PostgreSQL 17と PostgreSQL開発最前線

PostgreSQL Major Contributor & Committer

Senior Software Development Engineer

澤田　雅彦

Index

- PostgreSQL 17
- PostgreSQL 開発最前線 (v18 and beyond)

本発表について

- 本発表に掲載している検証結果は、以下の環境で取得したものでdす
- 環境や条件などによっては異なる結果となる可能性があります
- AWS EC2 m6ld.metal, RHEL 8.6, 128 vCPUs, 512GB RAM, SSD

PostgreSQL 17リリース！ (9/26)

2681個
のコミット

242個
の新機能

463人
の開発者

PostgreSQL 12がEOL

Version	Current minor	Supported	First Release	Final Release
17	17.2	Yes	September 26, 2024	November 8, 2029
16	16.6	Yes	September 14, 2023	November 9, 2028
15	15.10	Yes	October 13, 2022	November 11, 2027
14	14.15	Yes	September 30, 2021	November 12, 2026
13	13.18	Yes	September 24, 2020	November 13, 2025
12	12.22	No	October 3, 2019	November 21, 2024

PostgreSQL 17 主な新機能

- Vacuumの改善（性能向上、省メモリ）
- SQL/JSON標準実装の強化
- MERGEコマンドへの機能追加
- 高可用性、メジャーバージョンアップグレードのための論理レプリケーションの改善
- 増分バックアップをサポート
- COPYの性能向上、機能追加
- プラットフォーム非依存のCollation（照合順序）

などなど

Vacuumの改善

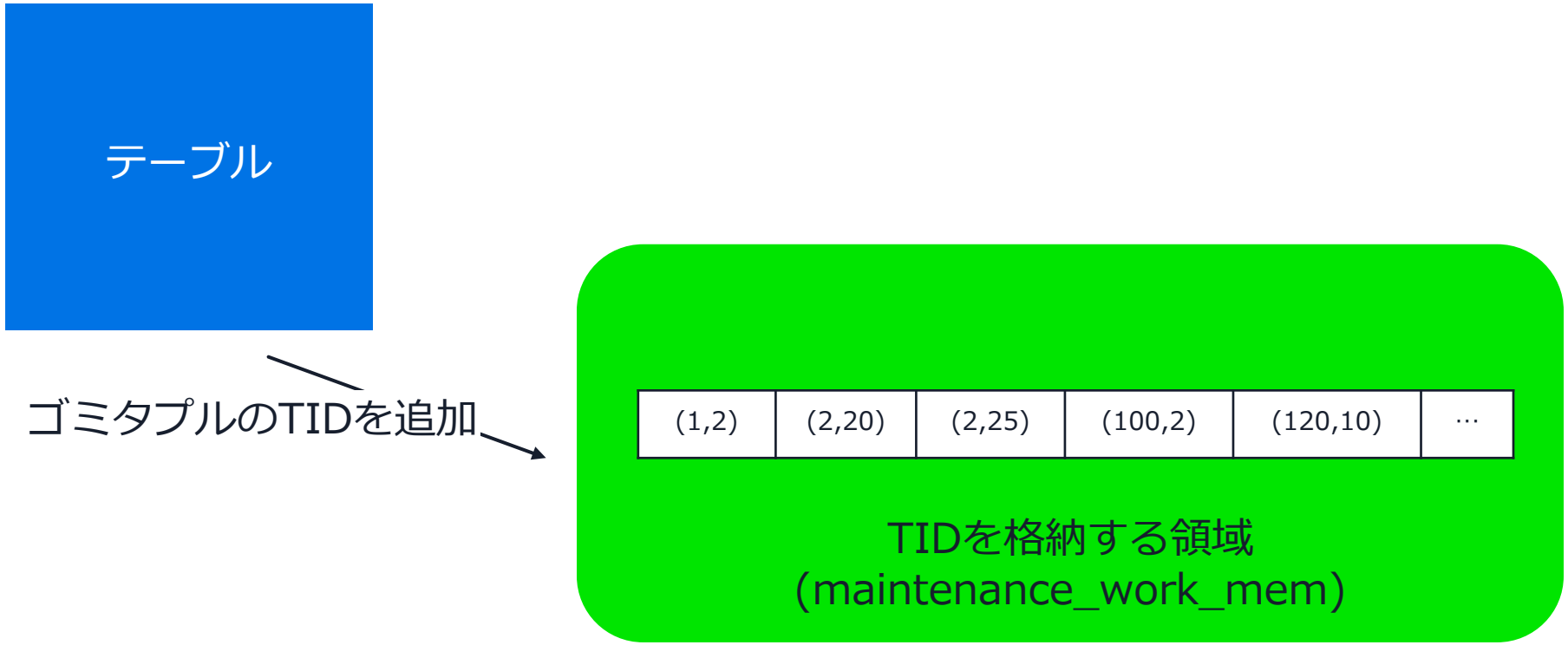
- PostgreSQL 17ではVacuumが大幅アップデート
  - メモリ使用量が最大で1/20
  - 2~3倍の性能向上
  - maintenance_work_memの1GB制限を排除
  - WAL出力量の削減
- なぜこんなに早くなった？

Vacuumのしくみ

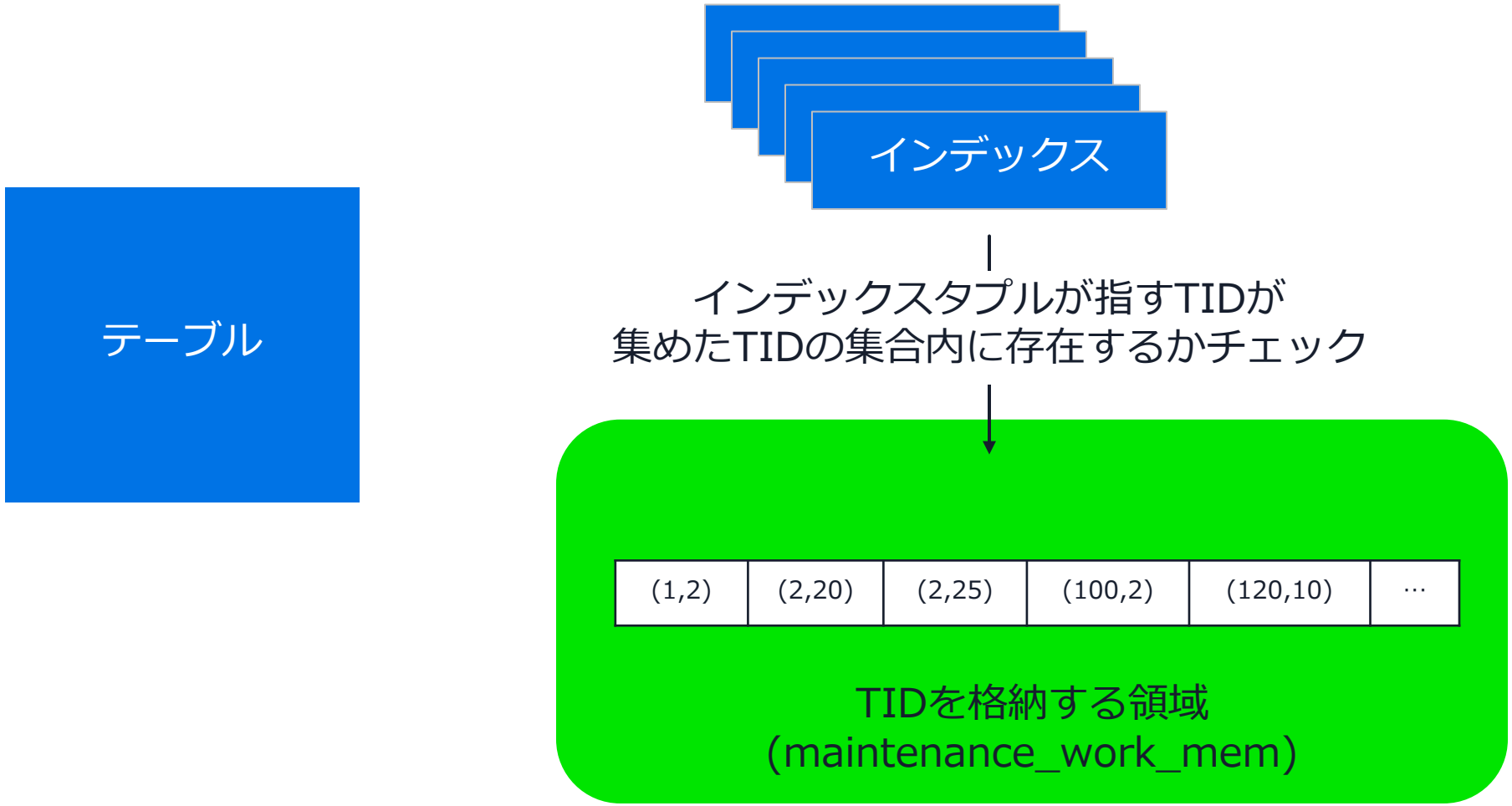
テーブルを
先頭からスキャン

TIDを格納する領域
(maintenance_work_mem)

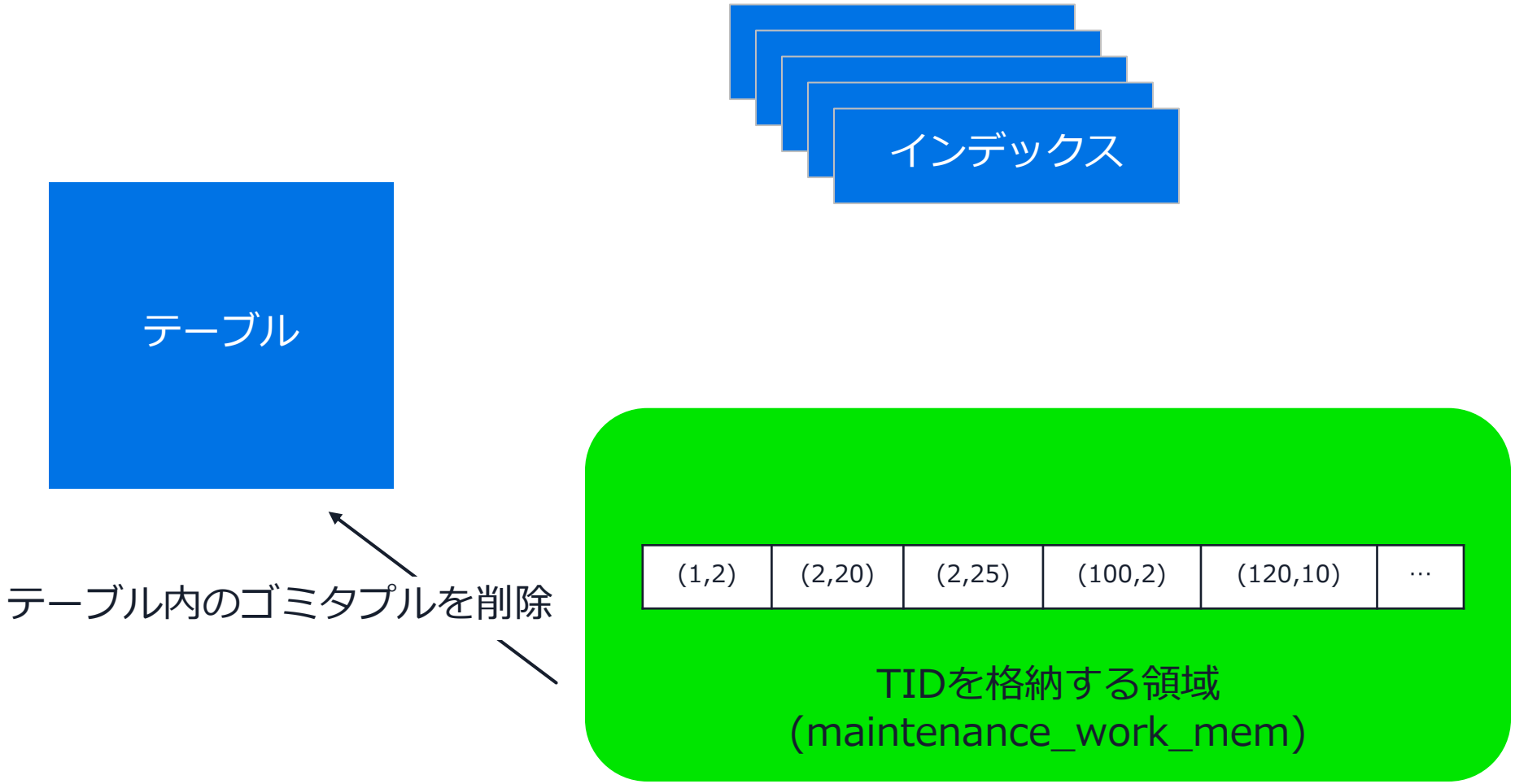
Vacuumのしくみ



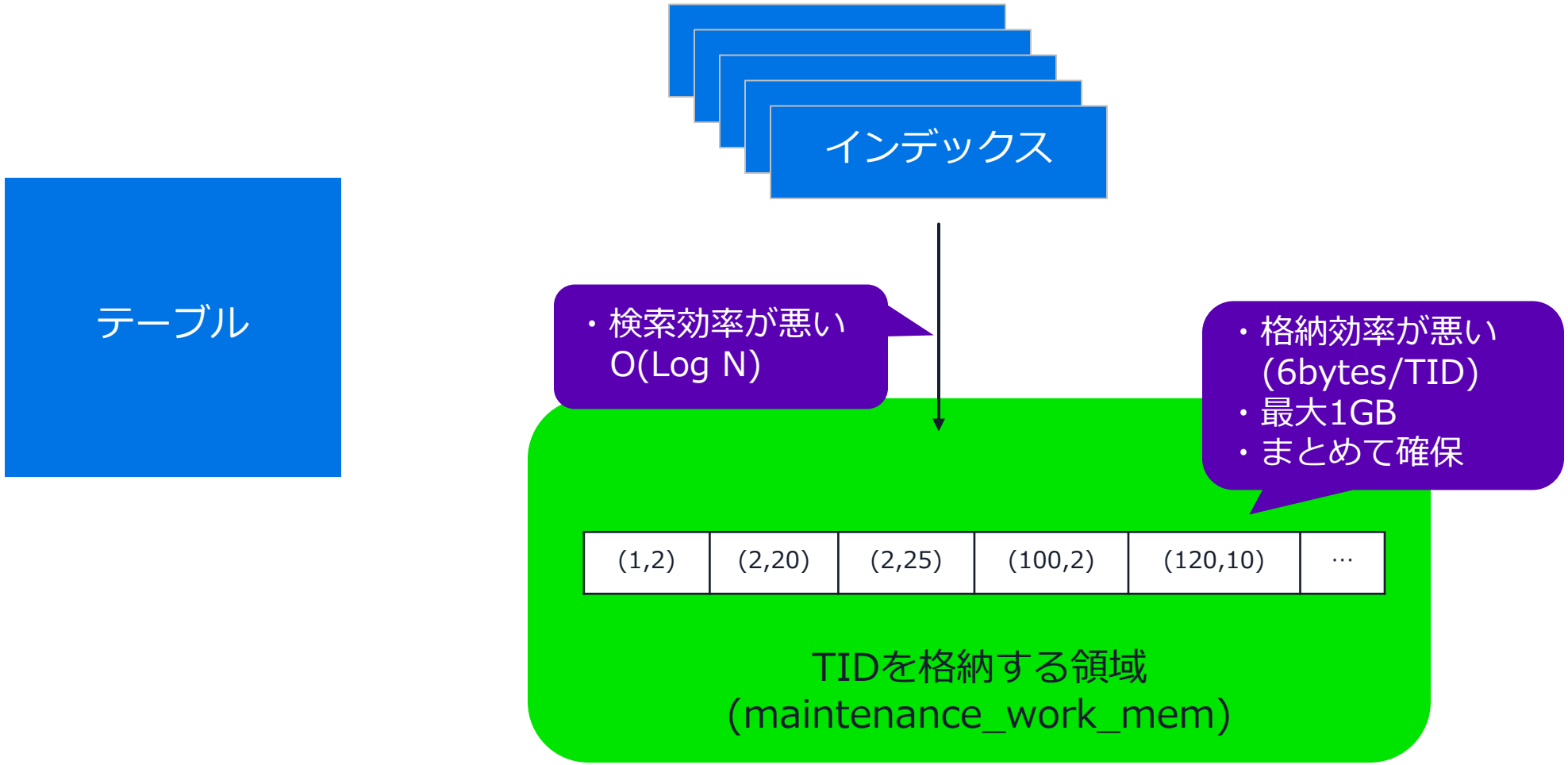
Vacuumのしくみ



Vacuumのしくみ

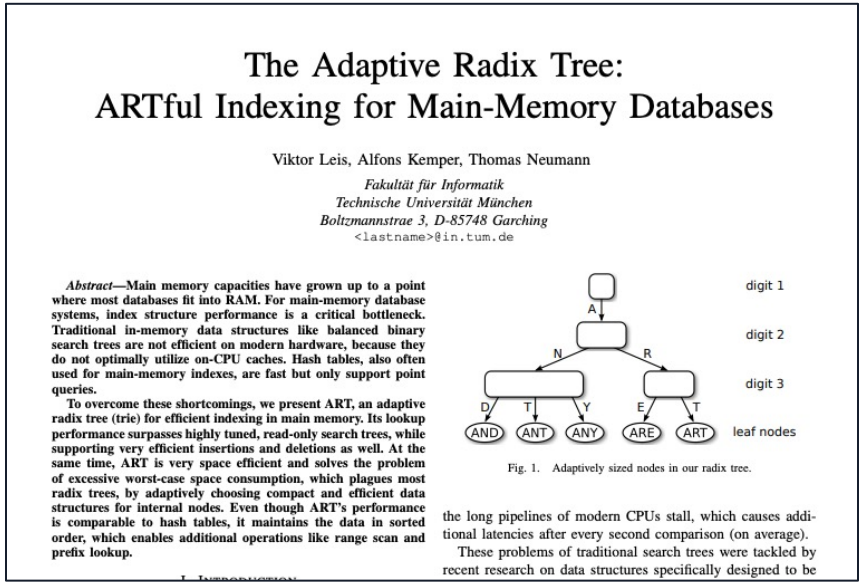


Vacuumのしくみ



新しいデータ構造を追加

- Adaptive Radix Tree(ART)を新しく実装
  - その上にTidStoreという新しいデータ構造を作成
  - キーがブロック番号、バリューがオフセット番号のビットマップ
  - 例) オフセットを100個格納してもサイズは28bytes (vs. 600 bytes)
    ※実際にはオフセットの分布によります
- 検索は常にO(k)。k=4 (ブロック番号のバイト数)
- メモリはARTの成長に合わせて段階的に確保



The Adaptive Radix Tree:
ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann

Fakultät für Informatik
Technische Universität München
Boltzmannstrae 3, D-85748 Garching
<lastname>@in.tum.de

***Abstract*—Main memory capacities have grown up to a point where most databases fit into RAM. For main-memory database systems, index structure performance is a critical bottleneck. Traditional in-memory data structures like balanced binary search trees are not efficient on modern hardware, because they do not optimally utilize on-CPU caches. Hash tables, also often used for main-memory indexes, are fast but only support point queries.**

To overcome these shortcomings, we present ART, an adaptive radix tree (trie) for efficient indexing in main memory. Its lookup performance surpasses highly tuned, read-only search trees, while supporting very efficient insertions and deletions as well. At the same time, ART is very space efficient and solves the problem of excessive worst-case space consumption, which plagues most radix trees, by adaptively choosing compact and efficient data structures for internal nodes. Even though ART's performance is comparable to hash tables, it maintains the data in sorted order, which enables additional operations like range scan and prefix lookup.

Fig. 1. Adaptively sized nodes in our radix tree.

the long pipelines of modern CPUs stall, which causes additional latencies after every second comparison (on average).

These problems of traditional search trees were tackled by recent research on data structures specifically designed to be

https://db.in.tum.de/~leis/papers/ART.pdf

実行時間の違い(~v16)

```
=# create index idx1 on test (id);
CREATE INDEX

=# delete from test where id % 10 = 0;
DELETE 10000000

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 636943 remain, 636943 scanned (100.00% of total)
tuples: 10000000 removed, 90000000 remain, 0 are dead but not yet removable
removable cutoff: 745, which was 0 XIDs old when operation ended
new relfrozenxid: 731, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 636943 pages from table (100.00% of total) had 10000000 dead item identifiers removed
index "idx1": pages: 274194 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 95.265 MB/s, avg write rate: 94.991 MB/s
buffer usage: 1911087 hits, 274193 misses, 273404 dirtied
WAL usage: 2184215 records, 179 full page images, 178320700 bytes
system usage: CPU: user: 13.24 s, system: 1.18 s, elapsed: 22.48 s
VACUUM
```

int型の列にインデックスを作成

1000万行をDELETE

22.48秒で完了

実行時間の違い(v17)

```
=# create index idx1 on test (id);
CREATE INDEX

=# delete from test where id % 10 = 0;
DELETE 10000000

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test"
INFO:  finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 636943 remain, 636943 scanned (100.00% of total)
tuples: 10000000 removed, 90000000 remain, 0 are dead but not yet removable
removable cutoff: 753, which was 0 XIDs old when operation ended
new relfrozenxid: 750, which is 12 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 636943 pages from table (100.00% of total) had 10000000 dead item identifiers removed
index "idx1": pages: 274194 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 209.752 MB/s, avg write rate: 217.129 MB/s
buffer usage: 1911088 hits, 274193 misses, 283836 dirtied
WAL usage: 2184215 records, 179 full page images, 178957643 bytes
system usage: CPU: user: 8.38 s, system: 0.84 s, elapsed: 10.21 s
VACUUM
```

v16: 22.48秒
v17: 10.21秒

VACUUMの実行速度自体が向上
(メモリ使用量: 60MB vs. 25MB)

メモリ使用量の違い①(~v16)

```
=# delete from test where ((ctid::text::point)[0])::int % 2 = 0;
DELETE 200000056

=# set maintenance_work_mem to '3GB';
SET

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test2"
INFO:  finished vacuuming "postgres.public.test2": index scans: 2
pages: 0 removed, 1769912 remain, 1769912 scanned (100.00% of total)
tuples: 199997344 removed, 200002656 remain, 0 are dead but not yet re
tuples missed: 2712 dead from 12 pages not removed due to cleanup lock
removable cutoff: 747, which was 0 XIDs old when operation ended
new relfrozenxid: 732, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 884944 pages from table (50.00% of total) had 199997344 dead item identifiers removed
index "idx1": pages: 1096762 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 817.252 MB/s, avg write rate: 530.109 MB/s
buffer usage: 2752140 hits, 6060334 misses, 3931025 dirtied
WAL usage: 3 records, 0 full page images, 613 bytes
system usage: CPU: user: 47.61 s, system: 9.03 s, elapsed: 57.93 s
```

偶数番のページにあるタプルをDELETE

内部的に1GBに制限される

maintenance_work_memが足りず、
インデックスVacuumを２回実施

メモリ使用量の違い①(v17)

```
=# delete from test where ((ctid::text::point)[0])::int % 2 = 0;
DELETE 200000056

=# set maintenance_work_mem to '3GB';
SET

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test2"
INFO:  finished vacuuming "postgres.public.test2": index scans: 1
pages: 0 removed, 1769912 remain, 1769912 scanned (100.00% of total)
tuples: 199995988 removed, 200004012 remain, 0 are dead but not yet removable
tuples missed: 4068 dead from 18 pages not removed due to cleanup lock contention
removable cutoff: 755, which was 0 XIDs old when operation ended
new relfrozenxid: 740, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 884938 pages from table (50.00% of total) had 199995988 dead item identifiers removed
index "idx1": pages: 1096762 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 820.899 MB/s, avg write rate: 828.955 MB/s
buffer usage: 2751753 hits, 3867189 misses, 3905143 dirtied
WAL usage: 3 records, 2 full page images, 6819 bytes
system usage: CPU: user: 28.53 s, system: 7.02 s, elapsed: 36.80 s
```

1GBの制限はない

v16: 57.93秒 (index scans: 2)
v17: 36.80秒 (index scans: 1)

1GBの制限がなくなったため、インデックスVacuumの回数が少ない
(メモリ使用量: 1.1GB vs. 62MB)

メモリ使用量の違い②(~v16)

```
=# delete from test2 where (ctid::text::point)[1] > 220;
DELETE 7964598

=# set maintenance_work_mem to '64MB';
SET

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test2"
INFO:  finished vacuuming "postgres.public.test2": index scans: 1
pages: 0 removed, 1769912 remain, 1769912 scanned (100.00% of total)
tuples: 10619220 removed, 389380780 remain, 0 are dead but not yet removable
tuples missed: 246 dead from 41 pages not removed due to cleanup lock contention
removable cutoff: 746, which was 0 XIDs old when operation ended
new relfrozenxid: 732, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 1769870 pages from table (100.00% of total) had 10619220 dead item identifiers removed
index "idx2": pages: 1096762 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 541.233 MB/s, avg write rate: 542.360 MB/s
buffer usage: 3079680 hits, 3327416 misses, 3334341 dirtied
WAL usage: 2 records, 0 full page images, 425 bytes
system usage: CPU: user: 40.37 s, system: 7.48 s, elapsed: 48.03 s
VACUUM
```

各ページの後ろにある行のみを削除

インデックスVacuumは1回で済む

メモリ使用量の違い②(v17)

```
=# delete from test2 where (ctid::text::point)[1] > 220;
DELETE 7964598

=# set maintenance_work_mem to '64MB';
SET

=# vacuum (verbose) test;
INFO:  vacuuming "postgres.public.test2"
INFO:  finished vacuuming "postgres.public.test2": index scans: 2
pages: 0 removed, 1769912 remain, 1769912 scanned (100.00% of total)
tuples: 10618752 removed, 389381248 remain, 0 are dead but not yet removable
tuples missed: 714 dead from 119 pages not removed due to cleanup lock contention
removable cutoff: 754, which was 0 XIDs old when operation ended
new relfrozenxid: 740, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 1769792 pages from table (99.99% of total) had 10618752 dead item identifiers removed
index "idx2": pages: 1096762 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 716.248 MB/s, avg write rate: 540.848 MB/s
buffer usage: 3079364 hits, 4424421 misses, 3340939 dirtied
WAL usage: 2 records, 2 full page images, 6406 bytes
system usage: CPU: user: 35.91 s, system: 12.24 s, elapsed: 48.25 s
VACUUM
```

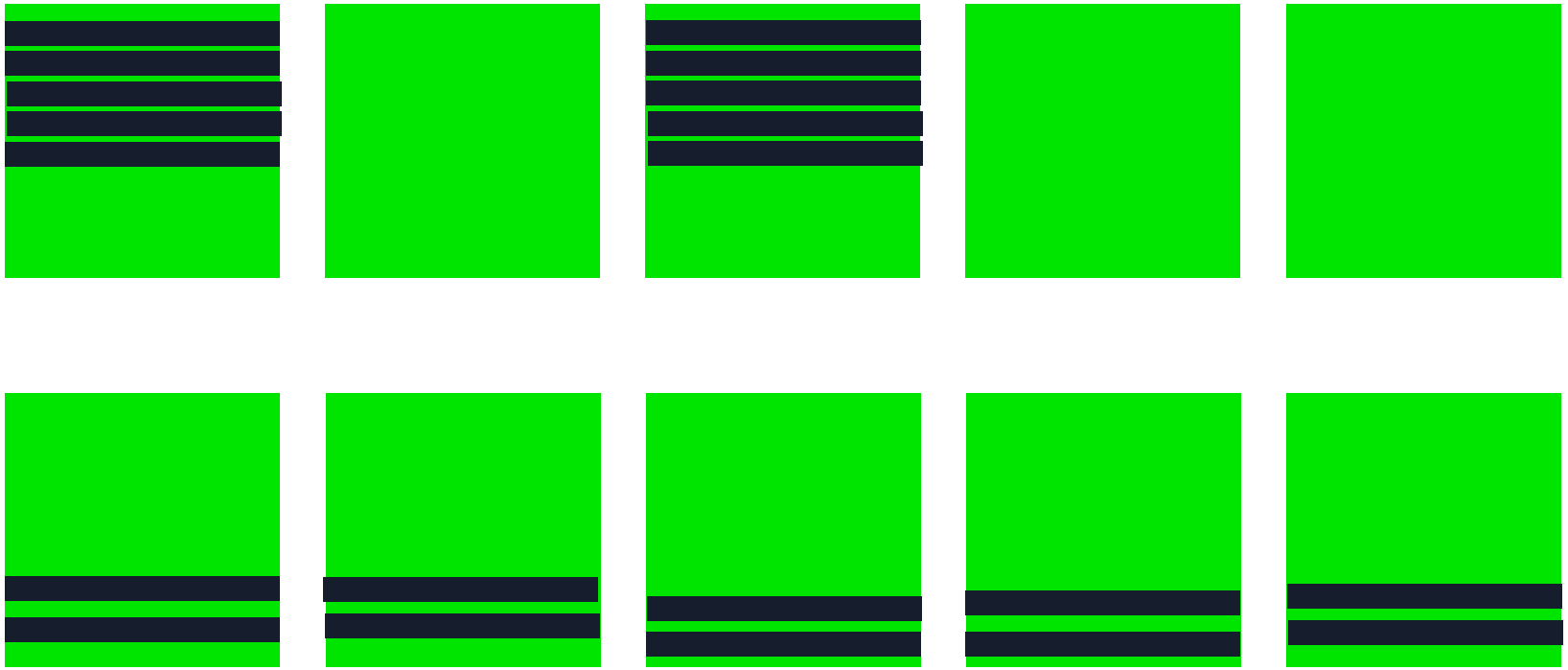
v16よりもメモリを使っている

v16: 48.03秒 (index scans: 1)
v17: 48.25秒 (index scans: 2)

不要タプルの分布によってはより多くのメモリを使うことも
(メモリ使用量: 48MB vs. 70MB)

メモリの使い方の違い

- ゴミタプルの分布によってはメモリを多く使うこともあり得る



v16: 6 * 10 = 60 bytes
v17: (4+8) * 2 = 24 bytes

v16: 6*10 = 60 bytes
v17: (4+16) * 5 = 100 bytes

Vacuum改善のまとめ

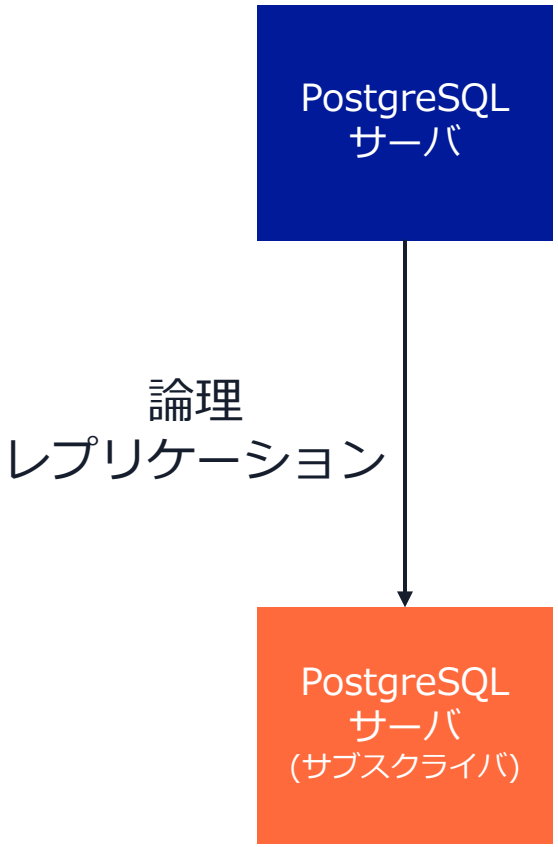
- メモリ使用効率が悪い
  ⇛ メモリ使用量は最大1/20に
- 検索が遅い O(log N)
  ⇛ O(k)で検索可能
- maintenance_work_memが1GBまでしか使えない
  ⇛ 1GB制限廃止
- メモリは一度にまとめて確保
  ⇛ 段階的に確保

省メモリ・効率的に動くようになったためインデックスVacuumの回数が減った。
Vacuum単体の性能も向上。

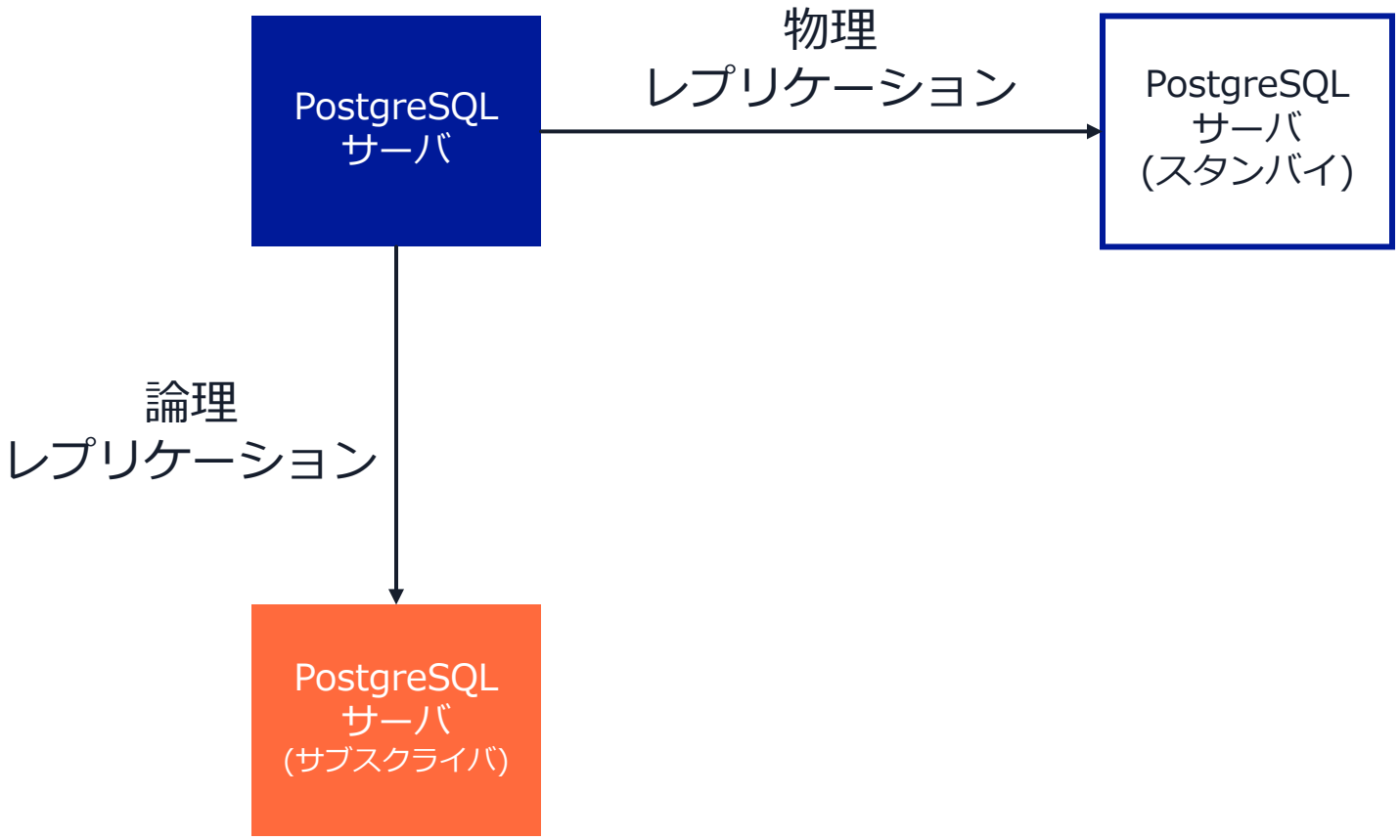
論理レプリケーションの新機能も多数！

- **ファイルオーバー後もシームレスに論理レプリケーションが継続可能**
- **pg_upgradeと論理レプリケーションを組み合わせたアップグレードが容易に**
- pg_createsubscriberの追加

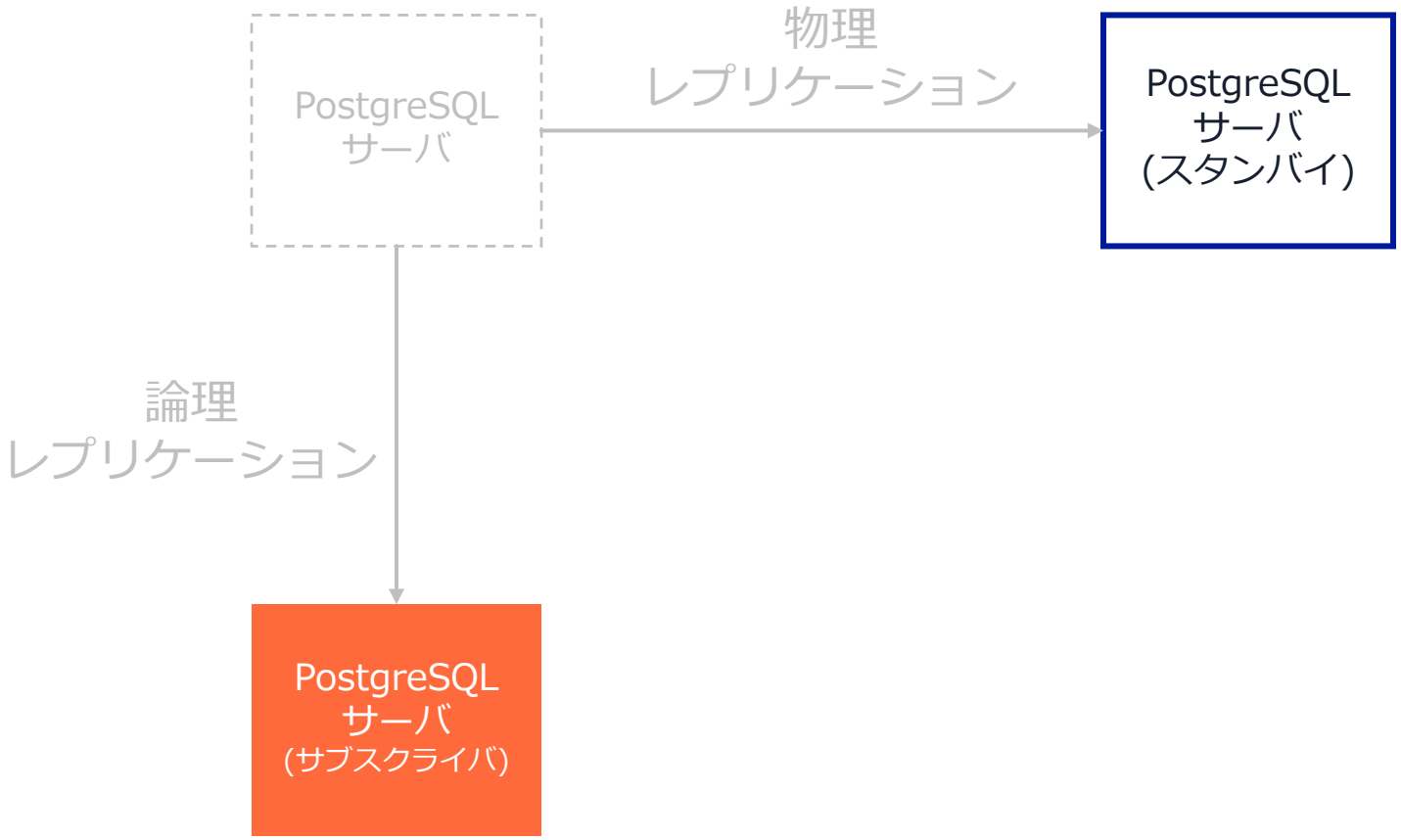
フェイルオーバー対応



フェイルオーバー対応



フェイルオーバー対応



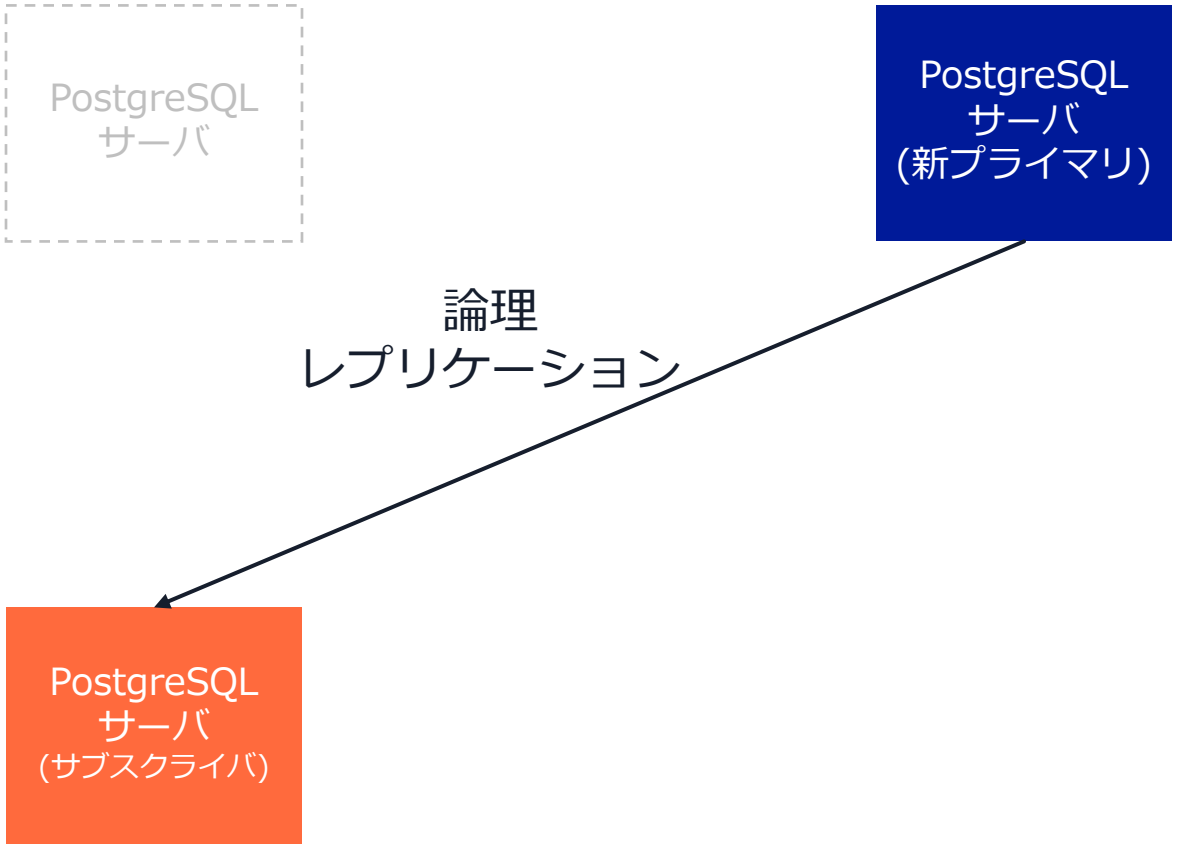
フェイルオーバー対応

PostgreSQL
サーバ

PostgreSQL
サーバ
(新プライマリ)

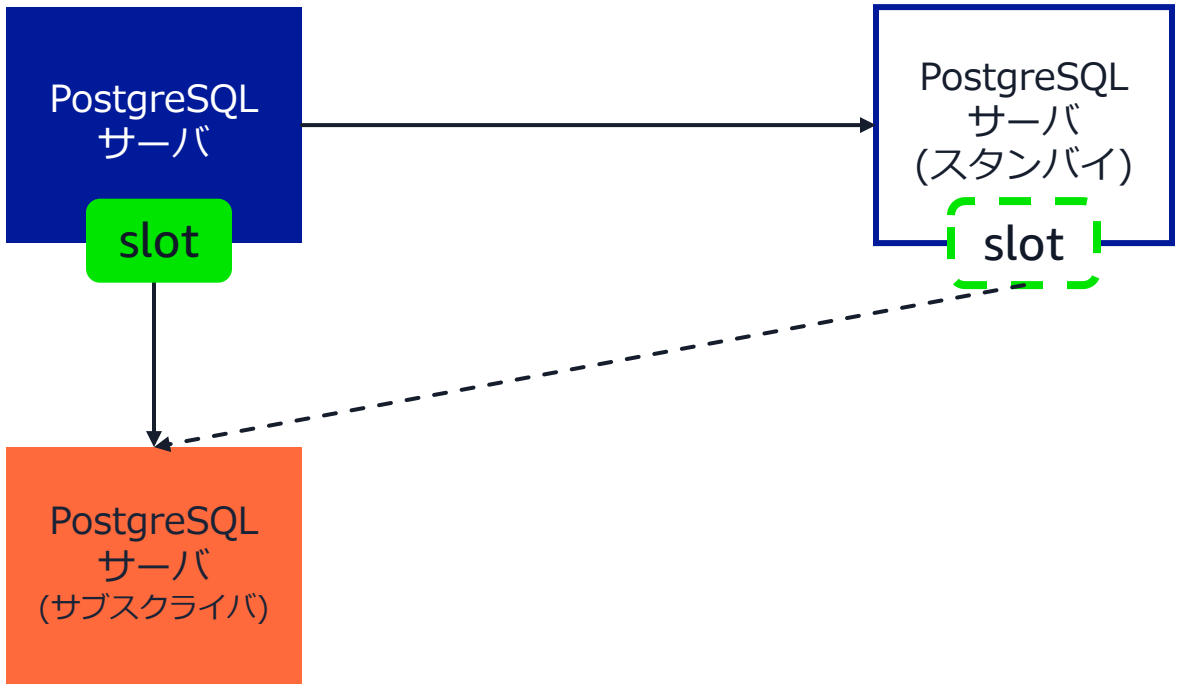
PostgreSQL
サーバ
(サブスクライバ)

フェイルオーバー対応



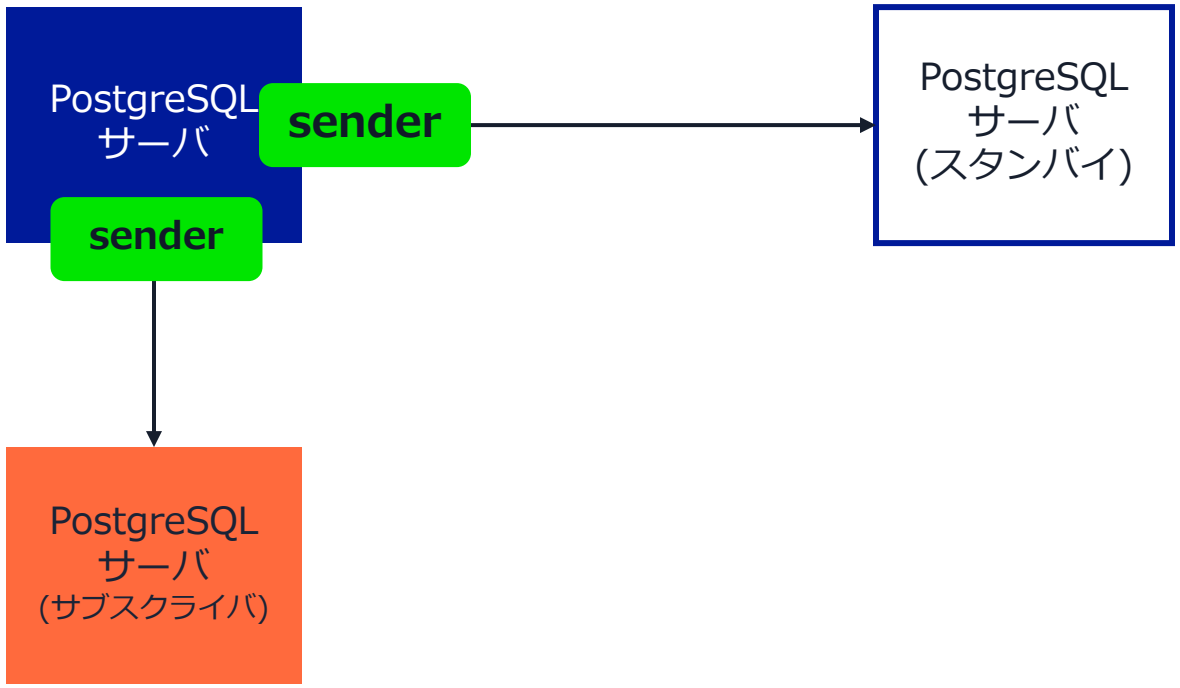
フェイルオーバーするときの問題点①(v16まで)

- 論理レプリケーションにはレプリケーションスロットが必要
  - どこまでWALを送信したか、などの情報が入っている
- プライマリサーバ上のスロット情報（作成、更新、削除）はスタンバイに伝搬されない
  - スタンバイでは新しいスロットが作られない。手動では作れるがWALは溜まりっぱなしになる

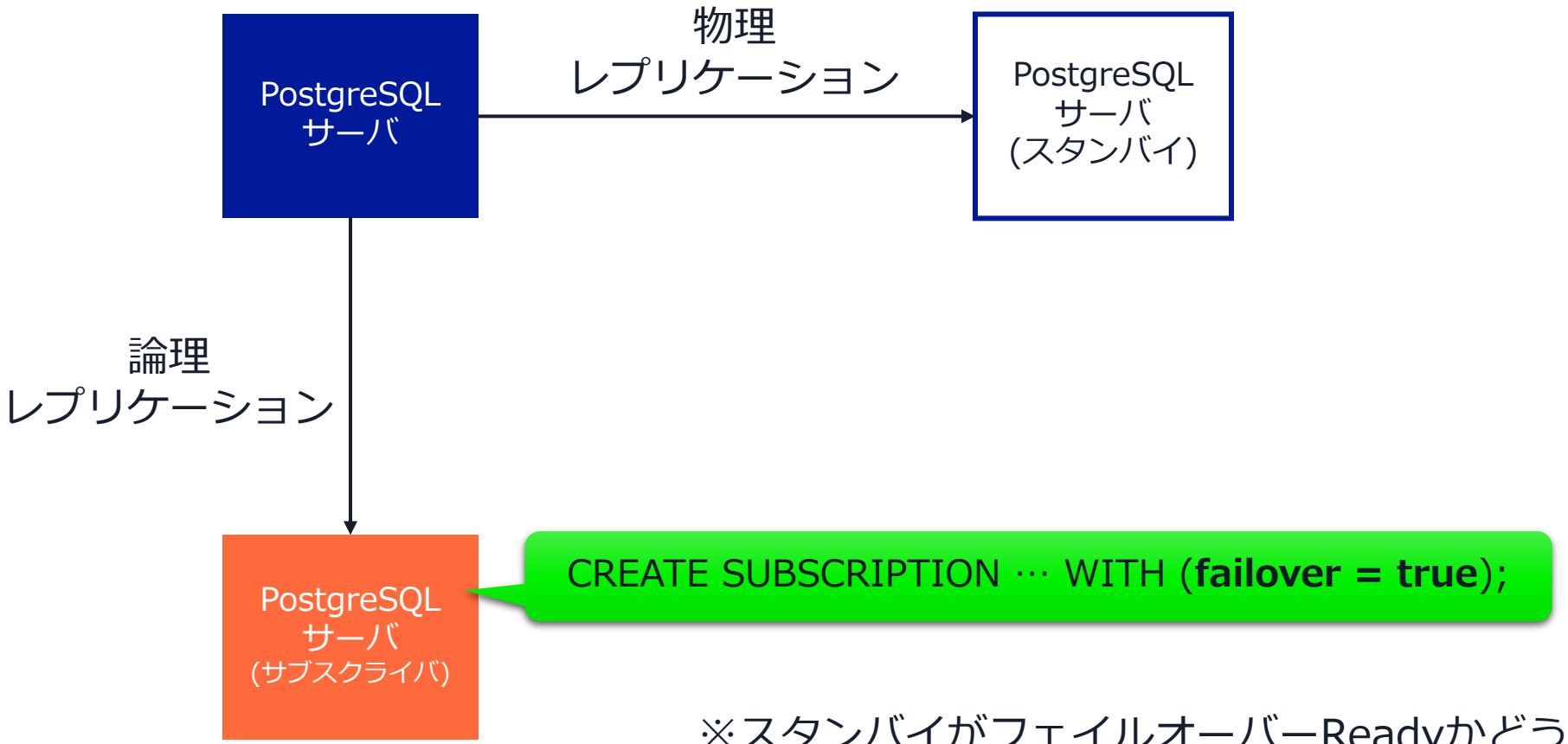


フェイルオーバーするときの問題点②(v16まで)

- サブスクライバがスタンバイより”先”に進む可能性がある
  - ※各walsenderプロセスの送信順序は不定
- フェイルオーバー後、サブスクライバがほしい”続きの情報”がスタンバイにない、という状況になってしまう

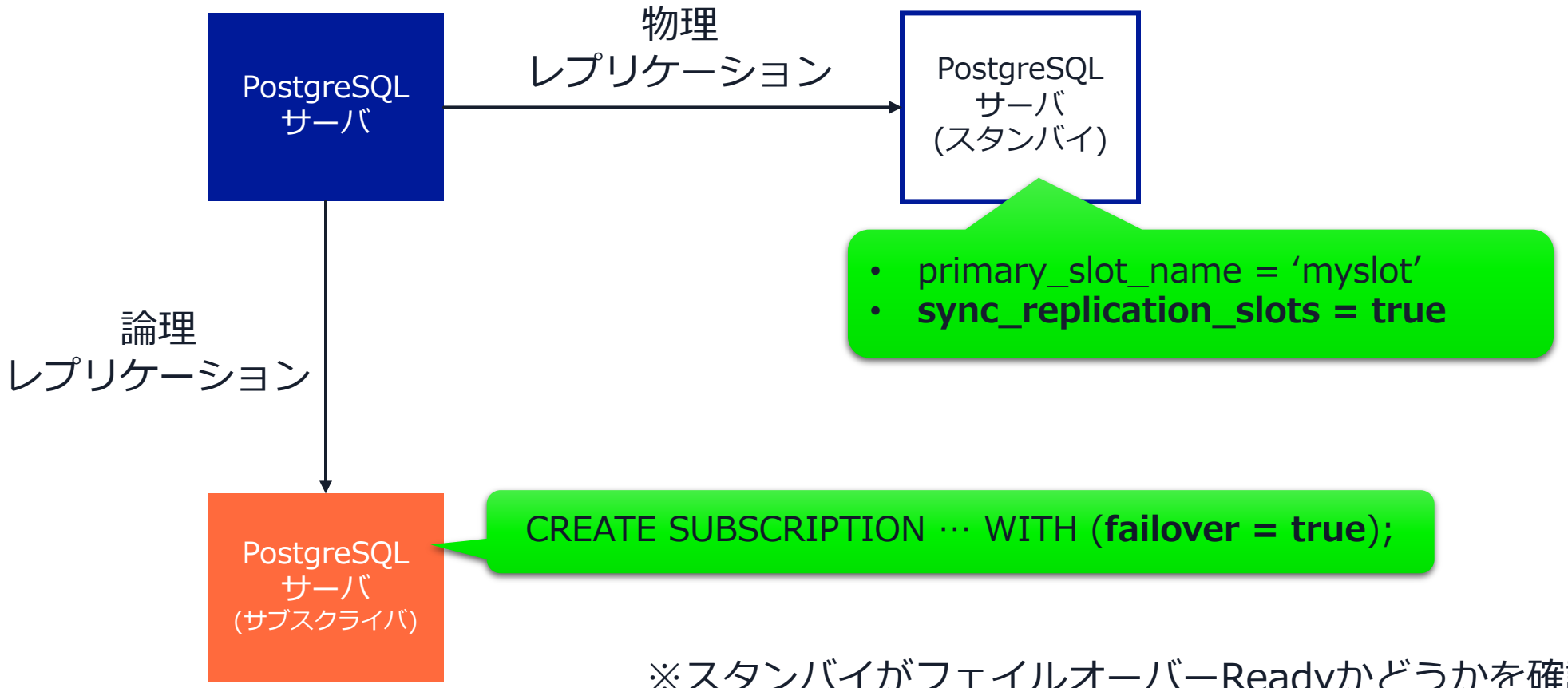


フェイルオーバー対応に必要な設定



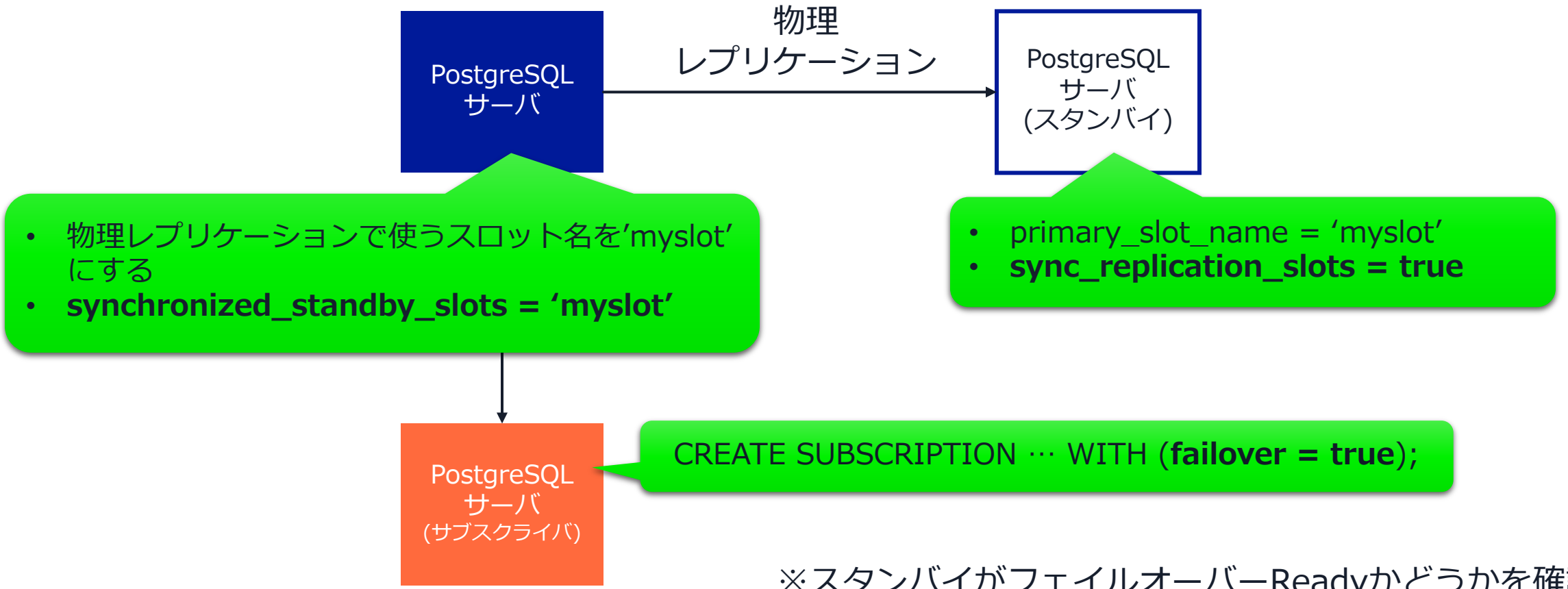
※スタンバイがフェイルオーバーReadyかどうかを確認する手順の詳細はマニュアルに記載されています

フェイルオーバー対応に必要な設定



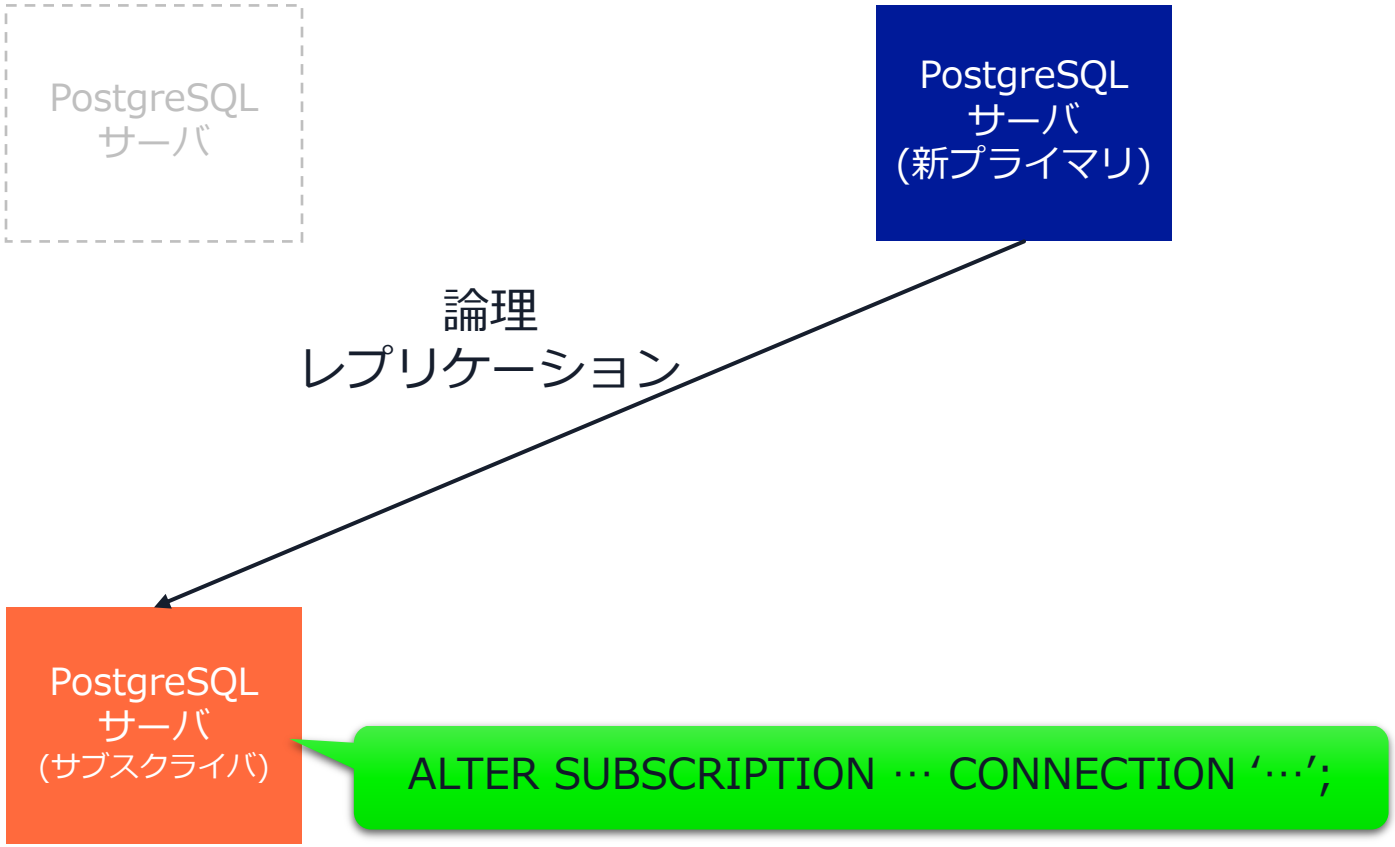
※スタンバイがフェイルオーバーReadyかどうかを確認する手順の詳細はマニュアルに記載されています

フェイルオーバー対応に必要な設定



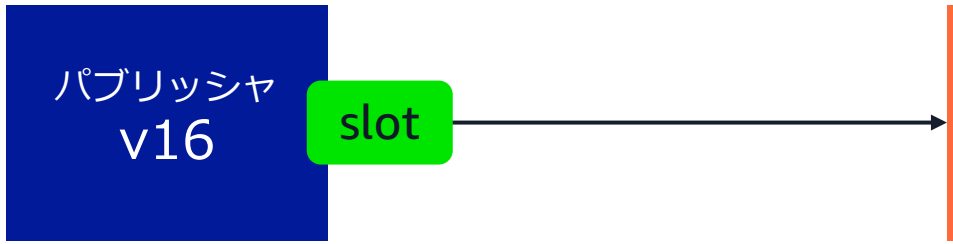
※スタンバイがフェイルオーバーReadyかどうかを確認する手順の詳細はマニュアルに記載されています

論理レプリケーションの再開手順



pg_upgradeにある問題(v16まで)

- pg_upgrade
  - 既存のデータベースクラスタをアップグレードするためのツール（コマンド）
  - 新しいバージョンのデータベースクラスタにテーブルファイル等をコピーする（もしくはリンクする）ことで実現する
- レプリケーションスロットとpg_subscription_relデータは引き継がれなかった
  - pg_subscription_relシステムカタログ：どのテーブルに対する変更を受信するか、が管理されている

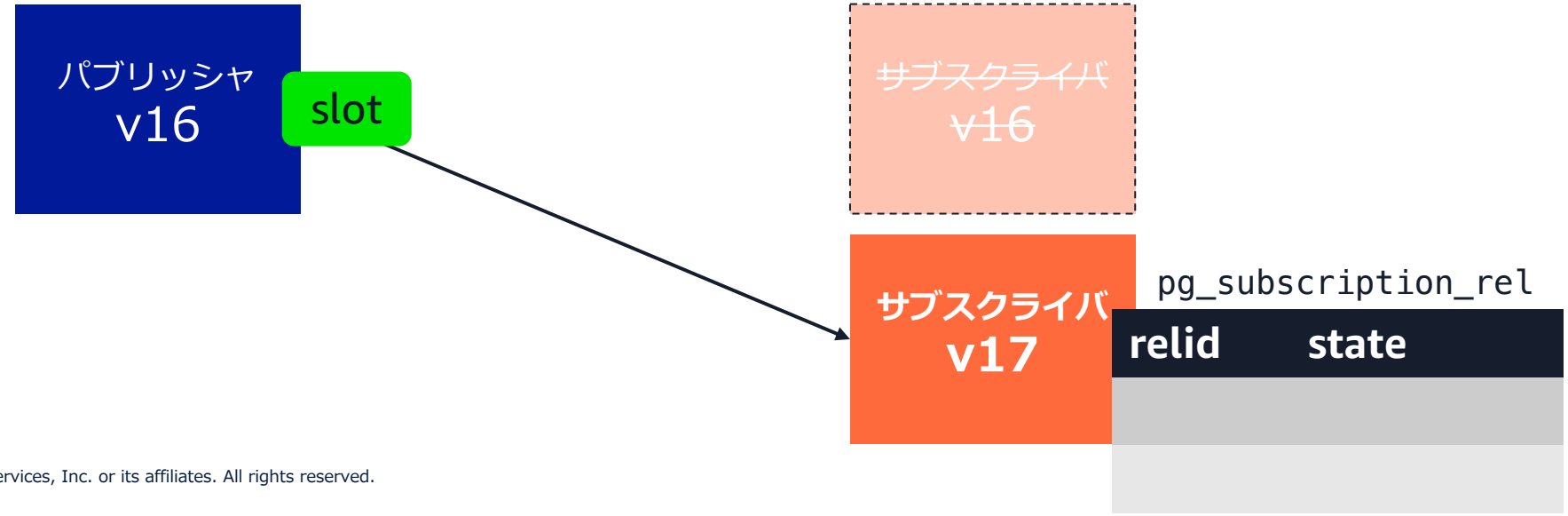


pg_subscription_rel

relid	state
12345	ready
23456	data-sync

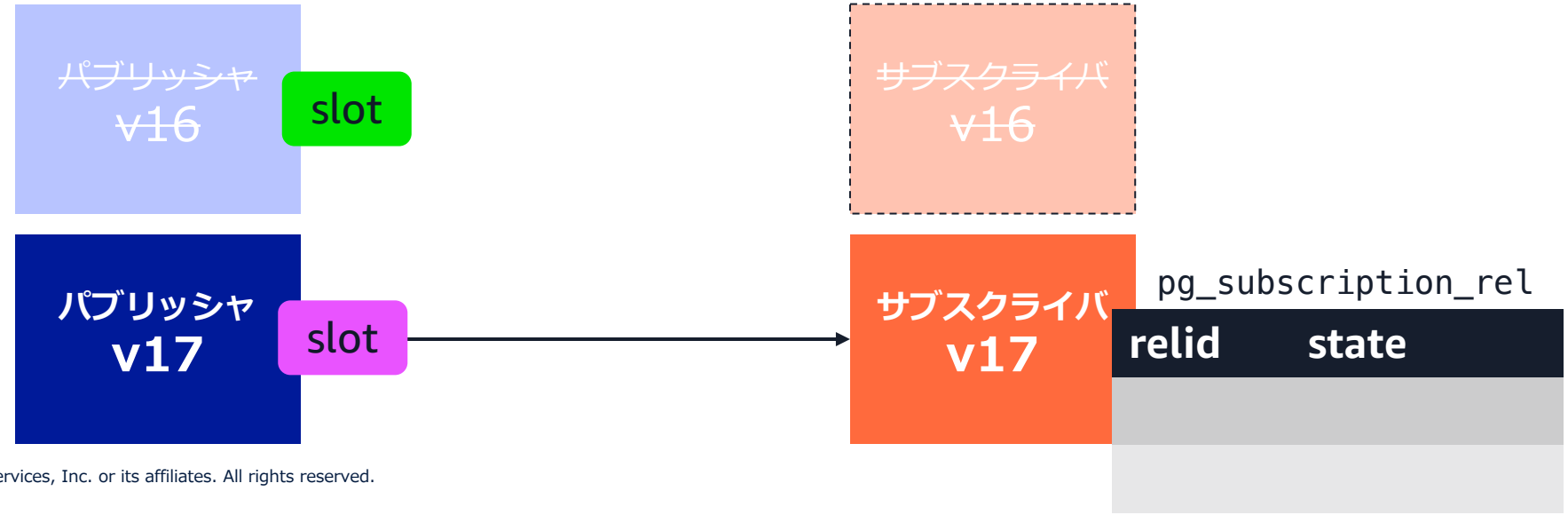
pg_upgradeにある問題(v16まで)

- pg_upgrade
  - 既存のデータベースクラスタをアップグレードするためのツール（コマンド）
  - 新しいバージョンのデータベースクラスタにテーブルファイル等をコピーする（もしくはリンクする）ことで実現する
- レプリケーションスロットとpg_subscription_relデータは引き継がれなかった
  - pg_subscription_relシステムカタログ：どのテーブルに対する変更を受信するか、が管理されている



pg_upgrade後も論理レプリケーションが継続可能に

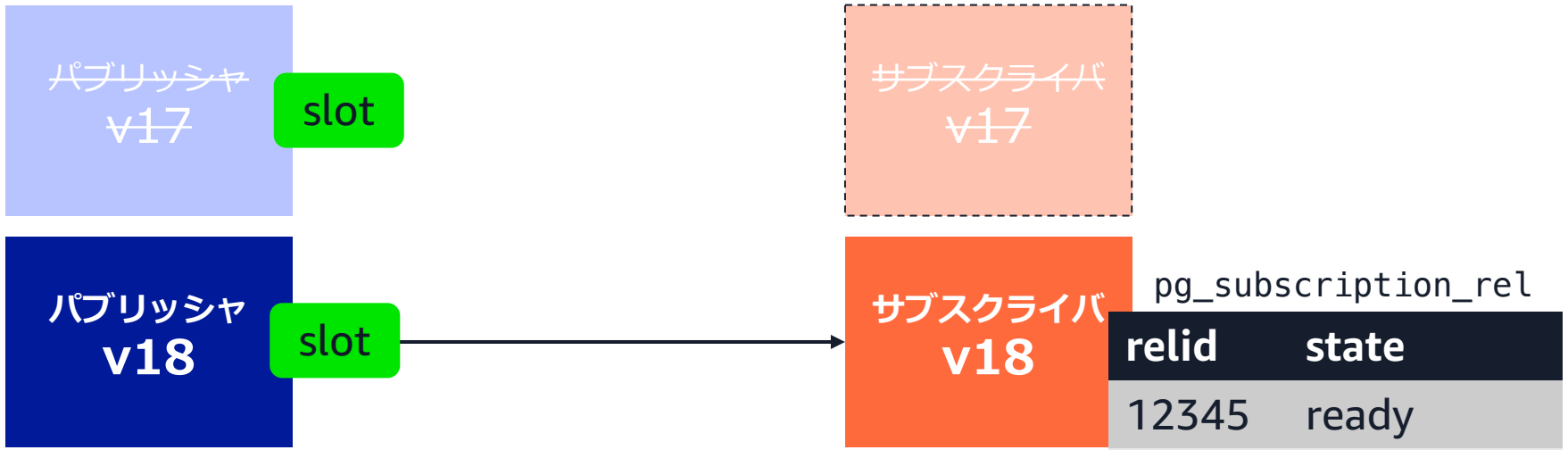
- pg_upgrade
  - 既存のデータベースクラスタをアップグレードするためのツール（コマンド）
  - 新しいバージョンのデータベースクラスタにテーブルファイル等をコピーする（もしくはリンクする）ことで実現する
- レプリケーションスロットとpg_subscription_relデータは引き継がれなかった
  - pg_subscription_relシステムカタログ：どのテーブルに対する変更を受信するかが管理されている



pg_upgrade後も論理レプリケーションが継続可能に

- **v17から**のアップグレードでは、どちらも引き継がれるようになった

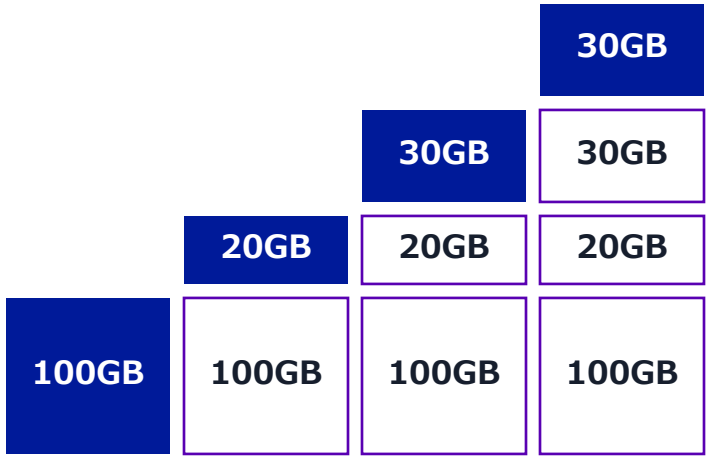
  ※ v16->v17は不可。シャットダウン時にconfirm_flush_lsnがディスクに保存される保証がないため
- アップグレード後にサーバを起動するだけで、論理レプリケーションが再開される
  - pg_upgrade実行前に、パブリッシャで起こった変更がすべてサブスクライバで適用済みである必要がある



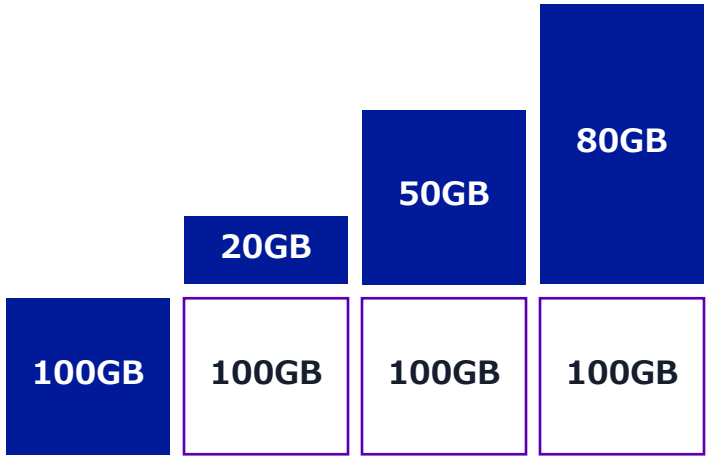
増分バックアップ(Incremental Backup)

- 更新されたデータのみバックアップする手法
  - 差分バックアップ≠増分バックアップ
- バックアップサイズ、時間の削減
- ベースバックアップ＋（増分バックアップ＊N）で最新時点まで復元できる

増分バックアップ

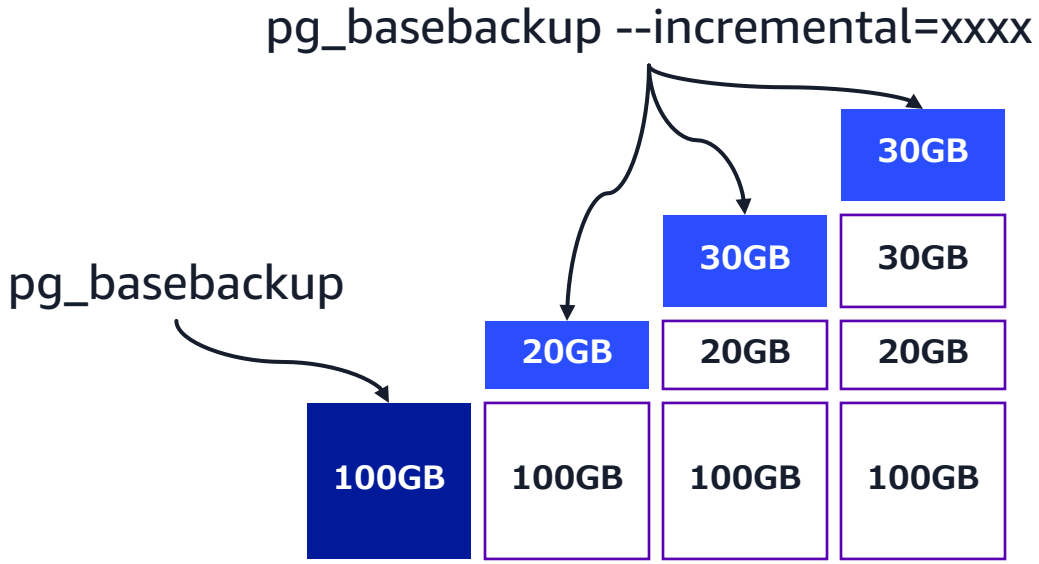


差分バックアップ



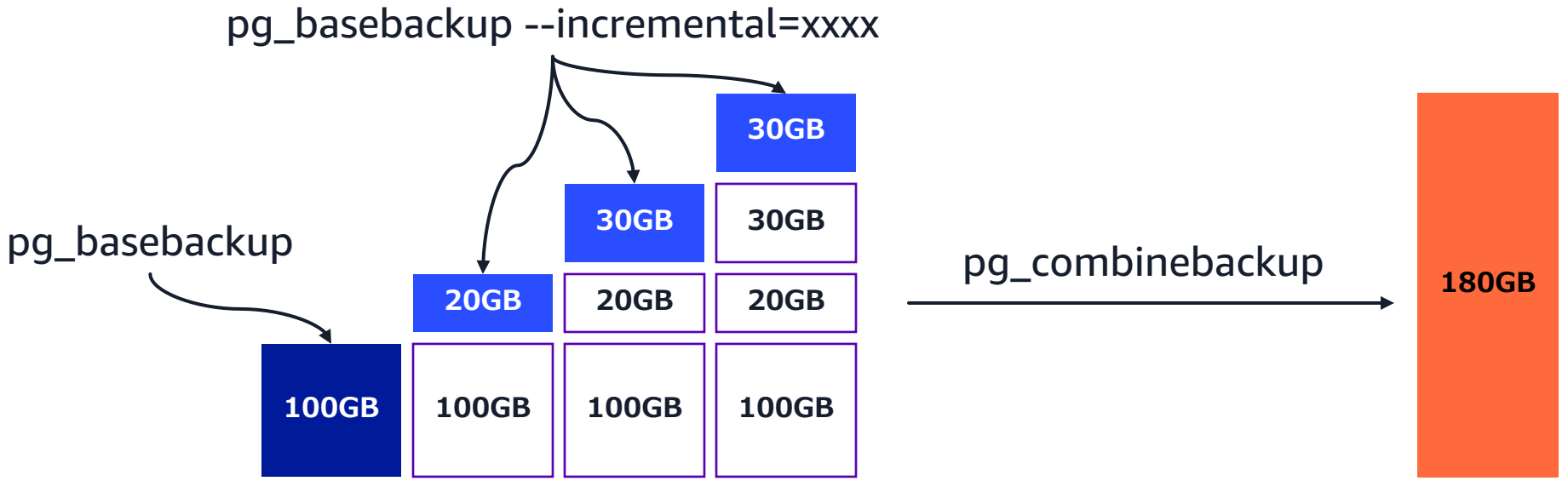
増分バックアップの取得

```
$ pg_basebackup –D sunday
$ pg_basebackup ––incremental=sunday/backup_manifest –D monday
  # ––incrementalオプションを指定することで増分バックアップを取得
$ pg_basebackup ––incremental=monday/backup_manifest –D tuesday
```



pg_combinebackup

```
## ベースと増分を組み合わせてリストア可能なバックアップを作成
$ pg_combinebackup sunday monday tuesday -o tuesday-full
## 増分バックアップを更に組み合わせることも可能
$ pg_combinebackup tuesday-full wednesday thursday friday  saturday -o week-full
```



wal summarizerプロセス

- 「summarize_wal = on」で起動
  - 増分バックアップを使うためにはon必須
- どのブロックが変更されたのか？を検出するためにWALを解析してくれるプロセス
- $PGDATA/pg_wal/summariesディレクトリのWALのサマリファイルを書き出す
  - wal_summary_keep_timeでサマリファイルを保持する期間を指定
- WALサマリファイルはpg_walsummaryコマンドで確認することも可能

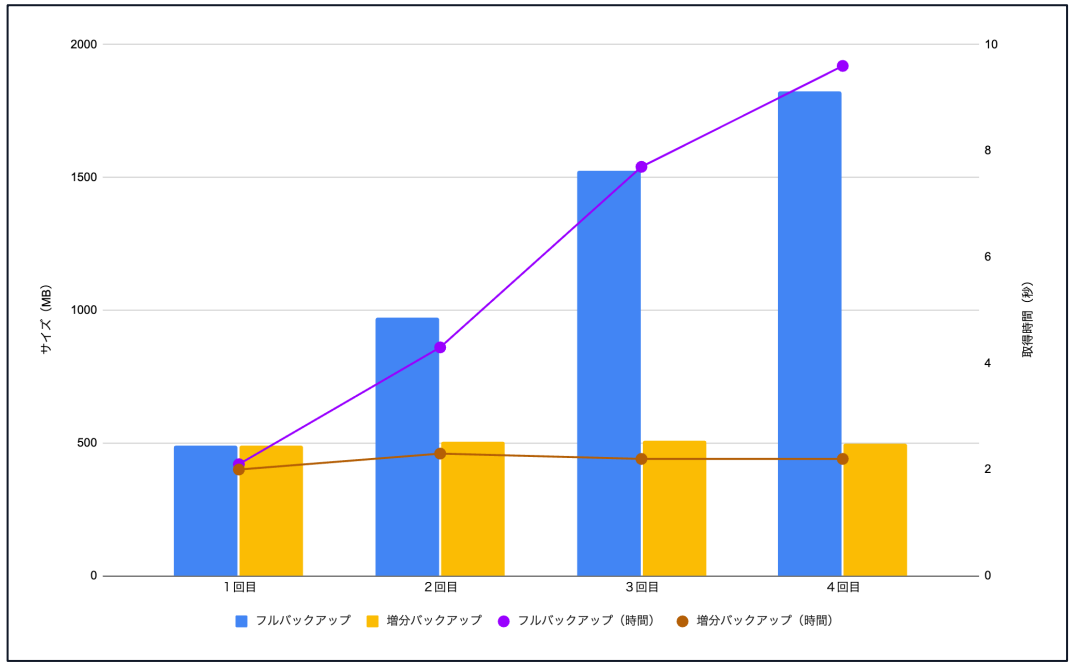
```
$ ls $PGDATA/pg_wal/summaries
000000010000000009000028000000016000028.summary 000000010000000001B000028000000001D000028.summary
000000010000000001F000028000000021000028.summary 000000010000000022000028000000024000028.summary

$ pg_walsummary $PGDATA/pg_wal/summaries/000000010000000001B000028000000001D000028.summary
TS 1663, DB 5, REL 1259, FORK main: block 0
TS 1663, DB 5, REL 2619, FORK main: block 21
TS 1663, DB 5, REL 2696, FORK main: block 2
TS 1663, DB 5, REL 16411, FORK main: blocks 0..4424
TS 1663, DB 5, REL 16411, FORK vm: block 0
```

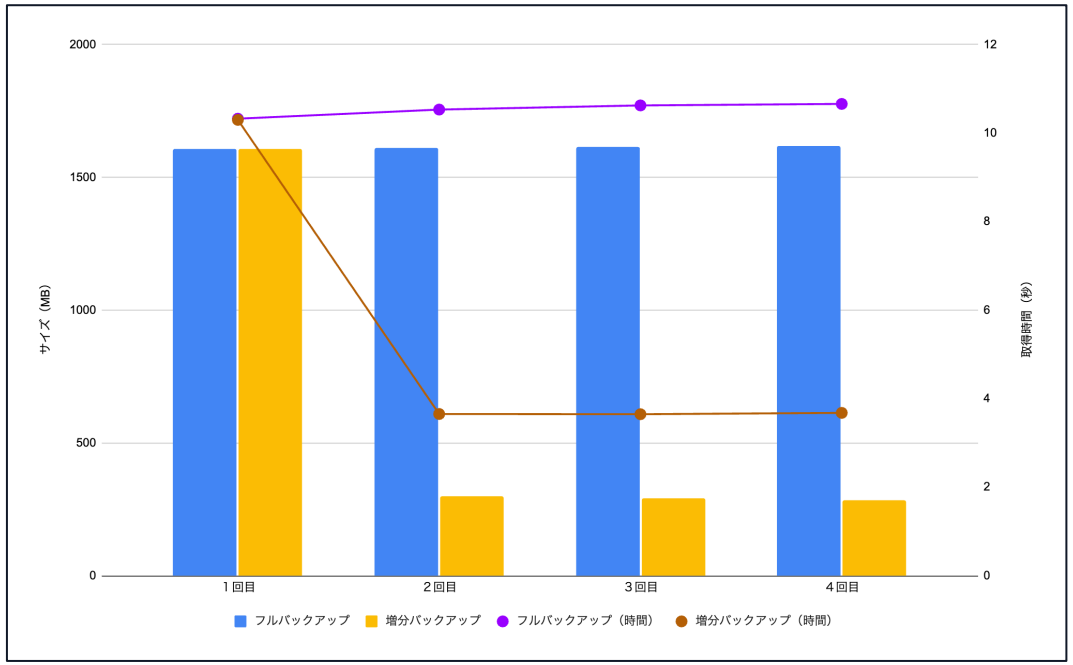
バックアップサイズ、取得時間の比較

- 異なるタイプのトランザクションを流しながら、バックアップを取得する
  - ケース１：INSERTのみ
    - `pg_combinebackup incr1 incr2 incr3 incr4 –o full` は約7秒で完了
  - ケース２：pgbenchを実行する

ケース１



ケース2



留意点・気づいたこと

- --incrementalオプションには、前回取得したフルまたは増分バックアップを指定する
- (auto)vacuumもページを変更することに注意
- tar形式で取得したバックアップに対しても、増分バックアップの取得は可能
  - ただし、pg_combinebackupはtar形式をサポートしていない
- summarize_walは設定ファイルのリロードでon/offが可能
- 複数の増分バックアップを1つの増分バックアップにまとめることは（まだ）サポートされていない

```
$ bin/pg_basebackup -D bkup --format=tar --gzip
$ ls bkup
backup_manifest  base.tar.gz  pg_wal.tar.gz
```

COPYコマンドの改善

- ON_ERRORオプションが追加
- LOG_VERBOSITYオプションが追加
- エンコーディングの変換が不要な場合の性能改善

ON_ERROR、LOG_VERBOSITYオプション

- 不正なデータ（対象列のデータ型へ変換できない）をスキップできるようになった
- ON_ERROR = STOP | IGNORE
  - データ変換エラーが発生したときの動作を指定する（列が足りない、主キー重複などは依然エラー）
- LOG_VERBOSITY = DEFAULT | VERBOSE
  - スキップした行数、データなどの表示する

ON_ERROR、LOG_VERBOSITYオプション

```
$ cat /tmp/test-data.csv
1,1
2,a
a,a

$ psql
=# create table test (a int, b int);
CREATE TABLE

=# copy test from '/tmp/test-data.csv' with (format 'csv');
ERROR:  invalid input syntax for type integer: "a"
CONTEXT:  COPY test, line 2, column b: "a"

=# copy test from '/tmp/test-data.csv' with (format 'csv', on_error 'ignore');
NOTICE:  2 rows were skipped due to data type incompatibility
COPY 1

=# table test;
 a | b
---+---
 1 | 1
(1 row)

=# copy test from '/tmp/test-data.csv' with (format 'csv', on_error 'ignore', log_verbosity 'verbose');
NOTICE:  skipping row due to data type incompatibility at line 2 for column "b": "a"
NOTICE:  skipping row due to data type incompatibility at line 3 for column "a": "a"
NOTICE:  2 rows were skipped due to data type incompatibility
COPY 1
```

INT型の列を２つ持つテーブルを作成

INT型への変換エラー

エラーとなる行はスキップされる

エラーとなった行の詳細が出力される

モニタリングの強化

- pg_stat_progress_vacuumビューにインデックスVacuumの進捗も表示されるようになった
  - indexes_total列とindexes_processed列
- pg_stat_checkpointerビューが追加
  - pg_stat_bgwriterビューに含まれていたいくつかの列が移動している
- pg_wait_eventビューが追加
  - 待機イベントのタイプとイベント名の一覧が見れるビュー

```
=# select * from pg_wait_events limit 2;
   type   |      name       |                    description
----------+-----------------+-----------------------------------------------
 Activity | ArchiverMain    | Waiting in main loop of archiver process
 Activity | AutovacuumMain | Waiting in main loop of autovacuum launcher process
(2 rows)
```

PostgreSQL開発最前線

新バージョンの開発

- 2024/7からすでにPostgreSQL 18の開発がスタート
  - リリースは2025年秋予定
- これまでに1110件がすでにコミット済み
- 様々な新機能、改善案が議論されています
- Commit Festには300件以上がエントリ

PostgreSQL 18以降に入るかも？①

- 非同期I/O (Asynchronous I/O)
- マルチスレッド化
- Skip Scan対応（Btree）
- SQL関連機能の追加
  - Row Pattern Recognition
  - SQL/PGQ

PostgreSQL 18以降に入るかも？②

- モニタリング関連
  - Vacuumの統計情報
  - バックエンド毎のI/O統計情報
- マテリアライズド・ビューの差分更新
- 論理レプリケーションの改善
  - 衝突検知、解決
  - シーケンスレプリケーション
  - DDLレプリケーション

PostgreSQL 18以降に入るかも？③

- 統計情報のインポート・エクスポート
- 列レベルの透過的暗号化
- COPYコマンドの強化（カスタムフォーマット、JSONなど）
- パラレルクエリ
  - パラレルVacuumの強化
  - GINインデックスの作成
- 再起動なしでshared_buffersのサイズを変更する

PostgreSQL開発者を増やす取り組み

- Google Summer of Code (GSoC)
  - 特定のプロジェクトを提案し、採択されればメンターがつきながら開発ができる(学生向け？)
- Advanced Patch Feedback Session at PGCon.dev 2024
  - コミッタが対面でパッチについてアドバイスをくれるワークショップ
- Mentoring program for code contributions
  - コミッタがメンターとして付いてくれるプログラム（登録制）
- PostgreSQL Hacking Workshop
  - 毎月、特定のテーマに沿ってPostgreSQLの開発という観点で参加者で議論する（登録制）
  - Discordサーバもあり、気軽にコミッタ等に質問できる

まとめ

- Vacuumが省メモリかつ効率的に動作
- 論理レプリケーションの改善
- 増分バックアップ
- COPYの性能向上、機能追加
- SQL/JSON実装の強化
- MERGEコマンドへの機能追加
- プラットフォーム非依存のCollation（照合順序）
- 現在議論中の機能も魅力的なものばかり

参考資料

- 篠田の虎の巻「PostgreSQL 17 GA新機能検証結果」
  - https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/lcc/pdf/PostgreSQL_17_GA_New_Features_ja_20240930_1.pdf
- PostgreSQL 17 検証レポート
  - https://www.sraoss.co.jp/tech-blog/wp-content/uploads/2024/10/pg17_report_20241021.pdf
- Waiting for Postgres 17: Incremental base backups
  - https://pganalyze.com/blog/5mins-postgres-17-incremental-backups

Thank you!

澤田　雅彦

sawadamm@amazon.com