リカバリからBDRまでを支えるPostgreSQL内部技術

鈴木幸市

April 2021

EDB

EDB

PostgreSQLの振る舞いの理解のために

内容:

- データベースエンジンの概要
- リカバリ、バックアップ、ログ
- 物理レプリケーション
- 論理レプリケーション
- BDR

About Me

2018年までNTTグループでした

- 主な仕事:
  - ITCへの日本語導入 (EUC, 電子メールへの日本語導入など)
  - Oracle の Unix 移植
  - UniSQL リレーショナルデータベース
  - NTTオープンソースセンタ (データベース関連グループ)
    - PostgreSQL開発プロモーション
    - Postgres-XC/XL

2019年 2ndQuadrant 入社

2020年 EDBへ異動

- 現在、サポート及び分散トランザクション廻りのリサーチをやっています

Why today’s talk?

なぜこのような話題を？

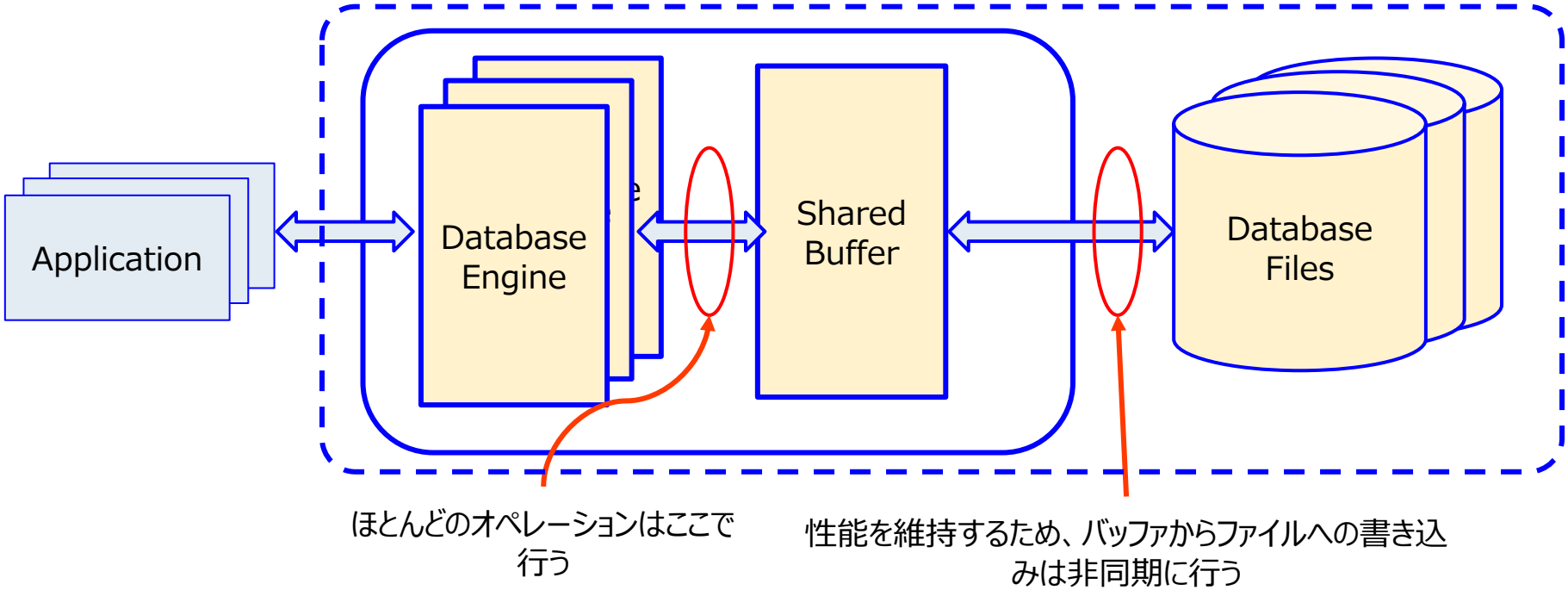
- PostgreSQL の振る舞いの理解の一助
  - 原理がわかると「書いていないこと」も想像がつく
  - 「書いていないこと」がどうなっているのか質問できる
- 方式、実装の適用限界が想像できる
  - これはあぶなそう
  - これは大丈夫そう
- サポートや技術担当とのやり取りが円滑になる

EDB

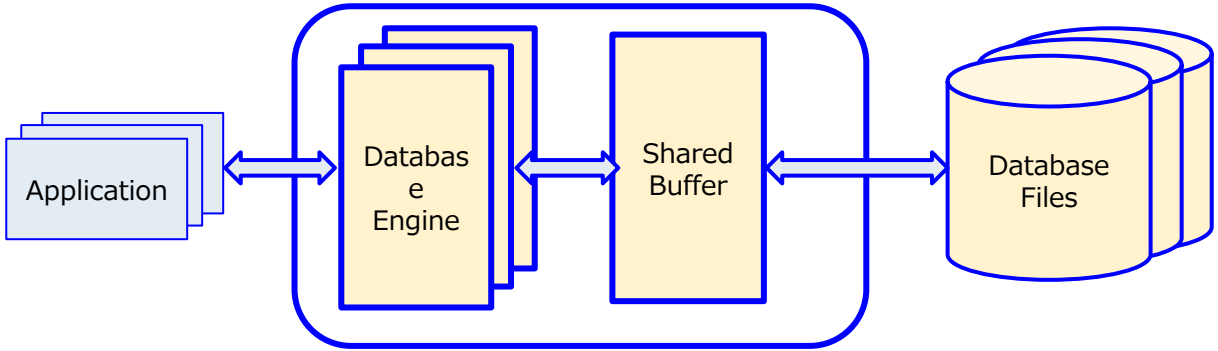
データベースの仕掛け

超単純バージョン

超単純化したデータベースの内部構造



これでいいの？



- トランザクションが失敗したらどうやって巻き戻す？
- サーバ電源が落ちたりデータベースがクラッシュしたら共有バッファの内容は失われる
  - OOM Killer への恐怖
  - データベースファイルの内容は最新ではない
  - ファイルへの書き込みが不完全で再立ち上げできないかもしれない
- バックアップはどうする？

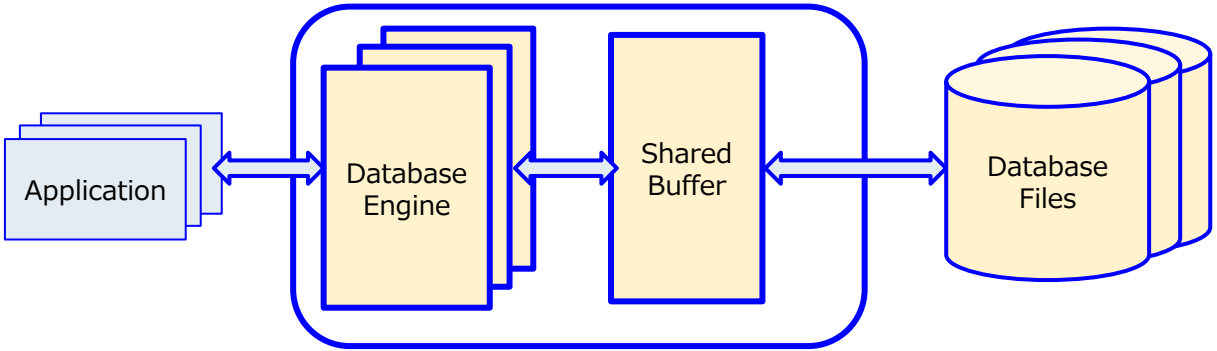
EDB

最初の改善

UndoログとRedoログ

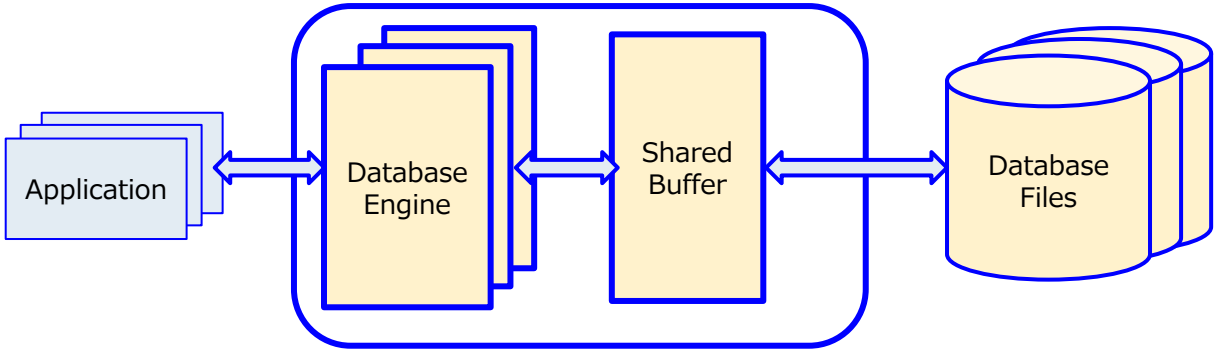
バージョン 8.0 以前のみちのり

とにかく記録しなきゃ



- トランザクションの巻き戻しのためのログを書く
  - Undo ログ
    - PostgreSQl では、これをデータベースファイル本体に残す方式
      - これを掃除するために vacuum が必要
- リカバリのためのログを書く
  - Redo　　ログ
    - 共有バッファへの変更の記録
    - PostgreSQLではこれを WAL あるいは XLOG と呼ぶ

なぜデータベースファイルに直接書かない？



- データベースファイルへの書き込みはランダムアクセス
  - シーケンシャルアクセスに比べて非常に遅い
- ディスクに記録するだけならシーケンシャルアクセスでいい
  - こちらは早いし、ある程度まとめて書ける

閑話休題 HDD/SDDの素の性能の例

- HDD_1: SATA インタフェース
- HDD_2: SATAインタフェース + Flashキャッシュ
- SSD: PCIe (INTEL750)
-
- シーケンシャルR/W: 32GB書き出し。ブロック 819200 バイト
- ランダムR/W: 500GBに8192バイト*1を10プロセスでそれぞれ1000回読み書き*2

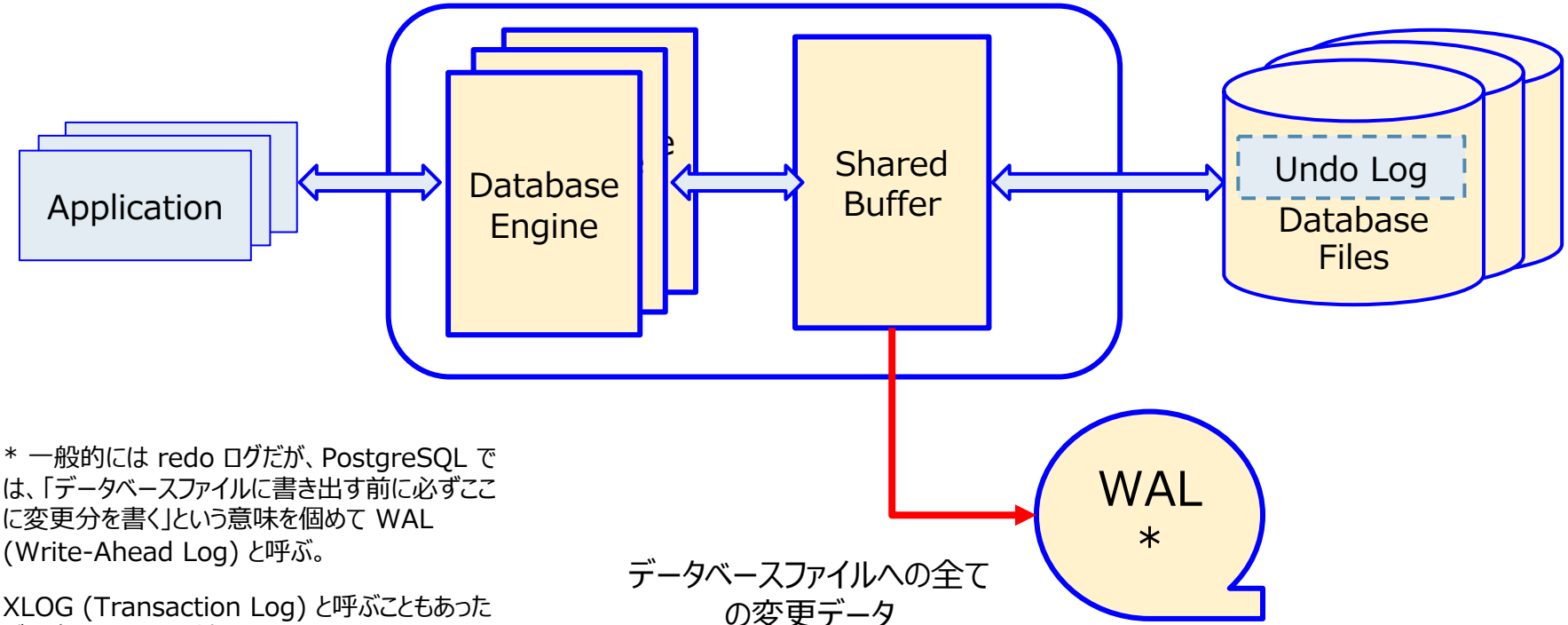
*1 DBMSでよく使われるブロックサイズ
*2 ファイルシステム経由の読み書きなので、その分の擾乱要因あり
*3 ここだけ10回書き出し毎に同期

Random/ Sequential	Read/ Write	Direct/Sync	Sync	Block (Byte)	HDD_1	HDD_2	SSD
Seq	Read	Direct	---	819200	182.0 MB/s	168.0 MB/s	1161.0 MB/s
Seq	Write	No Direct, No Sync	Each	8192	0.2 MB/s	0.1 MB/s	257.0 MB/s
Seq	Write	No direct	100	8192	119.0 MB/s	115.0 MB/s	900.0 MB/s*3
Random	Read	Direct	---	8192	0.8 MB/s	1.2 MB/s	100.0 MB/s
Random	Write	Direct	---	8192	0.3 MB/s	0.2 MB/s	218.0 MB/s
Random	Write	No Direct, No Sync	10	8192	1.6 MB/s	0.6 MB/s	488.0 MB/s
Random	Write	No Direct, No Sync	100	8192	1.6 MB/s	0.6 MB/s	490 MB/s

素のハードディスクはこれほど遅いので、実際にはインタフェースボードでキャッシュやブロッキングを行うことが多い。
→ 実際に書き込み完了しなくともCPUには完了通知が行く (write through)。その後の動作はインタフェースボードが保証する。

出典: Database, Tokyo City University lecture, by K.Suzuki

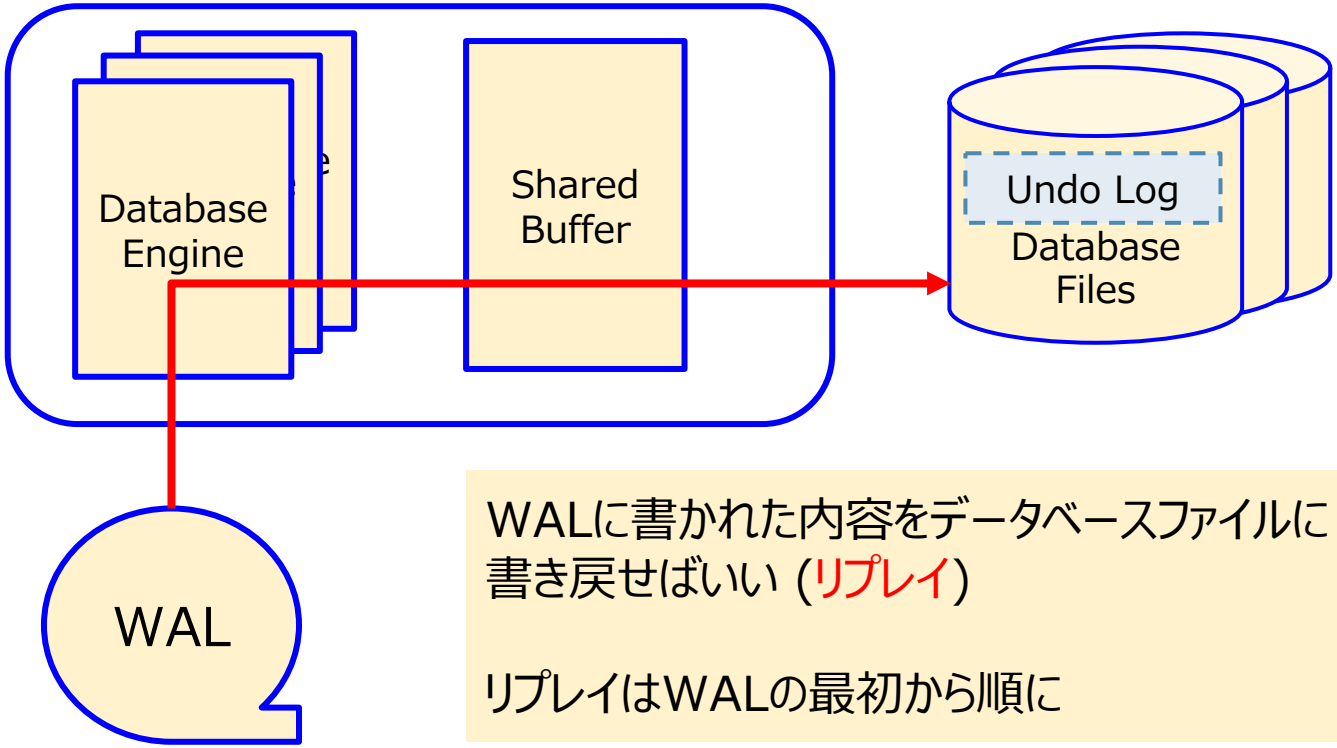
改善した仕掛け



* 一般的には redo ログだが、PostgreSQL では、「データベースファイルに書き出す前に必ずここに変更分を書く」という意味を込めて WAL (Write-Ahead Log) と呼ぶ。

XLOG (Transaction Log) と呼ぶこともあったが現在は WAL に統一されている

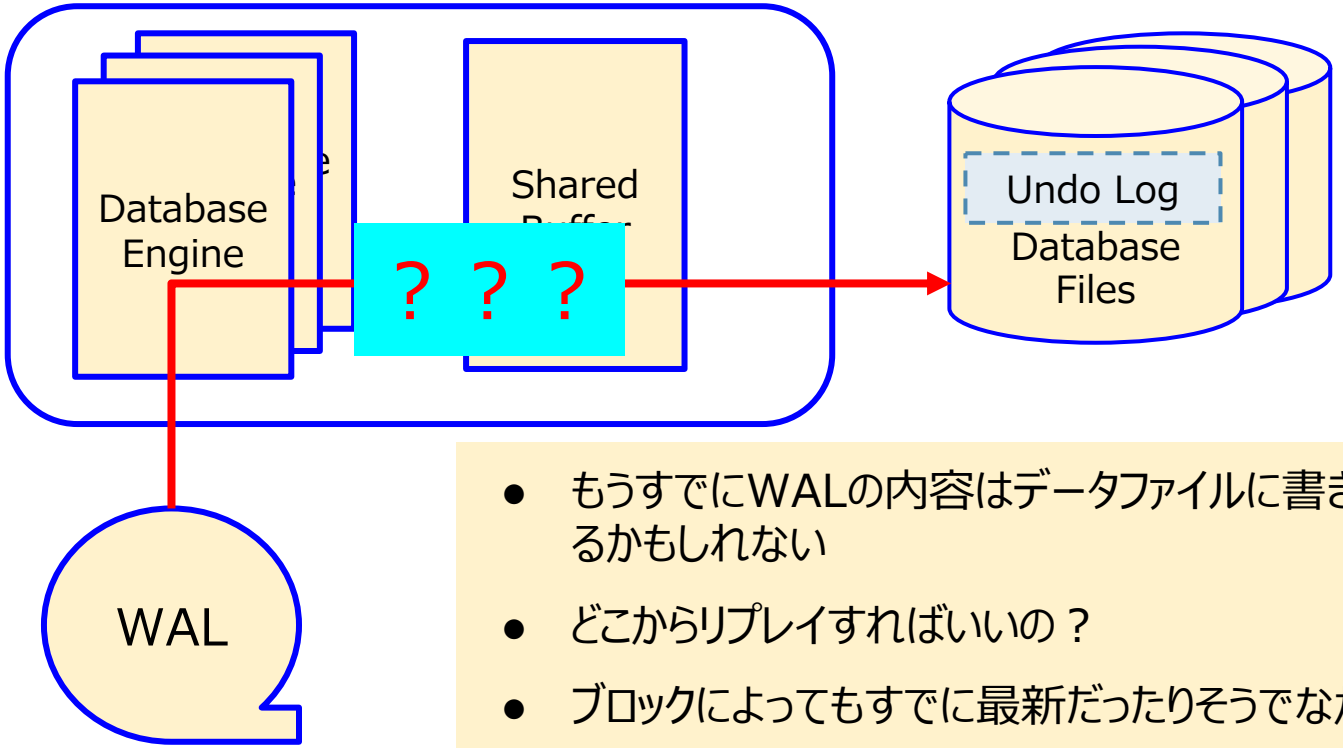
リカバリの方法



WALに書かれた内容をデータベースファイルに書き戻せばいい (リプレイ)

リプレイはWALの最初から順に

書き戻せと言ったって…



- もうすでにWALの内容はデータファイルに書き戻されているかもしれない
- どこからリプレイすればいいの？
- ブロックによってもすでに最新だったりそうでなかったりする

WAL のファイル構成

pg_wal ディレクトリ配下
サイズは 16MB 固定 (initdb で変更可能だが、やらない方が無難)
同期書き込み (具体的な方法は postgresql.conf で指定可能)
チェックポイント時に不要になった WAL は削除・再利用

標準的な PostgreSQL のディレクトリ構成

```
[koichi@ksubuntu:pg13database]$ ls
PG_VERSION        global/        pg_dynshmem/  pg_logical/    pg_replslot/  pg_stat/      pg_tblspc/   pg_xact/              postmaster.opts
base/             log/           pg_hba.conf   pg_multixact/  pg_serial/    pg_stat_tmp/  pg_twophase/ postgresql.auto.conf postmaster.pid
current_logfiles  pg_commit_ts/  pg_ident.conf  pg_notify/     pg_snapshots/ pg_subtrans/  pg_wal/      postgresql.conf
[koichi@ksubuntu:pg13database]$
```

このディレクトリ配下に WAL ファイルが配置されている。
バージョン10以前は pg_xlog

別のディレクトリに配置するように構成することも可能

WALが大きくなりそうなので独立したストレージに配置するなど

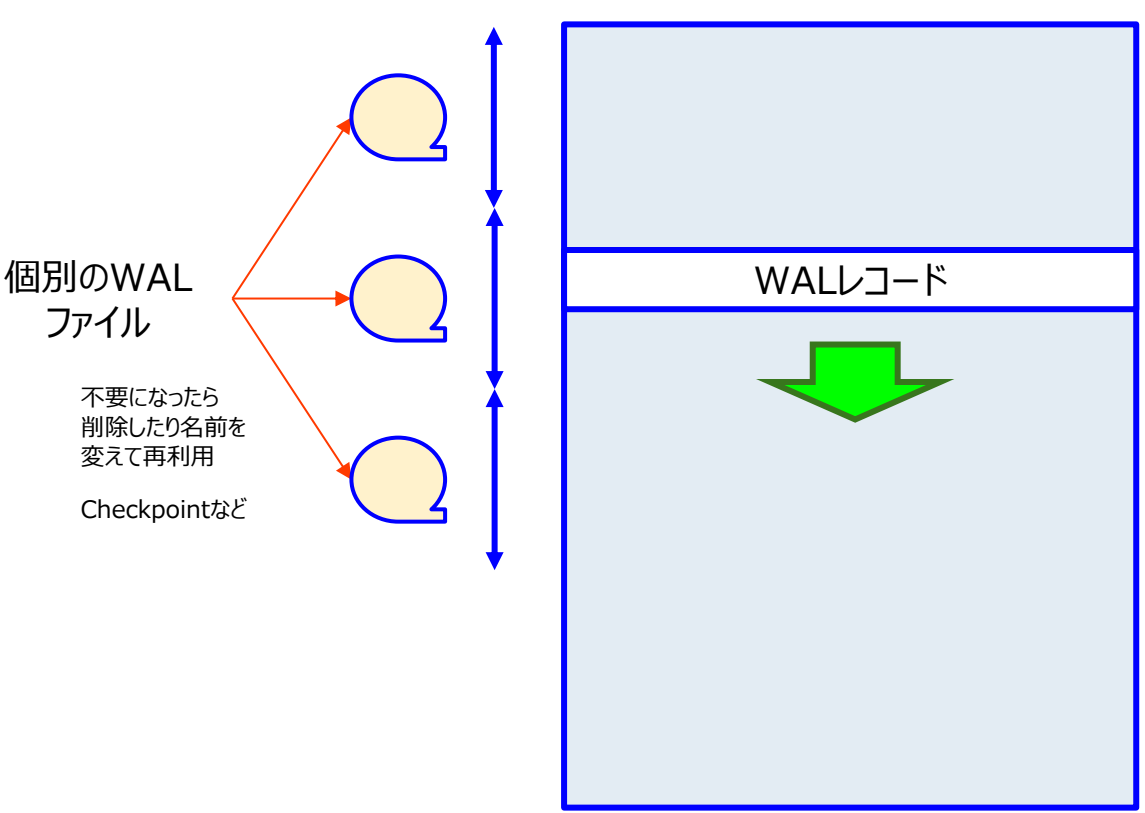
WAL のファイル構成

```
[koichi@ksubuntu:pg13database]$ ls pg_wal
000000010000000000000004E  000000010000000000000005F  000000010000000000000070  000000010000000000000081
000000010000000000000004F  000000010000000000000060  000000010000000000000071  000000010000000000000082
000000010000000000000050   000000010000000000000061  000000010000000000000072  000000010000000000000083
000000010000000000000051   000000010000000000000062  000000010000000000000073  000000010000000000000084
000000010000000000000052   000000010000000000000063  000000010000000000000074  000000010000000000000085
000000010000000000000053   000000010000000000000064  000000010000000000000075  000000010000000000000086
000000010000000000000054   000000010000000000000065  000000010000000000000076  000000010000000000000087
000000010000000000000055   000000010000000000000066  000000010000000000000077  000000010000000000000088
000000010000000000000056   000000010000000000000067  000000010000000000000078  000000010000000000000089
000000010000000000000057   000000010000000000000068  000000010000000000000079  00000001000000000000008A
000000010000000000000058   000000010000000000000069  00000001000000000000007A  00000001000000000000008B
000000010000000000000059   00000001000000000000006A  00000001000000000000007B  00000001000000000000008C
00000001000000000000005A   00000001000000000000006B  00000001000000000000007C  00000001000000000000008D
00000001000000000000005B   00000001000000000000006C  00000001000000000000007D  archive_status/
00000001000000000000005C   00000001000000000000006D  00000001000000000000007E
00000001000000000000005D   00000001000000000000006E  00000001000000000000007F
00000001000000000000005E   00000001000000000000006F  000000010000000000000080
[koichi@ksubuntu:pg13database]$
```

タイムラインID: リカバリする毎に1つづつ増える

LSN (Log Sequence Number) の上位 40ビット分

LSN (Log Sequence Number)



WALの先頭からの位置がLSN
64ビットの数値で表す

毎日 1TB ずつWALを書いても
5000年位は大丈夫

WAL には何が書かれているか

pg_waldump で調べることができる

```
[koichi@ksubuntu:pg13database]$ pg_waldump 000000010000000000000004E
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E000678, prev 0/4DFFE948, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27010 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E002398, prev 0/4E000678, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27011 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E0040B8, prev 0/4E002398, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27012 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E005DC0, prev 0/4E0040B8, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27013 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E007AE0, prev 0/4E005DC0, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27014 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E009800, prev 0/4E007AE0, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27015 FPW
rmgr: XLOG        len (rec/tot):     49/  7429, tx:        522, lsn: 0/4E00B520, prev 0/4E009800, desc: FPI , blkref #0: rel 1663/16384/32789 blk 27016 FPW

... (snip) ...

rmgr: Btree       len (rec/tot):     53/   153, tx:        534, lsn: 0/4E3247E0, prev 0/4E324548, desc: INSERT_LEAF off 3, blkref #0: rel 1663/16384/3080 blk 1 FPW
rmgr: Btree       len (rec/tot):     53/   161, tx:        534, lsn: 0/4E324880, prev 0/4E3247E0, desc: INSERT_LEAF off 1, blkref #0: rel 1663/16384/3081 blk 1 FPW
rmgr: Heap        len (rec/tot):     54/  5178, tx:        534, lsn: 0/4E324928, prev 0/4E324880, desc: INSERT off 85 flags 0x00, blkref #0: rel 1663/16384/2608 blk 58 FPW
rmgr: Btree       len (rec/tot):     53/  5853, tx:        534, lsn: 0/4E325D68, prev 0/4E324928, desc: INSERT_LEAF off 176, blkref #0: rel 1663/16384/2673 blk 32 FPW
rmgr: Btree       len (rec/tot):     53/  4397, tx:        534, lsn: 0/4E327460, prev 0/4E325D68, desc: INSERT_LEAF off 73, blkref #0: rel 1663/16384/2674 blk 37 FPW
rmgr: Heap        len (rec/tot):     54/  4026, tx:        534, lsn: 0/4E3285A8, prev 0/4E327460, desc: INSERT off 49 flags 0x00, blkref #0: rel 1663/16384/2609 blk 35 FPW
rmgr: Btree       len (rec/tot):     53/  6337, tx:        534, lsn: 0/4E329568, prev 0/4E3285A8, desc: INSERT_LEAF off 223, blkref #0: rel 1663/16384/2675 blk 14 FPW
rmgr: Heap        len (rec/tot):     54/  8010, tx:        534, lsn: 0/4E32AE48, prev 0/4E329568, desc: INSERT off 18 flags 0x01, blkref #0: rel 1663/16384/1255 blk 70 FPW
rmgr: Btree       len (rec/tot):     53/  2853, tx:        534, lsn: 0/4E32CDB0, prev 0/4E32AE48, desc: INSERT_LEAF off 138, blkref #0: rel 1663/16384/2690 blk 10 FPW
rmgr: Btree       len (rec/tot):     53/  4393, tx:        534, lsn: 0/4E32D8D8, prev 0/4E32CDB0, desc: INSERT_LEAF off 39, blkref #0: rel 1663/16384/2691 blk 8 FPW
rmgr: Heap        len (rec/tot):     80/    80, tx:        534, lsn: 0/4E32EA20, prev 0/4E32D8D8, desc: INSERT off 86 flags 0x00, blkref #0: rel 1663/16384/2608 blk 58
rmgr: Btree       len (rec/tot):     53/  6637, tx:        534, lsn: 0/4E32EA70, prev 0/4E32EA20, desc: INSERT_LEAF off 83, blkref #0: rel 1663/16384/2673 blk 29 FPW
rmgr: Btree       len (rec/tot):     72/    72, tx:        534, lsn: 0/4E330478, prev 0/4E32EA70, desc: INSERT_LEAF off 74, blkref #0: rel 1663/16384/2674 blk 37
rmgr: Heap        len (rec/tot):     80/    80, tx:        534, lsn: 0/4E3304C0, prev 0/4E330478, desc: INSERT off 87 flags 0x00, blkref #0: rel 1663/16384/2608 blk 58

... (snip) ...

[koichi@ksubuntu:pg13database]$
```

データベースへのファイルレベルでの更新情報が入っている。LSN のチェイン情報も入っている

データベースにどのWALをプレイバックするか

pg_controldata で調べることができる

```
[koichi@ksubuntu:~]$ pg_controldata -D pg13database/
pg_control version number:            1300
Catalog version number:               202007201
Database system identifier:           6918906660938445044
Database cluster state:               in production
pg_control last modified:             2021年03月25日 16時08分22秒
Latest checkpoint location:           0/4E34FB88
Latest checkpoint's REDO location:    0/4E34FB50
Latest checkpoint's REDO WAL file:    000000010000000000000004E
Latest checkpoint's TimeLineID:       1
Latest checkpoint's PrevTimeLineID:   1
Latest checkpoint's full_page_writes: on

... (snip) ...

WAL block size:                       8192
Bytes per WAL segment:                16777216

... (snip) ...

[koichi@ksubuntu:~]$
```

最後のチェックポイント位置

チェックポイント直前の更新

最後のチェックポイントデータが入っている WALファイル名

テーブルのページ上の WAL の情報

Pageinspect extension で調べることができる

```
[koichi@ksubuntu:~]$ psql
Null display is "[NULL]".
\timing on
Timing is on.
psql (13.2)
Type "help" for help.

koichi=# create extension pageinspect;
CREATE EXTENSION
Time: 7.827 ms
koichi=# select * from page_header(get_raw_page('pgbench_accounts', 0));
-[ RECORD 1 ]--------
lsn       | 0/16908E0
checksum  | 0
flags     | 4
lower     | 268
upper     | 384
special   | 8192
pagesize  | 8192
version   | 4
prune_xid | 0

Time: 0.357 ms
koichi=#

koichi=# \q
[koichi@ksubuntu:~]$
```

このブロックを更新した WAL の LSN
これより後ろの LSN の WAL が見つかったらプレイバックする

WALを保持しておけば

データベースのクラッシュリカバリができる

WALのどこまでをプレイバックするかを決めれば

Point-in-time Recovery (PITR) ができる

EDB

ここで
オンラインバックアップ
バージョン 9.0 以前のみちのり

WALとデータファイルを対でバックアップ

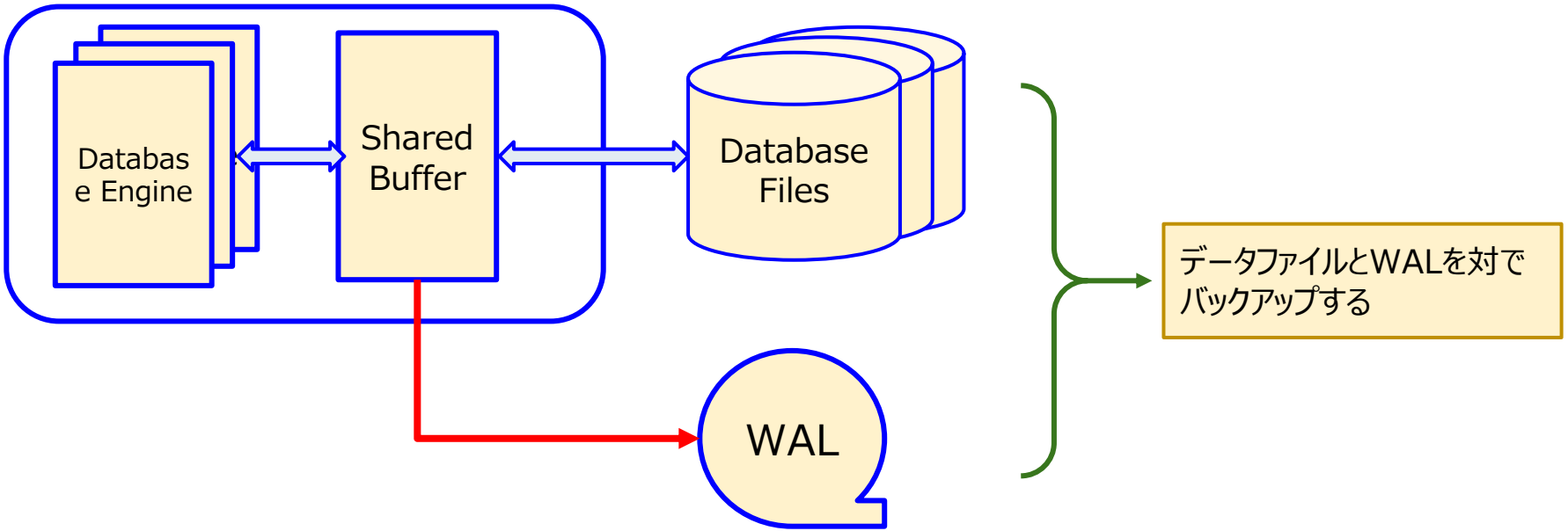
WALとデータファイルを対でバックアップすればオンラインバックアップができるんじゃないか？

データファイルも常に書き換えられているから、バックアップ取っている間は WAL にデータページを全部書こう (FPW --- Full Page Write。通常は差分のみ) -- `pg_start_backup()`

バックアップが終わったら WAL には差分だけを書くように戻そう
-- `pg_stop_backup()`

チェックポイントの後などでも FPW が書かれることがある

オンラインバックアップ



だんだん欲が出てきた

バックアップ取った後の WAL も全て保存しておけば、いつクラッシュしても直前の状態まで戻せるはず

WAL がいっぱいになったらこれを任意の場所にバックアップできるようにしよう --- archive_command

バックアップにWALをプレイバックし続ければリカバリの時間短縮もできる --- warm standby

EDB

レプリケーション登場

バージョン 9.0
及びそれ以降

レプリケーションへの発展

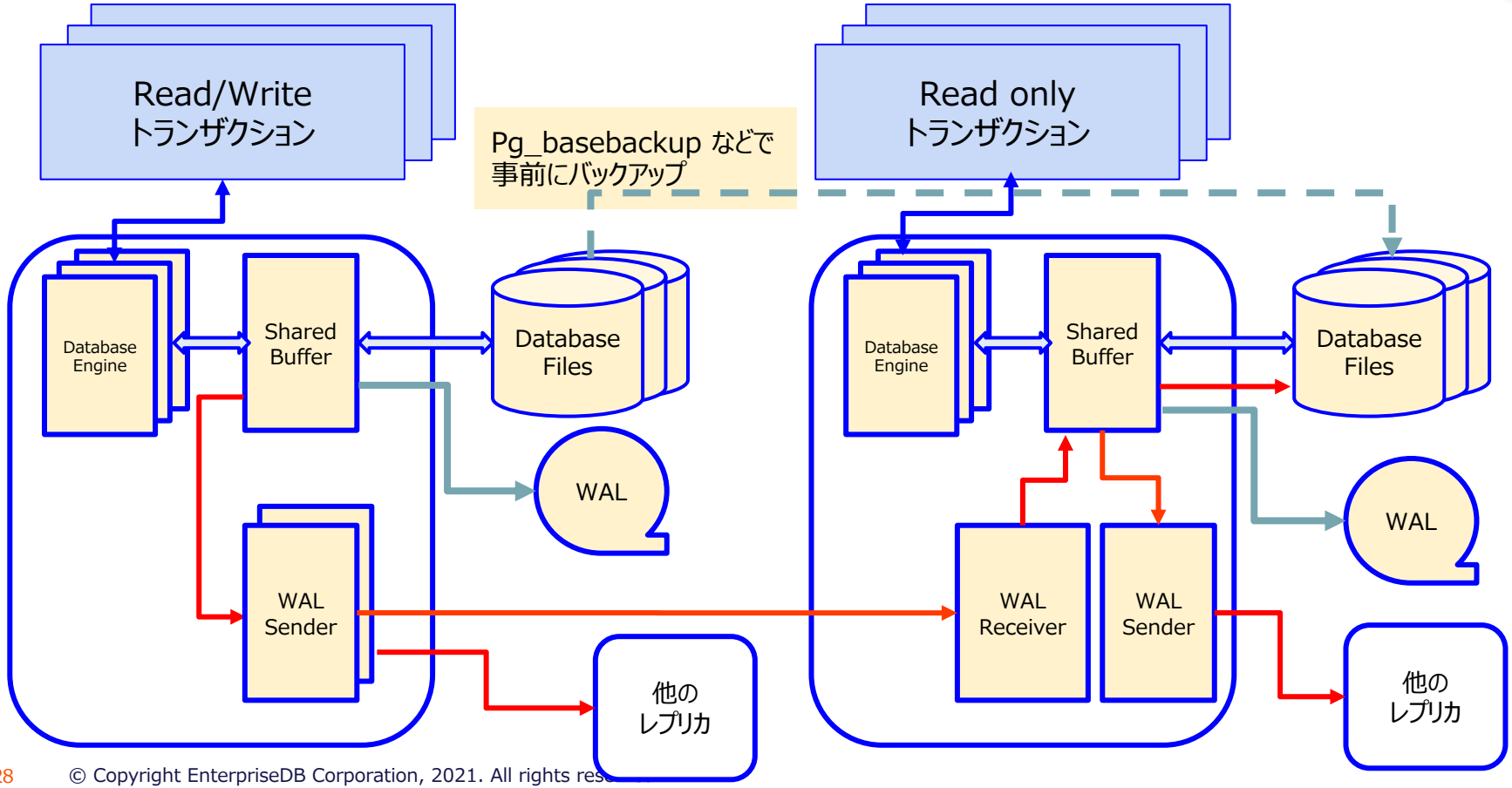
WAL を書くときにリモートに同時に送ってしまってそこでプレイバックを続けたら

元のデータベースと同じ状態を常時作っておける
WALが壊れても安全 (WAL アーカイブだと最後のWALがプレイバックできない可能性がある)

クラッシュ時のフェイルオーバが短時間に可能になる

読み込みのみのトランザクションならここで実行できる (スケールアウト)

ストリーミ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の特徴

長所：

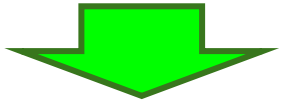
- ほとんどの更新がレプリケーションできる (マスター側のアプリケーションへの制約がない)
- 例外は WAL を書かないオブジェクト
    - Unlogged Table, 一時テーブル、一部の外部定義のインデックスなど
- レプリカで読み込みトランザクションを動かせる
- WAL を転送するだけなので比較的軽い
- 高速なフェイルオーバー

短所：

- レプリカ側では書き込みトランザクションは実行できない
- レプリカは元のデータベースの「完全な」コピーになる

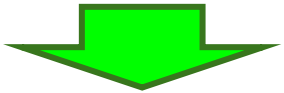
ストリーミングレプリケーションの種々の改良

- レプリカでの読み込みトランザクションを実行させるために、WAL プレイバックの遅延が必要な場合がある
  - Vacuum で古いオブジェクトを無条件に掃除できない
- レプリカの状態に応じて、プライマリでも WAL を残す必要がある



Replication Slot

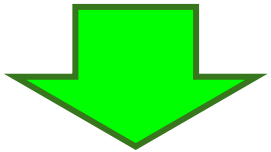
- レプリカをプライマリに昇格したあと、元のプライマリがうまく使えないことがある
  - 元のプライマリのオーバーシュートなど



Pg_rewind ユーティリティ

ストリーミングレプリケーションの限界

- レプリカは元のデータベースの「完全な」コピーになる
- 異なるデータベースにデータの更新を送る目的には使えない
  - Slony など、この目的のためのレプリケーションを置き換えるものではない
  - これらの異なるデータベース間でのデータ更新ではトリガが使われていたため、性能上の問題が発生することもあった



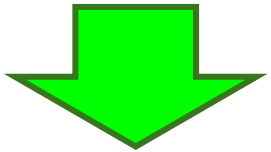
WALを使って何とか異なるデータベース間での柔軟なレプリケーションはできないものか

EDB

論理レプリケーション

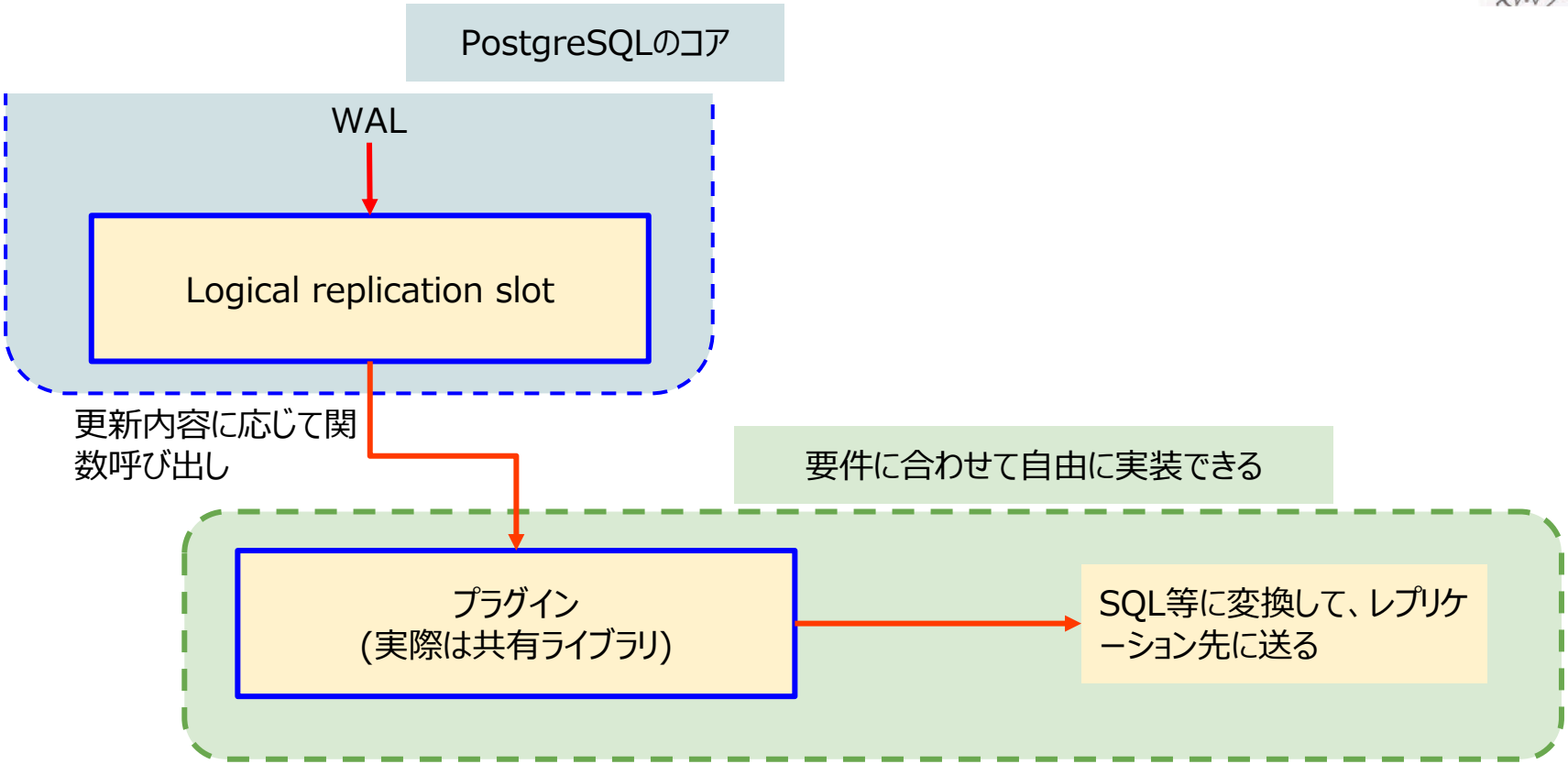
論理レプリケーションのアイデア

- WALに対応するSQL文を生成して、これを使ってプレイバックする
  - トリガに比べて必要なリソースが少ない
  - 本来のトランザクションとは独立に動作できる
- 別にSQL文を生成せずとも、他の形式でもいい
  - Rabbitmq にデータを供給することもできる
- アプリケーションにより種々の方法が取れるようにしたい

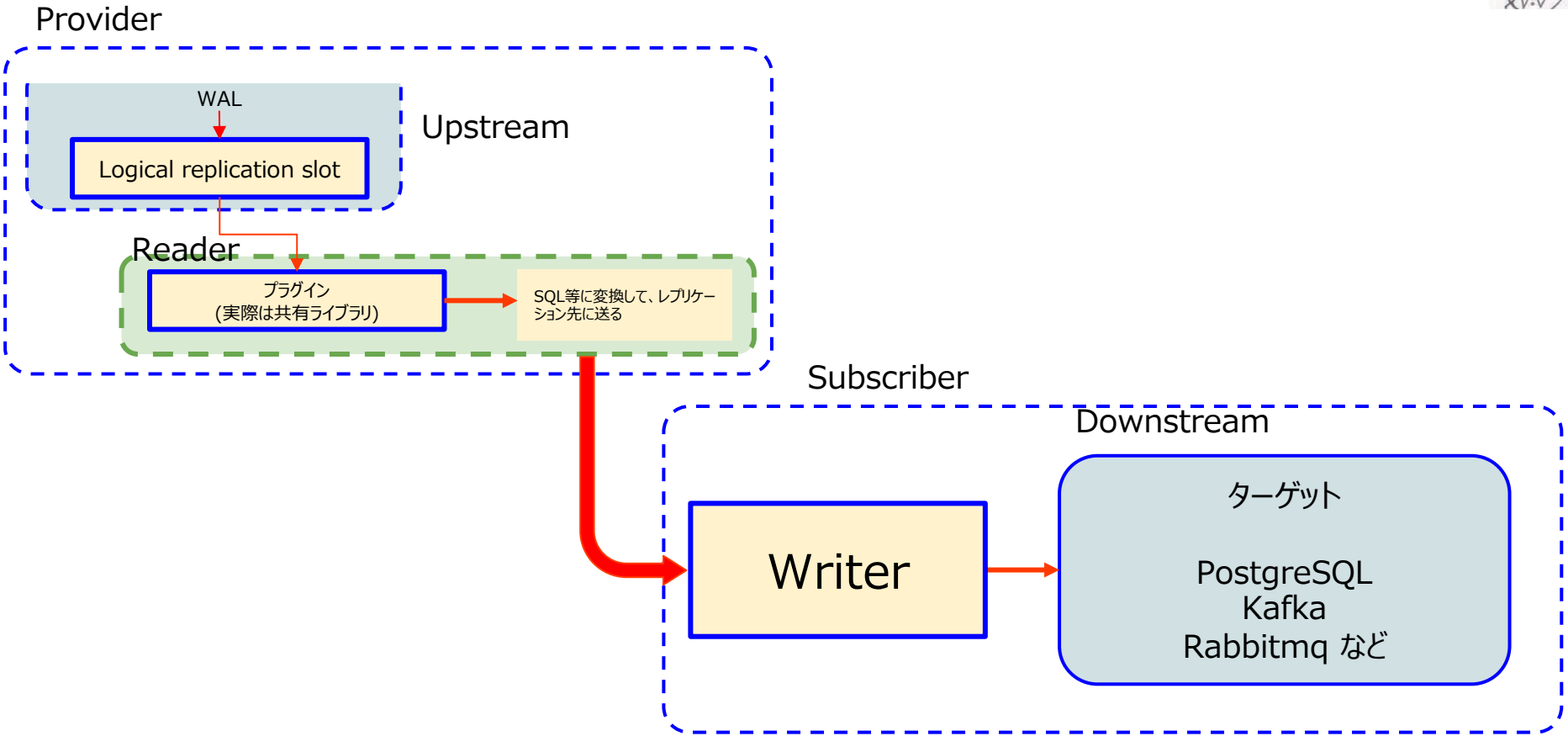


WAL sender をうまく使ってアプリケーションを組み込む方法を取ることに

論理レプリケーションの仕掛け



Provider, Subscriber, Reader, Writer



プラグインの実装例

Test_decoding

PostgreSQL付属のもの

Decoder_raw

WALをSQL文に変換するもの

https://github.com/michaelpq/pg_plugins/tree/master/decoder_raw

新たに Reader を書く際のよい指針

Wal2json

WALをJSONに変換するもの

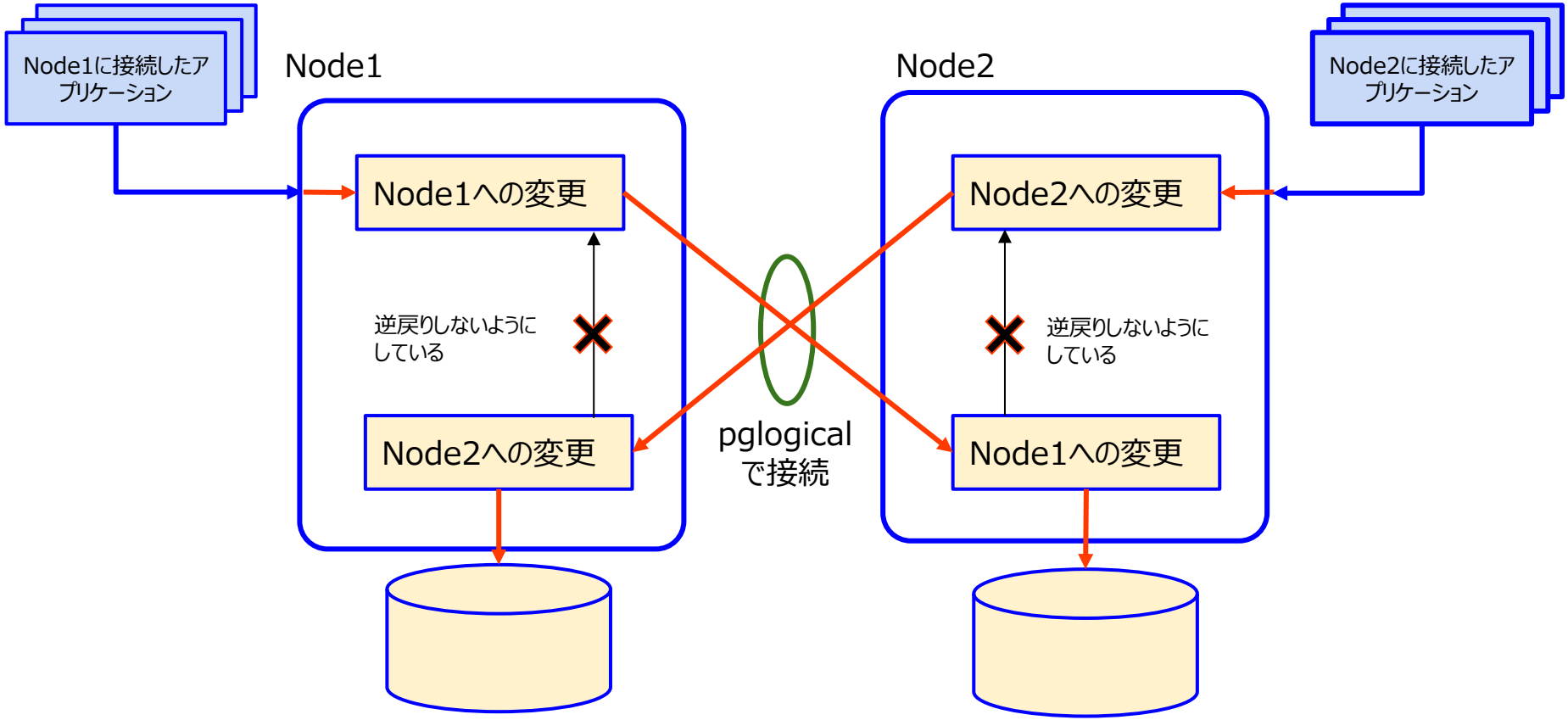
https://github.com/eulerto/wal2json.git

Pglogical

https://www.2ndquadrant.com/en/resources/pglogical/

これが BDR の基盤になっている

Pglogicalを双方向で使うとBDR ができる



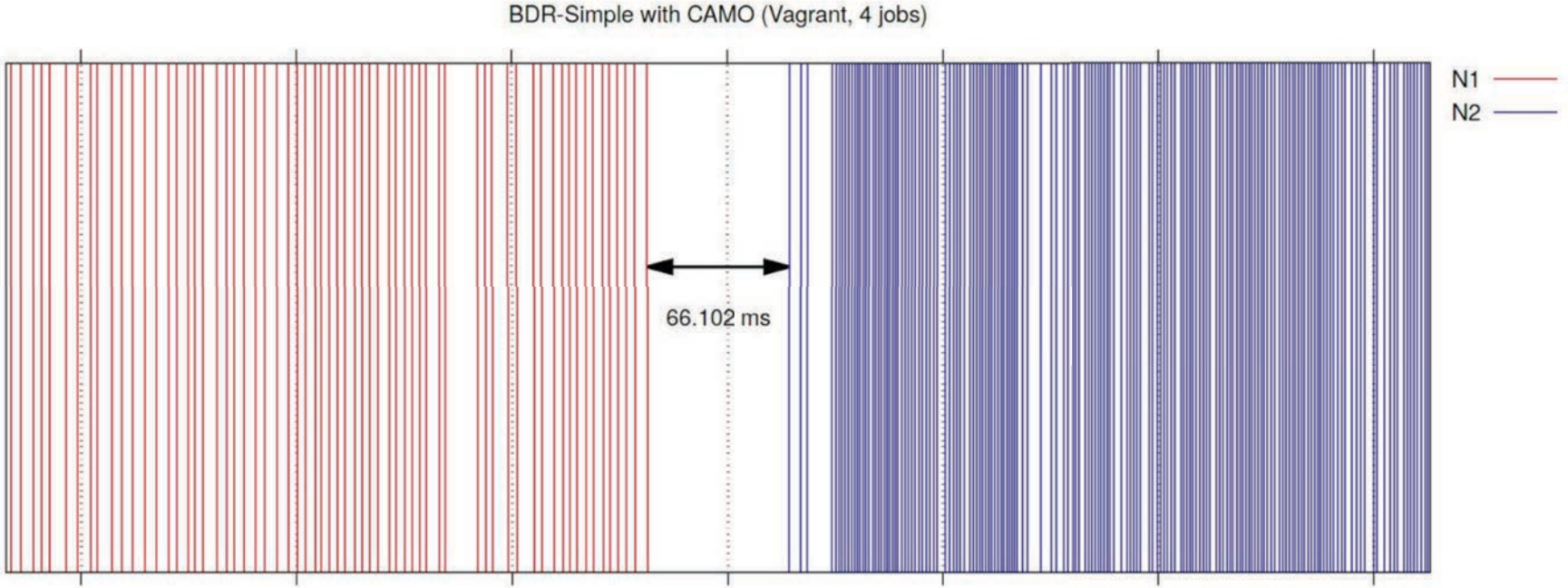
追加の機能及び特徴

- DDLの伝搬
  - 特定のノードで行ったテーブルの生成変更などを他のノードにも伝搬
- コンフリクト対策
  - ローカルな変更と矛盾する変更をレプリケートする際に、どうするかを設定可能
- グローバルなシーケンス
  - ノードの間でシーケンスの値が重複しないようにコントロール

- データベースに障害が生じたら、アプリケーションは生きているサーバに接続しなおすだけ。
  - この際フェイルオーバは不要
  - 超高速な切替が可能

障害時の高速切替

- Failover possible in <100ms
- Much improved over 30-90s required for single master replication



その他の情報

- EDB としてのアナウンス、展開は現在準備中
- 旧2ndQuadrant の情報：
  - https://www.2ndquadrant.com/en/resources/pglogical/
  - https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/

EDB

ご清聴ありがとう
ございました

koichi.suzuki@enterprisedb.com