PostgreSQL論理デコード入門 — PostgreSQLから変更データを取り出すしくみ

株式会社SRA OSS
馬 雪テイ

会社紹介

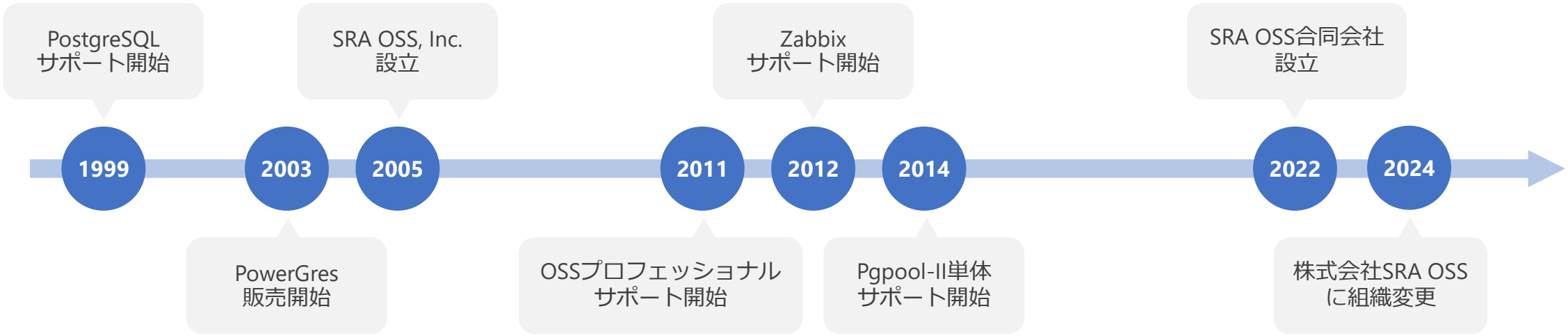
株式会社SRA OSS

所 在 地: 東京都豊島区南池袋2-32-8
設 立 日: 2022年6月17日
株　　主: 株式会社SRA
株式会社NTTデータ
資 本 金: 7,000万円
社　　長: 稲葉 香理

事業内容

- オープンソースソフトウェア (OSS) 関連のサポート、製品開発・販売、構築・コンサル
- OSSの教育、開発、コミュニティ運営支援
- ソフトウェアの研究開発

顧　　問: 石井 達夫
技術顧問: 増永 良文 (お茶の水女子大学名誉教授)



自己紹介

名前: 馬 雪テイ

所属: システムインフラ開発室

担当業務:

- PostgreSQLサポート
- PostgreSQL HAクラスタ構築・コンサルティング
- 社内システム開発

概要

- 背景
- 論理デコードとは
- 論理デコードの仕組み
- 出力プラグイン
- 簡単な実践
- 応用の方向性

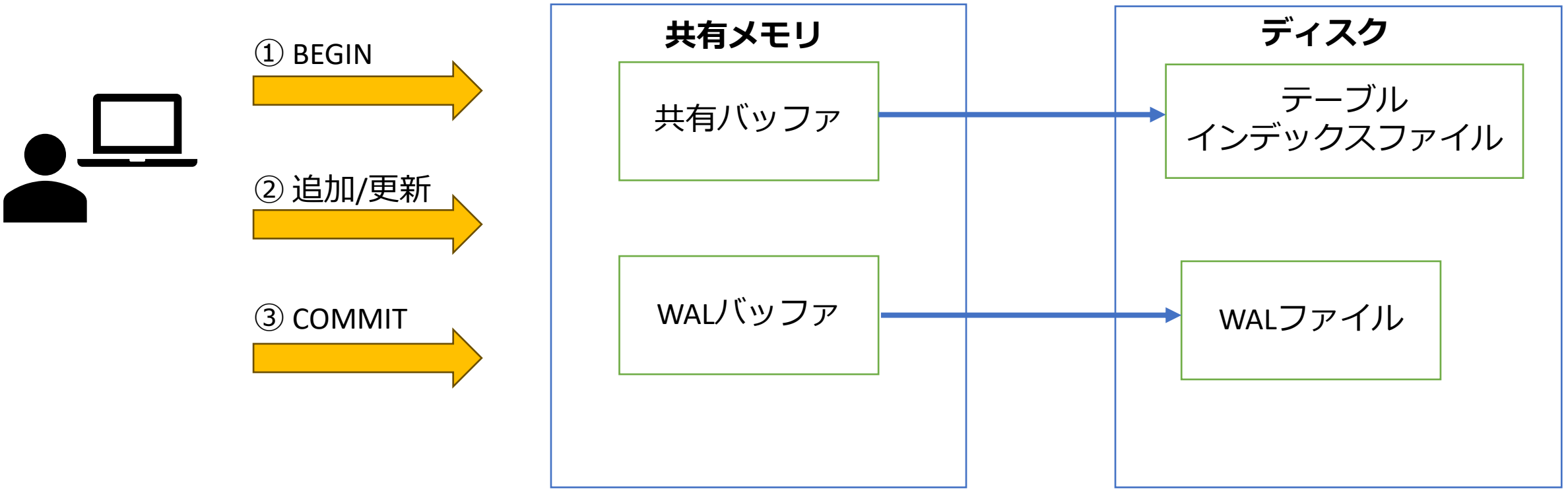
今日のゴール

- 「論理デコードがどのように動くか」
- 「どの設定をすれば有効になるか」
- 「簡単な実践テストを行えるようになる」

背景

WAL の復習

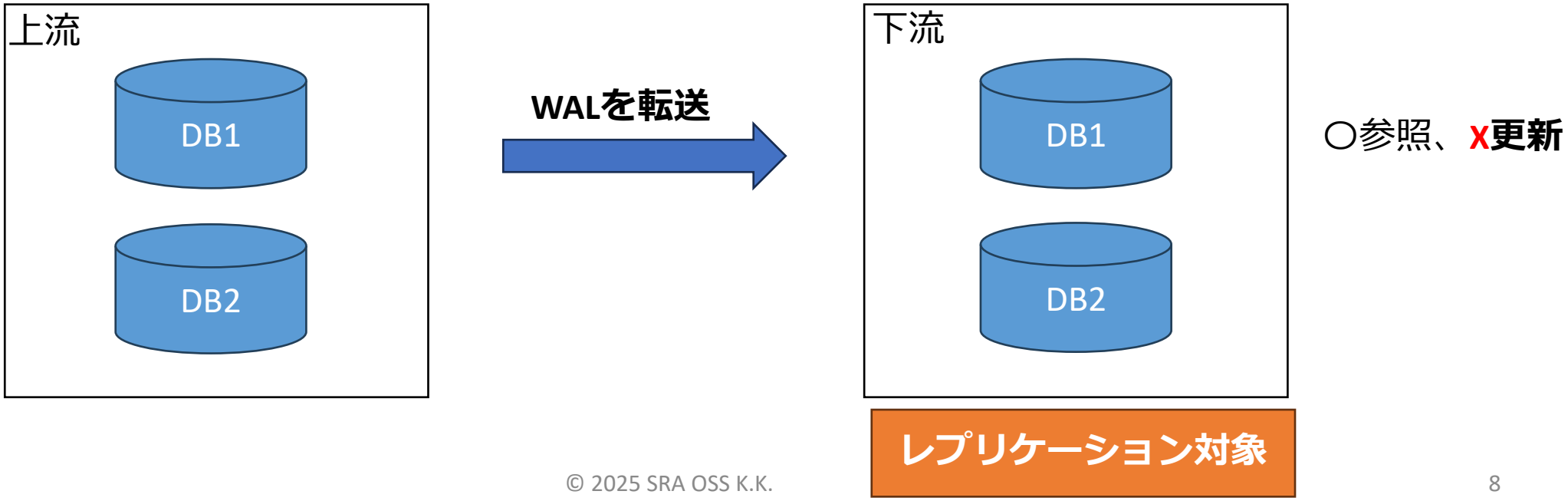
PostgreSQLのトランザクションの開始・終了とデータの変更情報のログを先にディスクへ記録すること。



レプリケーション方式の復習

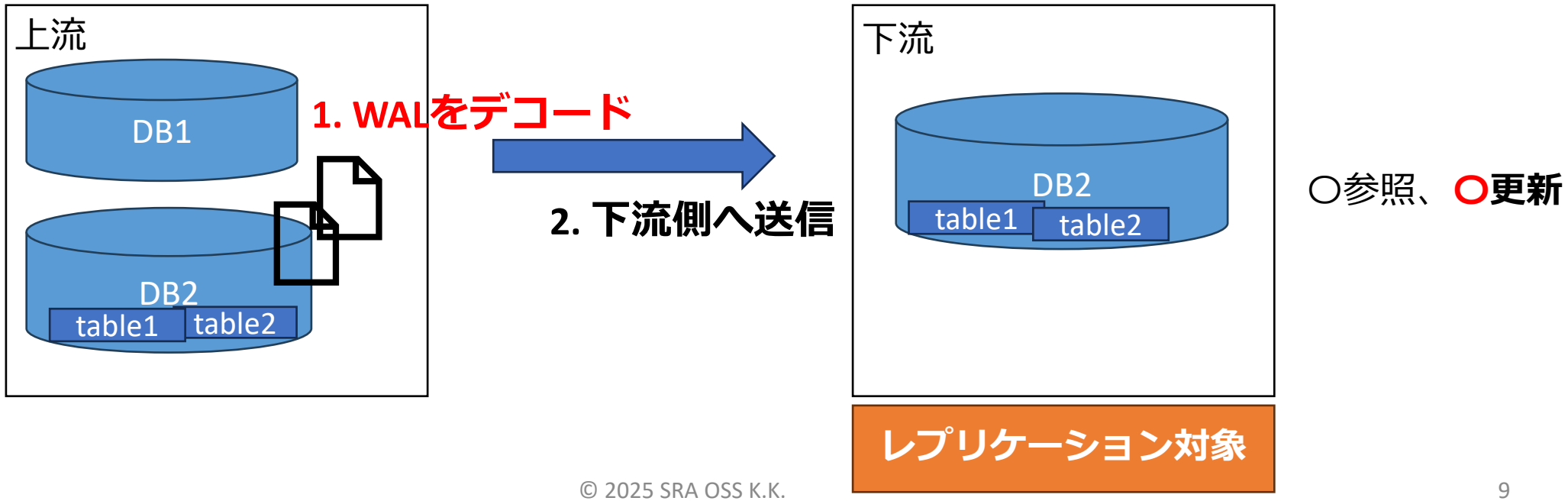
- 物理レプリケーション
  - 全DBを対象に、WALを丸ごと転送して複製する方式
  - 下流側では参照のみ（更新不可）
  - PostgreSQLのメジャーバージョンは同じである必要

📌 **用途：高可用性（HA）、災害対策（DR）**



レプリケーション方式の復習

- 論理レプリケーション
  - テーブル単位でレプリケーション対象を指定できる
  - WAL をデコードし、細かい単位で変更イベントとして下流へ送信
  - 下流側でも更新が可能（双方向構成も可能）
  - 異なるメジャーバージョン間でのレプリケーションも可能



なぜ “論理デコード” が必要なのか？

PostgreSQL の中では、
すべての変更（INSERT/UPDATE/DELETE）は WAL に書かれてから反映される。

しかし…

- WAL は**バイナリ形式**で、人間にもアプリにも読めない
- 物理レプリケーションでは**丸ごとコピーするだけ**
- 「アプリ側は、どの行が更新されたか知りたいケースが多い」
  例）監査ログ、差分同期、リアルタイム分析

=> その「別の仕組み」が **論理デコード（Logical Decoding）**。

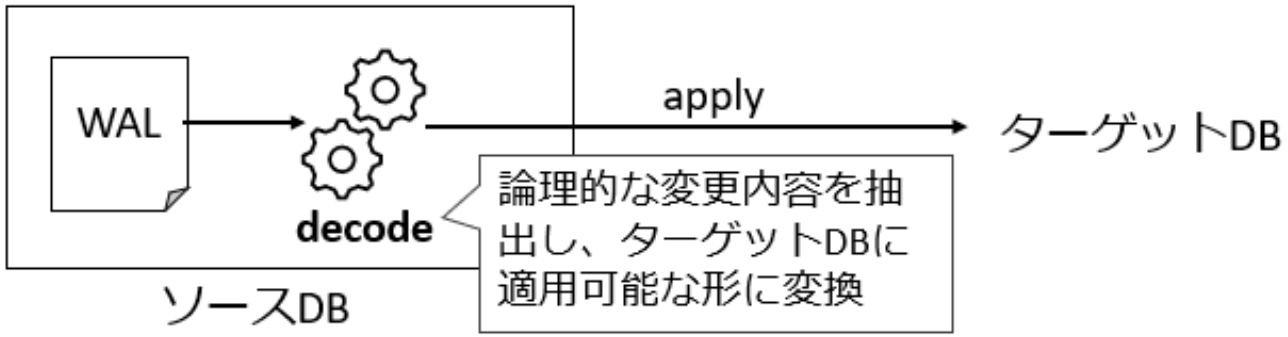
WAL から “意味のある変更情報” を取り出す翻訳装置と思えばよい。

何がうれしい？

論理デコードを使うと：

- 変更内容を細かい単位で取得できる
- JSON など好きな形式に変換できる
- PostgreSQL → 外部システムに通知できる
- 監査ログ・差分抽出・ETL の構築が可能

=> PostgreSQL を “他システムとつながるデータベース” に変えるコア技術



論理デコードとは

論理デコードとは

- WALに記録された DB 変更内容を PostgreSQL 外でも読める形式に変換して出力する。出力形式は使用するoutput pluginによって選択。

```
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('1');
postgres=*# INSERT INTO data(data) VALUES('2');
postgres=*# COMMIT;
```

(test_decodingプラグイン)

実行例では、
pg_logical_slot_get_changes() で
WAL を読みやすい形式に変換

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                     data
-----------+-------+-----------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)
```

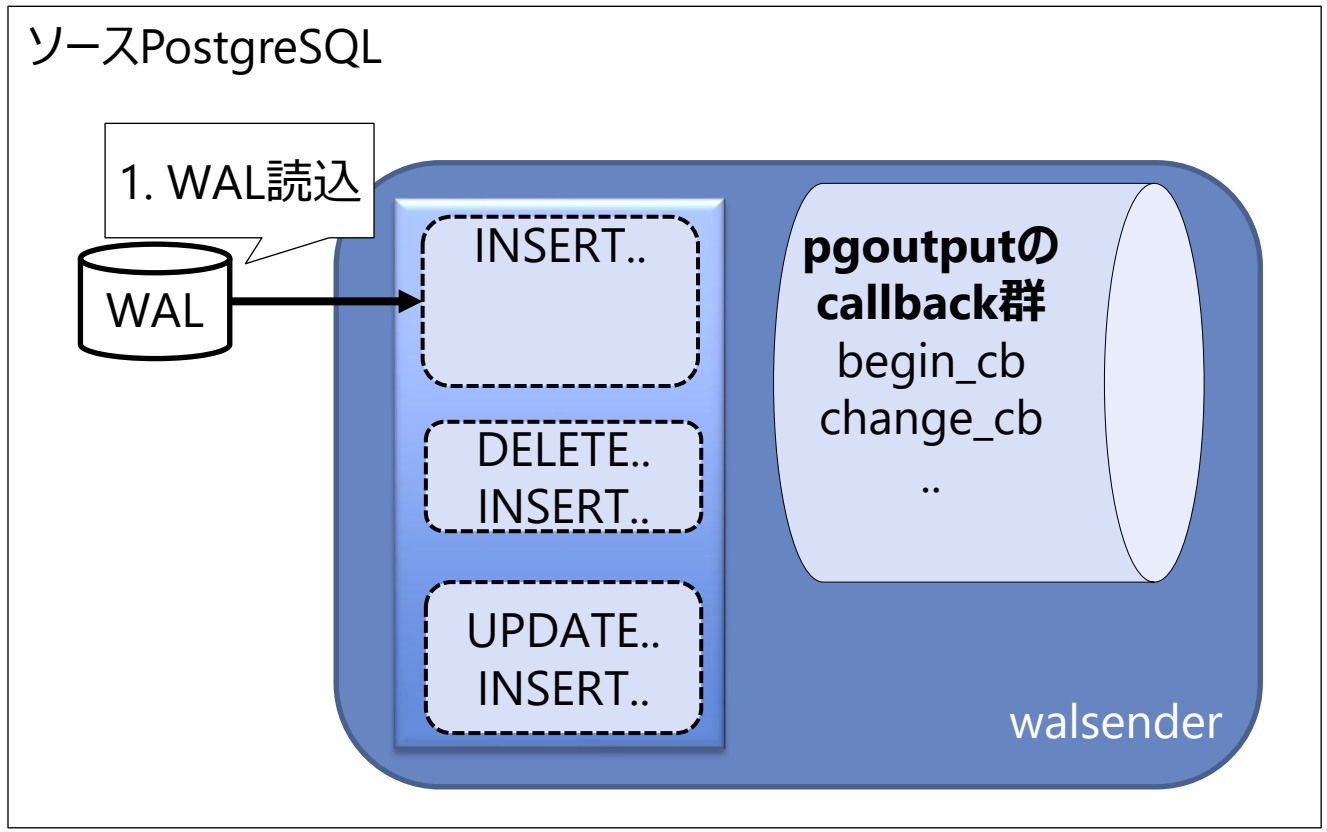
論理デコードの仕組み

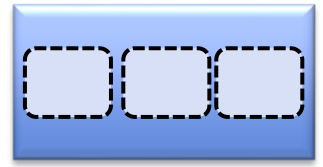
仕組み

トランザクションの変更
　　｜ (INSERT/UPDATE/DELETE)
　　▼
WAL に記録
　　｜
　　▼
WAL の変更情報を**元のトランザクション単位に並び替える**
　　｜
　　▼
Output Plugin で変換
　　｜
　　▼
レプリケーションスロットから変更データを取得する

PostgreSQLでのデータ処理

ロジカルデコーディングを実施。pgoutputを利用するので、PostgreSQL本体の論理レプリケーションと同じ処理となる。以下既存トランザクションに１件UPDATE・COMMIT実施した例:



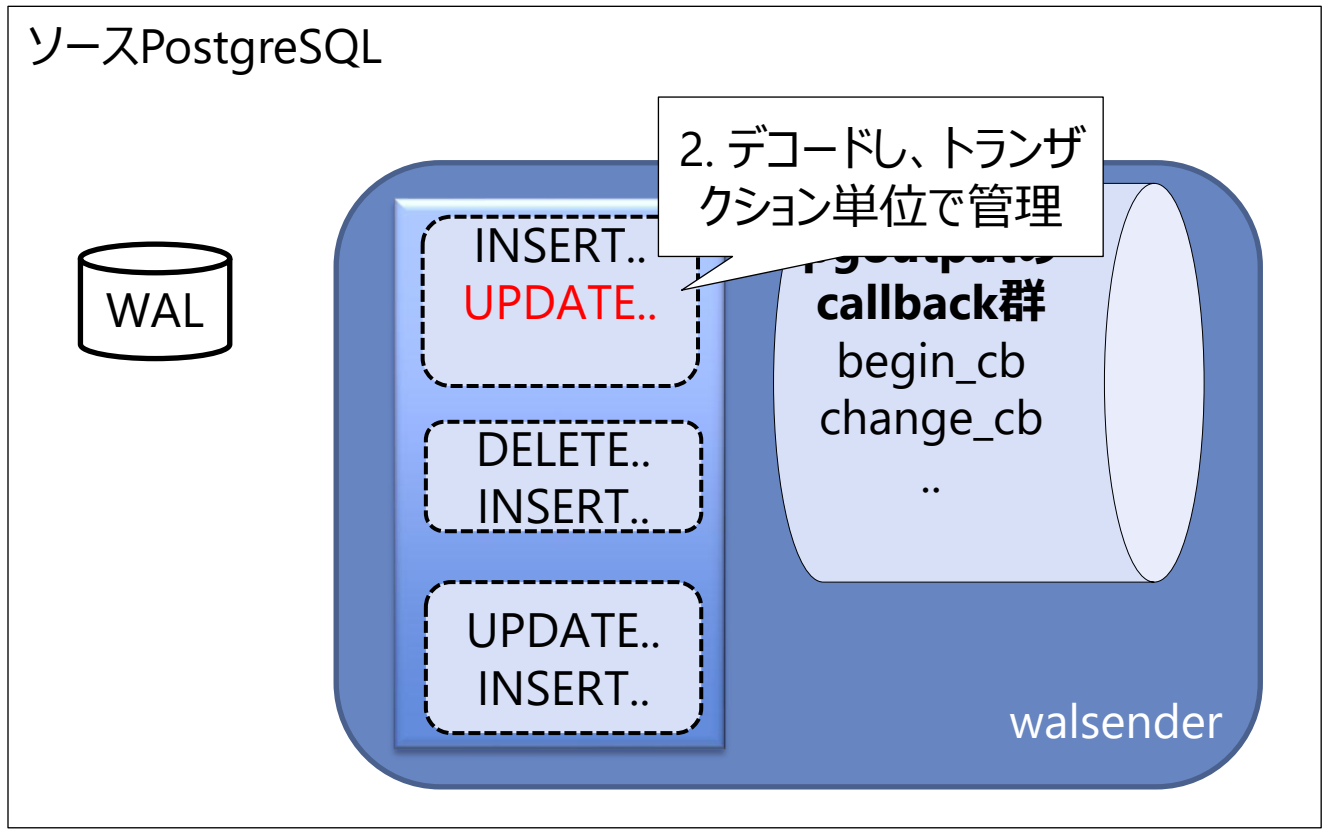


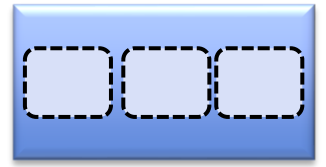
ReorderBuffer。
更新内容をトランザクションごとに管理

outputプラグイン。ここではpgoutput

PostgreSQLでのデータ処理

ロジカルデコーディングを実施。pgoutputを利用するので、PostgreSQL本体の論理レプリケーションと同じ処理となる。以下既存トランザクションに１件UPDATE・COMMIT実施した例:



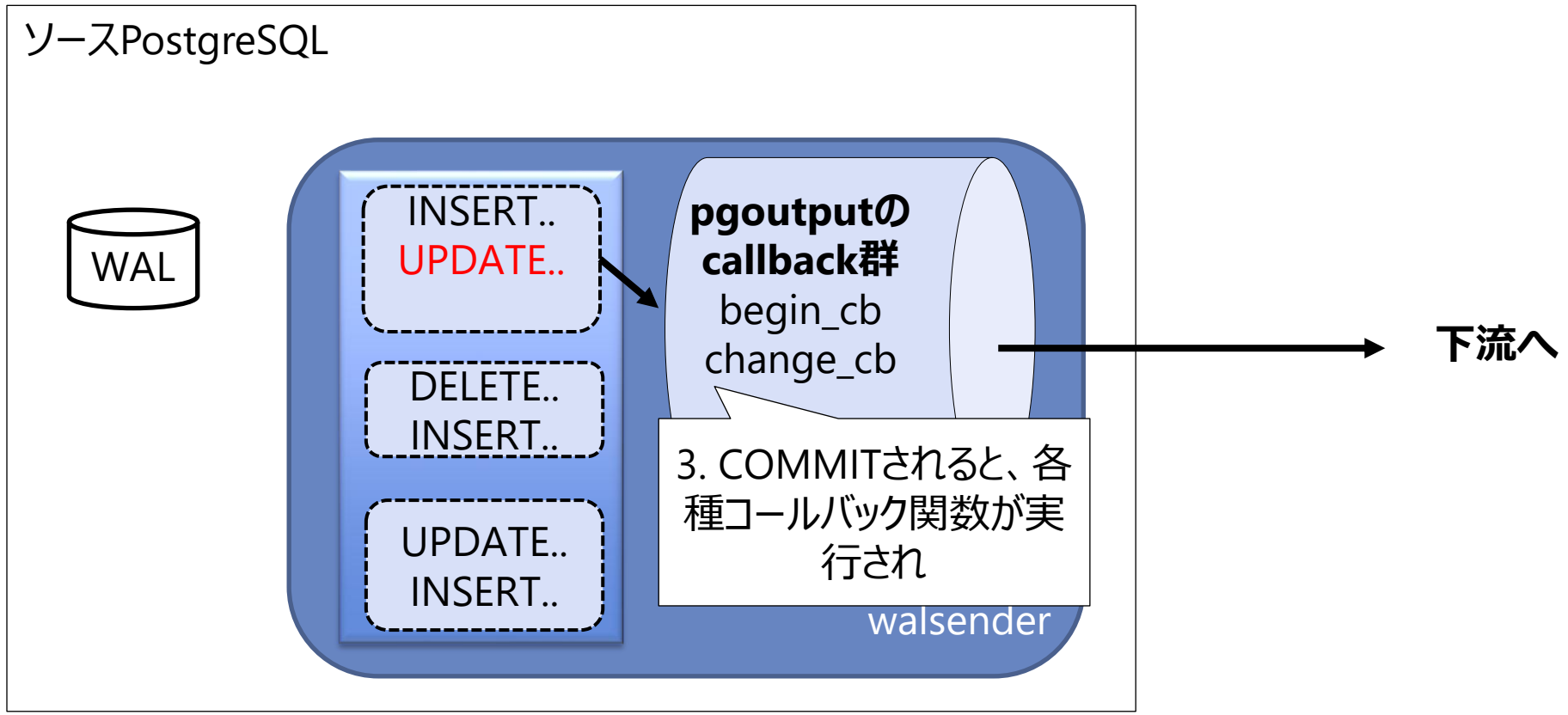


ReorderBuffer。
更新内容をトランザクションごとに管理

outputプラグイン。ここではpgoutput

PostgreSQLでのデータ処理

ロジカルデコーディングを実施。pgoutputを利用するので、PostgreSQL本体の論理レプリケーションと同じ処理となる。以下既存トランザクションに１件UPDATE・COMMIT実施した例:



データの取得方法

- **ストリーミングプロトコル**
  - レプリケーション接続専用のコマンドで制御する
    （通常の SQL 接続では使えない）

CREATE_REPLICATION_SLOT
→ 指定したプラグイン用の論理レプリケーションスロットを作る

DROP_REPLICATION_SLOT
→ スロットを削除（WAL 保持を解放）

START_REPLICATION SLOT ... LOGICAL
→ スロットから変更をストリーミング配信する

```
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

- **pg_recvlogical**
  - これらのコマンドを内部で呼び出すラッパツール
  - コマンドラインから論理デコードを試すときに便利

データの取得方法

- **SQL インターフェイス**
  - SQL コマンド経由で変更データを取得できる
  - 通常の SQL 接続で実行可能（レプリケーション接続不要）
  - シンプルな検証・学習用途に便利

関数	役割
pg_create_logical_replication_slot()	スロット作成
pg_logical_slot_get_changes()	変更データを取得（消費して前に進む）
pg_logical_slot_peek_changes()	変更データを取得（スロット位置は進めない）
pg_drop_replication_slot()	スロット削除

レプリケーション管理関数： https://www.postgresql.jp/docs/17/functions-admin.html#FUNCTIONS-REPLICATION

論理デコードの前提条件

- WALレベルを logical に設定（変更内容を取得可能にするため）
- 論理レプリケーションスロットを作成できる状態

  → max_replication_slots と max_wal_senders の設定が必要
- 出力プラグインの選択（pgoutput / test_decoding / 他）
- 主キーまたは一意キーの存在（更新/削除の識別に必須）
- pg_hba.conf で replication ロールのアクセスを許可

出力プラグイン

出力プラグイン：test_decoding

- PostgreSQL の 標準バンドル（contrib）に含まれており、デフォルトで同梱されている
- デコードされた変更イベントをテキスト形式で確認できる
- 本番で利用しない。検証・デバッグ用。

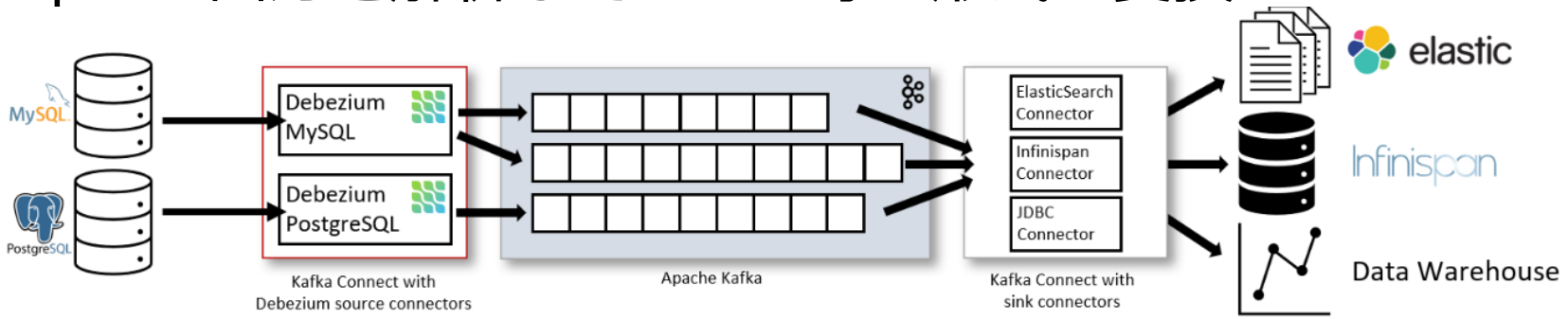
```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-xids', '0');
 location   | xid |                    data
------------+-----+--------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

出力プラグイン： pgoutput

- PostgreSQLの標準的な論理デコーディング出力プラグイン
- PostgreSQL 10以降で導入され、PostgreSQLのネイティブな論理レプリケーション機能の基盤として使用されている

代表的な利用例

- PostgreSQL → PostgreSQL
  （論理レプリケーション / サブスクライバーへ apply）
- 一部CDCツールでも利用(例 Debezium)
  → pgoutputの出力を解析してJSON 等の形式に変換



https://debezium.io/documentation/reference/stable/architecture.html

出力プラグイン：wal2json

- wal2json

https://github.com/eulerto/wal2json

➡ JSON形式で出力するプラグイン。アプリケーション側で直接JSONを扱えるため扱いやすく、自作アプリなどで利用されている。


```
{
    "change": [
        {
            "kind": "insert",
            "schema": "public",
            "table": "table1_with_pk",
            "columnnames": ["a", "b", "c"],
            "columntypes": ["integer", "character varying(30)", "timestamp without time zone"],
            "columnvalues": [1, "Backup and Restore", "2018-03-27 11:58:28.988414"]
        }
        ,{
            "kind": "insert",
            "schema": "public",
            "table": "table1_with_pk",
            "columnnames": ["a", "b", "c"],
            "columntypes": ["integer", "character varying(30)", "timestamp without time zone"],
            "columnvalues": [2, "Tuning", "2018-03-27 11:58:28.988414"]
        }
        ,{
            "kind": "insert",
            "schema": "public",
            "table": "table1_with_pk",
            "columnnames": ["a", "b", "c"],
            "columntypes": ["integer", "character varying(30)", "timestamp without time zone"],
            "columnvalues": [3, "Replication", "2018-03-27 11:58:28.988414"]
        }
        ,{
            "kind": "message",
            "transactional": true,
            "prefix": "wal2json",
            "content": "this message will be delivered"
        }
```

出力プラグインを比較する

	test_decoding	pgoutput
利用目的	学習・検証・デバッグ用	実運用向け / 論理レプリケーションの基盤として利用
出力形式	人間が読めるテキスト形式	構造化された内部形式
想定利用シーン	ローカルで動作確認・動作理解	PostgreSQL → PostgreSQL、CDCツールなど
主な利用例	内部管理関数を使って変更データを確認	サブスクライバーへ配信 / 外部ツールへ橋渡し
導入難易度	contrib に同梱、すぐ利用可能	PostgreSQL ネイティブ（10以降）

簡単な実践

test_decoding を使って論理デコードを試す

① スロット作成（前提条件は設定済）

```
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot', 'test_decoding', false,
true);
   slot_name    |    lsn
----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn, confirmed_flush_lsn
FROM pg_replication_slots;
   slot_name    |    plugin     | slot_type | database | active | restart_lsn | confirmed_flush_lsn
----------------+---------------+-----------+----------+--------+-------------+---------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   | 0/16A4440
(1 row)
```

test_decoding を使って論理デコードを試す

② 変更状況を見る（まだ変更がない）

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)
```

③ テーブル作成

```
postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE
```

④ DDLはレプリケーションされないので、見えるのはトランザクションだけ

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)
```

test_decoding を使って論理デコードを試す

⑤ 変更が読み込まれると、それらは消費され、次の呼び出して送出されない

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)
```

⑥ 実際のデータを入れる

```
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('1');
postgres=*# INSERT INTO data(data) VALUES('2');
postgres=*# COMMIT;
```

test_decoding を使って論理デコードを試す

⑦ 変更を見る

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                     data
-----------+-------+-----------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)
```

test_decoding を使って論理デコードを試す

⑧ 出力プラグインにオプションを渡すことで、フォーマットに影響を与えることができる

```
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL, 'include-timestamp',
'on');
    lsn    |  xid  |                    data
-----------+-------+---------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)
```

⑨ スロットを削除する

```
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
----------------------


(1 row)
```

レプリケーション管理関数

関数名	動作	WALの位置の扱い
pg_logical_slot_get_changes()	変更データを取得し、**読み進める**	読み取った位置まで**消費**される
pg_logical_slot_peek_changes()	変更データを取得するが、**読み進めない**	WAL の位置は**維持されたまま**

※
peek = 中身を確認するだけ
get = 中身を取り出して前進させる
誤って get してしまうと、その位置に戻せない（Slot 再作成が必要）

レプリケーション管理関数

```
postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- 変更を消費せずに変更ストリームを先読みすることもできる
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn     |  xid  |                      data
------------+-------+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- pg_logical_slot_peek_changes()の次の呼び出しでも同じ変更が返される
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn     |  xid  |                      data
------------+-------+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)
```

モニタリング

- 未使用のレプリケーションスロットは必ず削除

→ 放置すると WAL が蓄積し、ストレージ枯渇の危険

- スロットが“読み手なし”になるとログが消費されない

→ サブスクライバーの停止・設定ミスに注意

- **pg_replication_slots** で状態を監視（V14以降）

→ active / restart_lsn などを確認

```
=- SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]---------
slot_name        | sub1
plugin           | pgoutput
slot_type        | logical
datoid           | 5
database         | postgres
temporary        | f
active           | t
active_pid       | 4477
xmin             | [NULL]
catalog_xmin     | 849
restart_lsn      | 0/4816C1E8
confirmed_flush_lsn | 0/4816C220
wal_status       | reserved
safe_wal_size    | [NULL]
two_phase        | f
```

制限事項

① DDL（CREATE / ALTER TABLE など）は対象外

対策：

→ DDLのレプリケーションには別仕組みが必要

② PostgreSQLの **REPLICA IDENTITY** は、論理デコードで「どの列をキーとして行を識別するか」を決める

→ REPLICA IDENTITYがDEFAULTのまま、かつ主キーや一意キーがないテーブルでは、UPDATE／DELETE 時にbefore値（旧値）を取得できないため、UPDATE・DELETEがエラーとなる

対策：

→ 主キーまたは一意キーを設定する

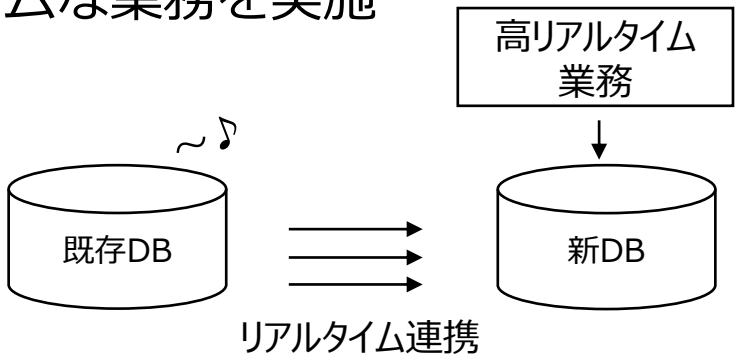
→ REPLICA IDENTITYにFULLを指定して全列を出力。（※性能への影響が大きくなる可能性あり）

応用の方向性

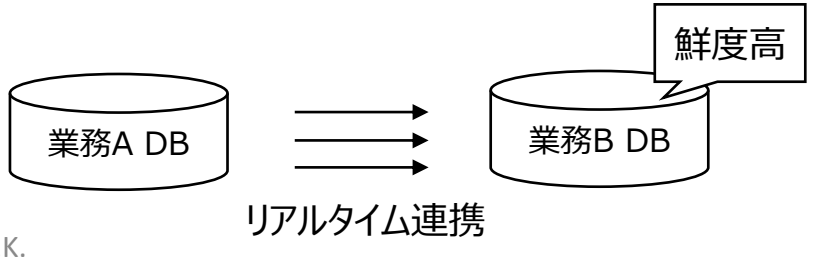
応用例

- CDC（Change Data Capture）
  → 論理デコードの抽出結果をリアルタイム活用する仕組み

  - データを新DBへ同期し、新DBにてリアルタイムな業務を実施



  - 日次バッチ処理をCDCに置換、高鮮度のデータで業務を実施



今日のゴール（再び）

- 「論理デコードがどのように動くか」
- 「どの設定をすれば有効になるか」
- 「簡単な実践テストを行えるようになる」

まとめ

まとめ

- PostgreSQL の WAL にはすべての変更が記録されている

- 論理デコードは WAL から意味のある変更を抽出する仕組み
  （INSERT / UPDATE / DELETE を細かい単位で取得できる）

- 変更内容は出力プラグインを通して外部へ連携可能
  - 学習・検証用：test_decoding
  - 実運用・CDC / 論理レプリケーション：pgoutput など

- PostgreSQL を **“他システムとつながるデータベース”** に進化させる重要機能

➡ まず test_decoding で小さく試してみるのがおすすめ

参考資料

- PostgreSQL Documentation
  https://www.postgresql.org/docs/devel/logicaldecoding.html

- 速習！論理レプリケーション@PostgreSQL Conference Japan 2022
  https://www.slideshare.net/slideshow/postgresql-logical-replication-postgresql-conference-japan-2022-nttdata/254219341

ご清聴ありがとうございました。

製品・サービスに関するお問い合わせ: sales@sraoss.co.jp 03-5979-2701

SRA OSS

– RECRUIT –

【 PostgreSQLエキスパート】
【 インフラ系OSSミドルウェアエンジニア 】

OSSのように、オープンで情熱的であれ。

全世界のエンジニア・OSSコミュニティの揺るぎない信念とたゆまぬ努力に敬意を。
私たちも情熱を持ち、会社の枠を超えてオープンであり続けます。

Our Value

- ■ OSS貢献
- ■ カスタマーファースト
- ■ マルチジョブ/マルチキャリア

株式会社SRA OSS 人事担当
✉ personnel@sraoss.co.jp

業務内容、待遇、カジュアル面談などお気軽にお問合せください。

SRA OSS