot('test_slot');
 pg_drop_replication_slot
--------------------------

(1 row)
```

(2)デコーディングの実行

スロットの作成が完了したら、そのスロットからWALを読み込みデコーディングしていきます。pg_logical_slot_get_changes関数、pg_logical_slot_peek_changes関数を実行することで、デコーディングを実行できます。

pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])

pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])

関数の引数として、スロット名(slot_name)、トランザクション位置(upto_lsn)、トランザクション数(upto_nchanges)、プラグインで利用されるオプション(options)を指定します。upto_lsn, upto_nchangeでトランザクション位置及びトランザクション数によるデコード対象の絞り込みが行えます。両方ともNULLが指定されている場合は全ての結果が返されます。pg_logical_slot_get_changes関数は一度デコードされたデータは再度デコードされません。一方、pg_logical_slot_peek_changes関数は一度デコードされたデータも再度デコードの対象となります。戻り値は、トランザクション位置(location)、トランザクションID(xid)、デコードしたデータ(data)です。

リスト 7.13 スロットの確認と削除

```
testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL,
NULL);
 location  | xid |                 data
-----------+-----+---------------------------------------
 0/152E4F8 | 880 | BEGIN 880
 0/152E4F8 | 880 | table public.tbl: INSERT: i[integer]:1
 0/152E6B8 | 880 | COMMIT 880
(3 rows)

testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL,
NULL);
 location  | xid |                 data
-----------+-----+---------------------------------------
 0/152E4F8 | 880 | BEGIN 880
 0/152E4F8 | 880 | table public.tbl: INSERT: i[integer]:1
 0/152E6B8 | 880 | COMMIT 880
(3 rows)

testdb=# select * from pg_logical_slot_get_changes('test_slot', NULL, NULL);
 location  | xid |                 data
-----------+-----+---------------------------------------
 0/152E4F8 | 880 | BEGIN 880
 0/152E4F8 | 880 | table public.tbl: INSERT: i[integer]:1
 0/152E6B8 | 880 | COMMIT 880
(3 rows)

testdb=# select * from pg_logical_slot_get_changes('test_slot', NULL, NULL);
 location | xid | data
----------+-----+------
(0 rows)
```

なお、処理結果をbytea型で出力する場合(OutputPluginOptions.output_typeをOUTPUT_PLUGIN_BINARY_OUTPUTと したプラグインの場合)には、pg_logical_slot_get_binary_changes関数、pg_logical_slot_peek_changes関数で 実行できます。

基本的な利用方法は上記のみです。以降はオリジンフィルター処理とメッセージ処理の実行方法について見ていきます。

(3)オリジンフィルター処理

オリジンの設定を行っておくと、WALの中に含まれるオリジンに関する情報を利用できるようになります。プラグインのオリジンフィルターコールバック関数は、この情報を元にデコードを行うか否かを 判別できるようになります。オリジンの作成はpg_replication_origin_create関数で行います。

```
pg_replication_origin_create(node_name text)
```

関数の引数は、オリジン名(node_name)です。戻り値は、そのオリジンを一意に特定するID(oid)です。

オリジンの設定はpg_replication_origin_session_setup関数で行います。

```
pg_replication_origin_session_setup(node_name text)
```

関数の引数は、オリジン名(node_name)です。戻り値はありません(void)。

オリジンの削除はpg_replication_origin_drop関数で行います。

```
pg_replication_origin_drop(node_name text)
```

関数の引数は、オリジン名(node_name)です。戻り値はありません(void)。

リスト 7.14 オリジンの作成、設定、削除

```
testdb=# select pg_replication_origin_create('test_origin');
 pg_replication_origin_create
------------------------------
                            1
(1 row)


testdb=# select pg_replication_origin_session_setup('test_origin');
 pg_replication_origin_session_setup
-------------------------------------

(1 row)


testdb=# select pg_replication_origin_drop('test_origin');
 pg_replication_origin_drop
----------------------------

(1 row)
```

test_decodingでは、デコーディングのオプションとしてonly-localオプションを指定できます。このオプションを指定するとオリジンが設定されたセッションから実行された更新処理を デコードの対象から外すようになります。

リスト 7.15 オリジンフィルター処理

```
testdb=# begin;
BEGIN
testdb=# insert into tbl values (1000);
INSERT 0 1
testdb=# commit;
COMMIT
testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL,
NULL);
 location   | xid |                   data
------------+-----+-------------------------------------------
 0/152F880 | 888 | BEGIN 888
 0/152F880 | 888 | table public.tbl: INSERT: i[integer]:1000
 0/152FAD8 | 888 | COMMIT 888
(3 rows)

testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL, NULL,
'only-local', 'on');
 location | xid | data
----------+-----+------
(0 rows)

-- IN ANOTHER SESSION --------------------------------------
testdb=# begin;
BEGIN
testdb=# insert into tbl values (1001);
INSERT 0 1
testdb=# commit;
COMMIT
------------------------------------------------------------

testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL, NULL,
'only-local', 'on');
 location   | xid |                   data
------------+-----+-------------------------------------------
 0/152FBF0 | 889 | BEGIN 889
 0/152FBF0 | 889 | table public.tbl: INSERT: i[integer]:1001
 0/152FE88 | 889 | COMMIT 889
(3 rows)
```

(4)汎用メッセージ処理

汎用メッセージにより、WALに独自の文字列を含めることができます。プラグインの汎用メッセージコールバック関数は、この情報を元に独自の処理を行うことが できます。汎用メッセージの発行はpg_logical_emit_message関数で行います。

> pg_logcial_emit_message(transactional bool, prefix text, content text) pg_logcial_emit_message(transactional bool, prefix text, content bytea)

関数の引数は、トランザクションの一部か否かを表すフラグ(transactional)、判別に利用する接頭辞(prefix)、メッセージ本体(content:textもしくはbytea)です。戻り値はトランザクション位置(pg_lsn)です。

リスト 7.16 汎用メッセージ処理

```
testdb=# select pg_logical_emit_message(false, 'postgres', 'Hello,
LogicalDecoding');
 pg_logical_emit_message
-------------------------
 0/15301D0
(1 row)

testdb=# select * from pg_logical_slot_peek_changes('test_slot', NULL,
NULL);
 location   | xid |                                          data
------------+-----+-----------------------------------------------------------------
-------------------------
 0/15301D0 |   0 | message: transactional: 0 prefix: postgres, sz: 22
content:Hello, LogicalDecoding
(1 row)
```

7.3. ロジカルデコーディングの今後

ロジカルデコーディングを用いた正式なソリューション、まだPostgreSQL本体には含まれていません。このため、本機能を利用するためには、自らプラグインを作成するか、世の中に公開されている プラグインを適用する必要があります。導入に際しては、十分に検証を行った上で判断してください。

なお、2017年9月にリリース予定のPostgreSQL10では、ロジカルデコーディングの機構を ベースにした「ロジカルレプリケーション」機能が本体に追加される見込みです。現時点ではPostgreSQLからPostgreSQLへのデータ連携がメインの機能ですが、この機能の追加により、様々な異種DB間の連携ソリューションが加速することが期待されます。

8. 外部データラッパ(FDW)を用いた連携方式

本章では、PostgreSQLの外部データラッパを用いた異種データベース連携の方式に関して説明します。

8.1. 外部データラッパ(FDW)について

PostgreSQLはSQL/MED仕様を部分的に実装しており、PostgreSQLの外部にあるデータ(「外部データ」と呼びます)に対して SQL文を利用してアクセスすることが可能になります。

外部データには、外部データラッパ(以降、FDW)を通じてアクセスします。外部データラッパは、外部データと通信できるライブラリであり、接続の確立やデータ取得の機能を抽象化します。

本章では、特に異種データベースのスキーマ、テーブルに格納されたデータを「外部データ」の対象にし、異種データベースとの連携方法としてFDWを評価することを目的としています。

なお、2015年度のPGEConsの成果物 DB選定基準編(WG2) にも、PostgreSQLの特長的な機能として、外部データラッパの調査結果が整理されていますので、併せてご参照ください。

8.2. FDWを利用した異種DB連携環境の構築手順

FDWを利用して異種DBと連携する手順は、大きく以下の通りです。本節では、異種DBとしてOracle Database 11gと、外部データラッパとしてoracle_fdwを利用します。

1. Oracle Database向けのクライアントのセットアップ
    - 外部データラッパから利用するOracleクライアントの入手およびインストール
    - Oracleクライアントが利用する環境変数の設定
    - OracleクライアントからのOracle Databseへの接続確認
2. 異種データベース向けの外部データラッパのセットアップ
    - 異種データベース向けの外部データラッパの入手
    - コンパイルおよびインストール
3. 外部データへのアクセス手順
    - 外部サーバの作成(CREATE SERVER)
    - 外部サーバで利用するユーザ権限定義
    - PostgreSQLユーザと外部サーバのユーザの関連付け(CRTEATE USER MAPPING)
    - 外部テーブルの作成(CREATE FOREIGN TABLE)
4. 外部データへのアクセス
    - SQL文の記載方法
5. その他の機能
    - Oracleのスキーマ配下のテーブル定義を外部テーブルとしてインポートする(IMPORT FOREIGN SCHEMA)

前出の2015年度のPGEConsの成果物の FDW(外部データラッパ)のアセスメント にOracleクライアントのセットアップ手順、oracle_fdwのセットアップ手順が記載されています。本節では、外部データラッパの利用に共通で必要な、PostgreSQL内での構築手順(3.,4.,5.)を中心に記載します。

8.2.1. 外部データへのアクセス手順

8.2.1.1. 外部サーバの作成(CREATE SERVER)

以下のパラメータを利用して外部サーバの定義を実施します。

パラメータ名	値
外部サーバ名	remote_oracle
利用する外部データラッパ名	oracle_fdw
接続先	192.168.10.2
ポート番号	1521
データベース名	XE

以下のSQL文を実行し、外部サーバを定義します。

```
testdb=# create server remote_oracle FOREIGN DATA WRAPPER oracle_fdw options (dbserver
'192.168.10.2:1521/XE');
```

8.2.1.2. 外部サーバで利用するユーザ権限定義

必要に応じて、外部サーバに接続するユーザに権限を付与します。

```
testdb=# GRANT USAGE ON FOREIGN SERVER remote_oracle TO oracle_user;
```

8.2.1.3. PostgreSQLユーザと外部サーバのユーザの関連付け(CRTEATE USER MAPPING)

PostgreSQLユーザと外部サーバのユーザの関連付けます。

```
testdb=# create user mapping for oracle_user server server remote_oracle options(user 'scott',
password 'tiger');
```

8.2.1.4. 外部テーブルの作成(CREATE FOREIGN TABLE)

ここでは、Oracle上のoracle_fdw_testという名前のテーブルに対し、PostgreSQL側ではoracle_testという名前で外部テーブルを作成することで対応付けます。

```
testdb=# create foreign table oracle_table(id int, name text) server remote_oracle options(table
'oracle_fdw_test');
```

8.2.2. 外部データへのアクセス

8.2.2.1. SQL文の記載方法

定義した外部テーブルに対し、ローカルのテーブルアクセスと同様にSQL文を記述します。以下に例を示します。

```
postgres=# select id from oracle_table ;
 id
----
  1
  2
  3
  4
  5
  6
  7
  8
(8 rows)
```

8.2.3. その他の機能

8.2.3.1. Oracleのスキーマ配下のテーブル定義を外部テーブルとしてインポートする(IMPORT FOREIGN SCHEMA)

IMPORT FOREIGN SCHEMAは、外部サーバ上に存在するテーブルを指す外部テーブルを作成するための構文です。

PostgreSQL 文書のIMPORT FOREIGN SCHEMAの記述 も参照してください。

以下は、Oracleのスキーマ「SYSTEM」配下のテーブルを外部テーブル定義としてインポートする例です。(スキーマ名SYSTEMは大文字ですので、二重引用符で囲む必要がある点に注意してください)

```
postgres=# create schema test_schema;
CREATE SCHEMA
postgres=# IMPORT FOREIGN SCHEMA "SYSTEM" FROM SERVER remote_oracle INTO test_schema;
IMPORT FOREIGN SCHEMA
```

システムカタログ「pg_foreign_table」を利用して、外部テーブル定義が実施されていることを確認します。

```
postgres=# select * from pg_foreign_table order by ftoptions limit 10;
 ftrelid | ftserver |                 ftoptions
---------+----------+-------------------------------------------
   36702 |    36065 | {schema=SYSTEM,table=AQ$DEF$_AQCALL}
   36179 |    36065 | {schema=SYSTEM,table=AQ$DEF$_AQCALL}
   36182 |    36065 | {schema=SYSTEM,table=AQ$DEF$_AQERROR}
   36705 |    36065 | {schema=SYSTEM,table=AQ$DEF$_AQERROR}
   36185 |    36065 | {schema=SYSTEM,table=AQ$_DEF$_AQCALL_F}
   36708 |    36065 | {schema=SYSTEM,table=AQ$_DEF$_AQCALL_F}
   36188 |    36065 | {schema=SYSTEM,table=AQ$_DEF$_AQERROR_F}
   36711 |    36065 | {schema=SYSTEM,table=AQ$_DEF$_AQERROR_F}
   36714 |    36065 | {schema=SYSTEM,table=AQ$_INTERNET_AGENTS}
   36191 |    36065 | {schema=SYSTEM,table=AQ$_INTERNET_AGENTS}
(10 rows)
```

8.3. FDWを利用する上での留意点

8.3.1. 外部データ参照の確認(実行計画)

外部テーブルへのアクセスの確認方法として、実行計画を参照する方法があります。以下は、通常のテーブルへのアクセス時の実行計画です。

```
postgres=# explain analyze select count(*) from pgbench_accounts;
                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=28480.42..28480.43 rows=1 width=0) (actual time=652.621..652.621 rows=1 loops=1)
   ->  Index Only Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.42..25980.42 rows=1
000000 width=0) (actual time=0.063..394.224 rows=1000000 loops=1)
         Heap Fetches: 0
 Planning time: 0.520 ms
 Execution time: 653.024 ms
 (5 rows)
```

以下は、外部テーブルへのアクセス時の実行計画です。テーブルへのアクセス方法として「Foreign Scan」が選択されています。

```
postgres=# explain analyze select count(*) from pgbench_accounts_remote;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=220.92..220.93 rows=1 width=0) (actual time=3768.933..3768.934 rows=1 loops=1)
   ->  Foreign Scan on pgbench_accounts_remote  (cost=100.00..212.39 rows=3413 width=0) (actual
time=1.761..3486.258 rows=1000000 loops=1)
 Planning time: 0.260 ms
 Execution time: 3769.992 ms
 (4 rows)
```

8.3.2. プッシュダウン

対象にする異種DBに対応するFDWの実装によっては、SQL文のWHERE句の条件に応じて 連携先データベース上で処理が実施されます(「プッシュダウン」と呼びます)。これにより、連携先のデータベースのリソースを有効に利用できます。

例を以下に示します。

```
postgres=# explain(analyze,verbose) select * from pgbench_accounts_remote where aid < 10 order by
bid;
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------
----------------------------------------
 Sort  (cost=116.37..116.55 rows=72 width=352) (actual time=19.894..19.895 rows=9 loops=1)
   Output: aid, bid, abalance, filler
   Sort Key: pgbench_accounts_remote.bid
   Sort Method: quicksort  Memory: 26kB
   ->  Foreign Scan on public.pgbench_accounts_remote  (cost=100.00..114.15 rows=72 width=352)
(actual time=19.869..19.872 rows=9 loops=1)
         Output: aid, bid, abalance, filler
         Remote SQL: SELECT aid, bid, abalance, filler FROM public.pgbench_accounts WHERE ((aid <
10))
 Planning time: 7.645 ms
 Execution time: 59.947 ms
(9 rows)
```

同じ定義のテーブルをローカルに作成し、同様のSQL文を実行した場合の実行計画は、以下の通りです。

```
postgres=# explain(analyze,verbose) select * from pgbench_accounts where aid < 10 order by bid;
                                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------------
-------------------------------------------------
 Sort  (cost=8.74..8.76 rows=9 width=97) (actual time=0.240..0.243 rows=9 loops=1)
   Output: aid, bid, abalance, filler
   Sort Key: pgbench_accounts.bid
   Sort Method: quicksort  Memory: 26kB
   ->  Index Scan using pgbench_accounts_pkey on public.pgbench_accounts  (cost=0.43..8.59 rows=9
width=97) (actual time=0.011..0.019 rows=9 loops=1)
         Output: aid, bid, abalance, filler
         Index Cond: (pgbench_accounts.aid < 10)
 Planning time: 1.557 ms
 Execution time: 0.472 ms
(9 rows)
```

WHERE句(WHERE aid < 10)の絞り込みが連携先のデータベースに対して投入されるため、連携先のリソースを利用できます。さらに、WHERE句で絞り込んだ結果のみを連携元に返却しているため、連携先、連携元を流通するデータ量が削減できています。

8.3.3. ストアドプロシージャ、ユーザ定義ファンクションの実行

異種DB上に配置されているストアドプロシージャやユーザファンクションは、FDWを通じて直接実行することができません。そのため、FDWを経由せず、連携先の異種DBに対して直接SQL文を実行します。例を以下に示します。

連携先(Oracle)での操作

```
-- Oracle上でユーザ定義ファンクションを作成
SQL> CREATE FUNCTION test_function(dt IN NUMBER) RETURN NUMBER
  2  IS
  3  d NUMBER;
  4  BEGIN
  5  d := dt * 2;
  6  RETURN d;
  7  END;
  8  /
```

連携元(PostgreSQL)での操作

```
-- 連携元(PostgreSQL)から外部テーブルにアクセス

postgres=# select id from oracle_table ;
 id
----
  1
  2
  3
  4
  5
  6
  7
  8
(8 rows)

-- 外部テーブル向けのアクセスになっていることを実行計画から確認

postgres=# explain analyze select id from oracle_table ;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
-----------------
 Foreign Scan on oracle_table  (cost=10000.00..20000.00 rows=1000 width=4) (actual
time=0.510..0.548 rows=8 loops=1)
   Oracle query: SELECT /*a973056b69582205b8a9e0cb679a7325*/ "ID" FROM "ORACLE_FDW_TEST"
 Planning time: 0.723 ms
 Execution time: 0.611 ms
(4 rows)

--Oracle上で作成したファンクションを実行

postgres=# select test_function(id) from oracle_table ;
ERROR:  function test_function(integer) does not exist
LINE 1: select test_function(id) from oracle_table ;
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type
casts.
postgres=# select id from oracle_table ;
```

連携先(Oracle)での操作

```
-- Oracle上で作成したファンクションを実行
SQL> select test_function(2) from dual;

TEST_FUNCTION(2)
----------------
               4
```

9. 著者

（企業・団体名順）

版	所属企業・団体名	部署名	氏名
第1.0版 (2016年度WG3)	NTTテクノクロス株式会社	クラウド＆セキュリティ事業部	勝俣 智成
	NTTテクノクロス株式会社	クラウド＆セキュリティ事業部	原田 登志
	日本電気株式会社	クラウドプラットフォーム事業部	川畠 輝聖
	日本電気株式会社	クラウドプラットフォーム事業部	湯村 昇平
	株式会社日立製作所	OSSソリューションセンタ	稲垣 毅
	株式会社日立製作所	OSSソリューションセンタ	田畑 義之
	富士通株式会社	ミドルウェア事業本部	山本 貢嗣