NTT DATA
Trusted Global Innovator

さらなる安定稼働を目指したPostgreSQLモニタリング機能の拡充

2020年11月13日

株式会社NTTデータ　鳥越淳、池田真洋

自己紹介

- 鳥越　淳(とりこし　あつし)
  2008年頃からオープンソースを扱う業務に従事
  PostgreSQLは9.6頃から
  『PostgreSQL徹底入門 第4版』(翔泳社)8～13章執筆

- 池田真洋(いけだ　まさひろ)
  2016年頃からコネクティッドカー向けの検証支援の業務に従事
  2020年5月頃からPostgreSQLの開発業務に参画

本講演について

本講演で紹介する機能や仕様は、開発中のものが多く含まれています。
そのため、将来的に変更される可能性があることにご注意ください。

その他、記載されている会社名、商品名、又はサービス名は、 各社の登録商標又は商標です。

目次

1. モニタリングとは
2. OSレイヤ（Disk I/O, CPU利用時間）に関する統計情報
3. WALに関する統計情報
4. メモリ利用状況に関する統計情報
5. 実行計画に関する統計情報

NTT DATA

モニタリングとは

モニタリング機能拡充の背景

システムのトラブルを事前に把握したり、トラブルを迅速に解決するためには、データベースの処理状況をモニタリングする必要があります。

モニタリングの実現には、情報を蓄積・加工・可視化する仕組みも必要になりますが、そもそもデータベースが必要な情報を提供していることが前提となります。

基幹システムのオープン化や商用データベース製品のライセンス問題を背景に、社会インフラ系基幹システムにおいてもPostgreSQLの適用が本格化しており、より高度なデータベース内部の情報を公開する仕組みが求められています。

そこで、本講演では、PostgreSQL v13の本体が提供している情報を整理した上で、

- 現状、提供されていない情報を取得するためのワークアラウンド
- 新たな情報をv14～で追加するために実施しているコミュニティへの提案状況

について、ご紹介いたします。

本日ご紹介する拡充を目指している情報

様々な情報の拡充を目指していますが、本日は下記4つに関する情報を共有いたします。

① OSレイヤ（Disk I/O, CPU利用時間）に関する統計情報
→ ワークアラウンドを2つご紹介（拡張機能「pg_stat_kcache」など）

② WALに関する統計情報
→ WAL bufferサイズのチューニングなどに必要な情報の追加

③ メモリ利用状況に関する統計情報
→ バックエンドプロセスのメモリ使用状況の把握

④ 実行計画に関する統計情報
→ generic/customプランの生成状況の把握

OSレイヤ（Disk I/O, CPU利用時間）に関する統計情報

Disk I/Oに関して、V13で取得可能な情報

pg_stat_statementsを活用することで、ブロック操作に関する情報が取得可能

- Read/Writeしたブロック数、キャッシュヒット数などを利用して、ボトルネックのクエリ特定などに利用

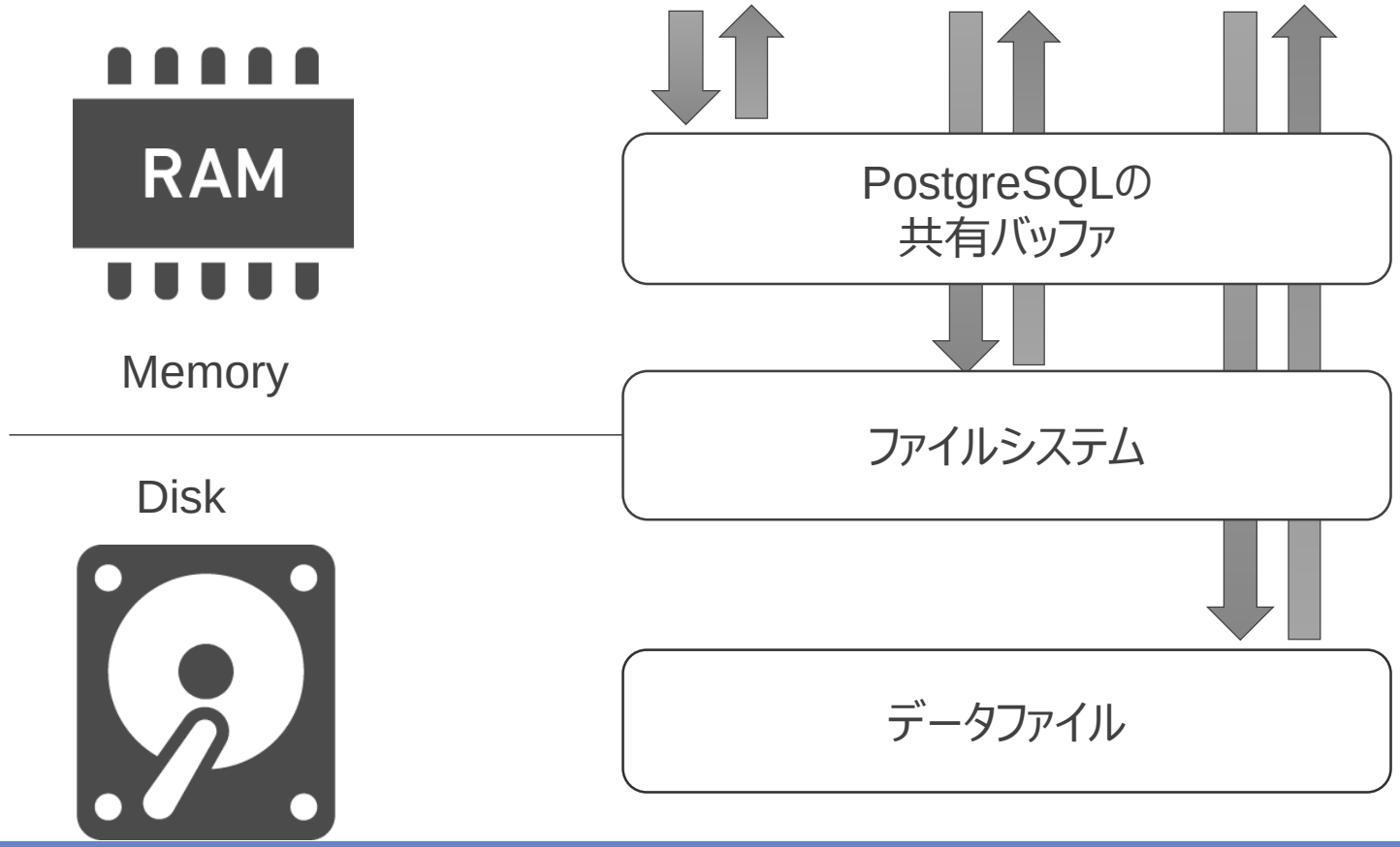
pg_stat_statementsビュー（ブロック操作に関する情報を一部抜粋）

```
query                | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
shared_blks_hit      | 157486      # SQL文によってヒットした共有ブロックキャッシュの総数
shared_blks_read     | 7799        # SQL文によって読み込まれた共有ブロックの総数
shared_blks_dirtied  | 2041
shared_blks_written  | 3490
local_blks_hit       | 0
local_blks_read      | 0
local_blks_dirtied   | 0
local_blks_written   | 0
temp_blks_read       | 0
temp_blks_written    | 0
blk_read_time        | 42.99650900000006
blk_write_time       | 53.03752099999988
```

ブロック取得フローの3パターン

PostgreSQLでは、共有バッファだけでなく、ファイルシステム上のキャッシュも利用している

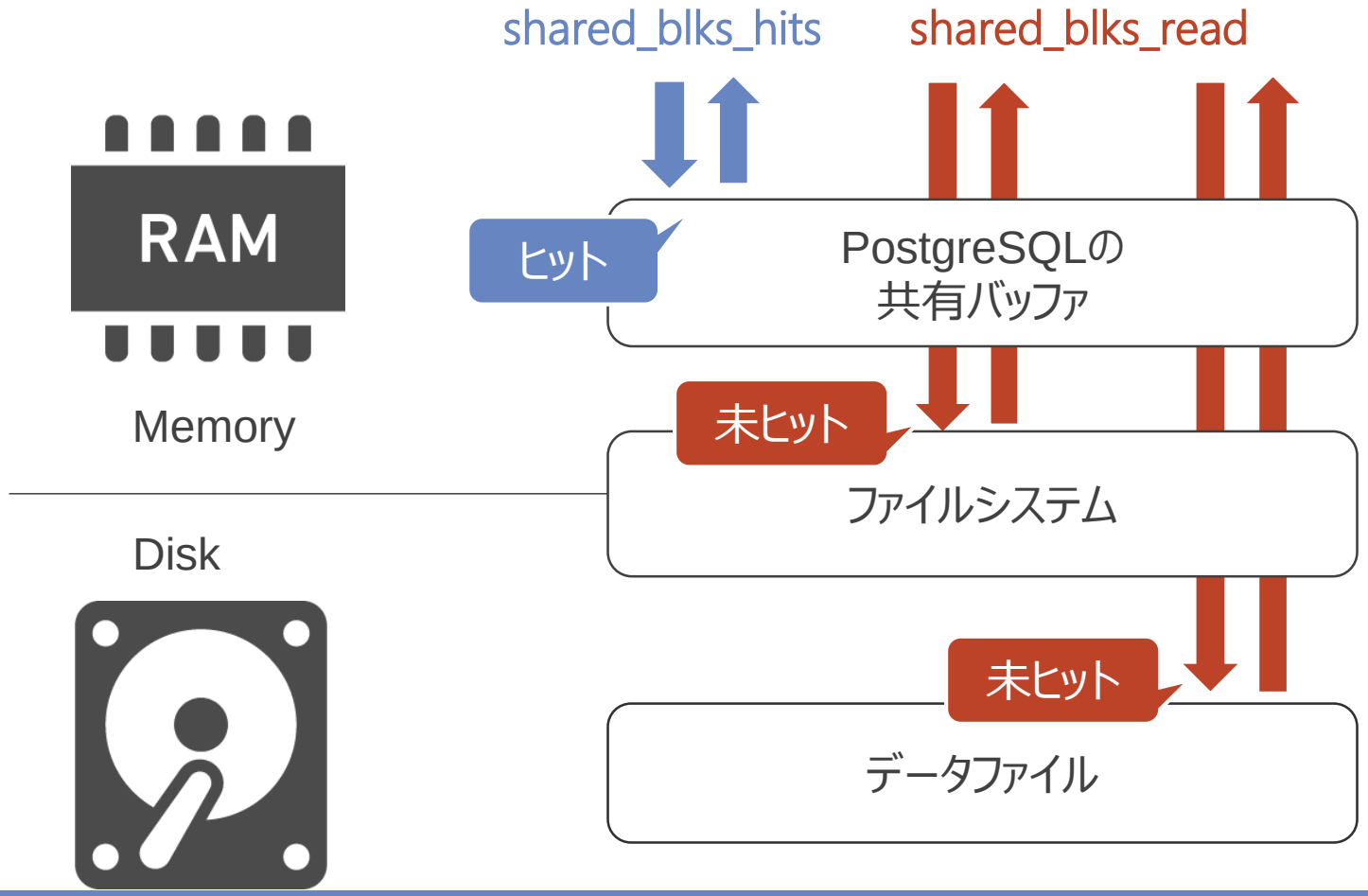
ブロック取得フローの3パターン



キャッシュヒット（shared_blks_hits）が表すもの

しかし、pg_stat_statementsでは、共有バッファにヒットしたかどうかで算出している

ブロック取得フローの3パターンとキャッシュヒット

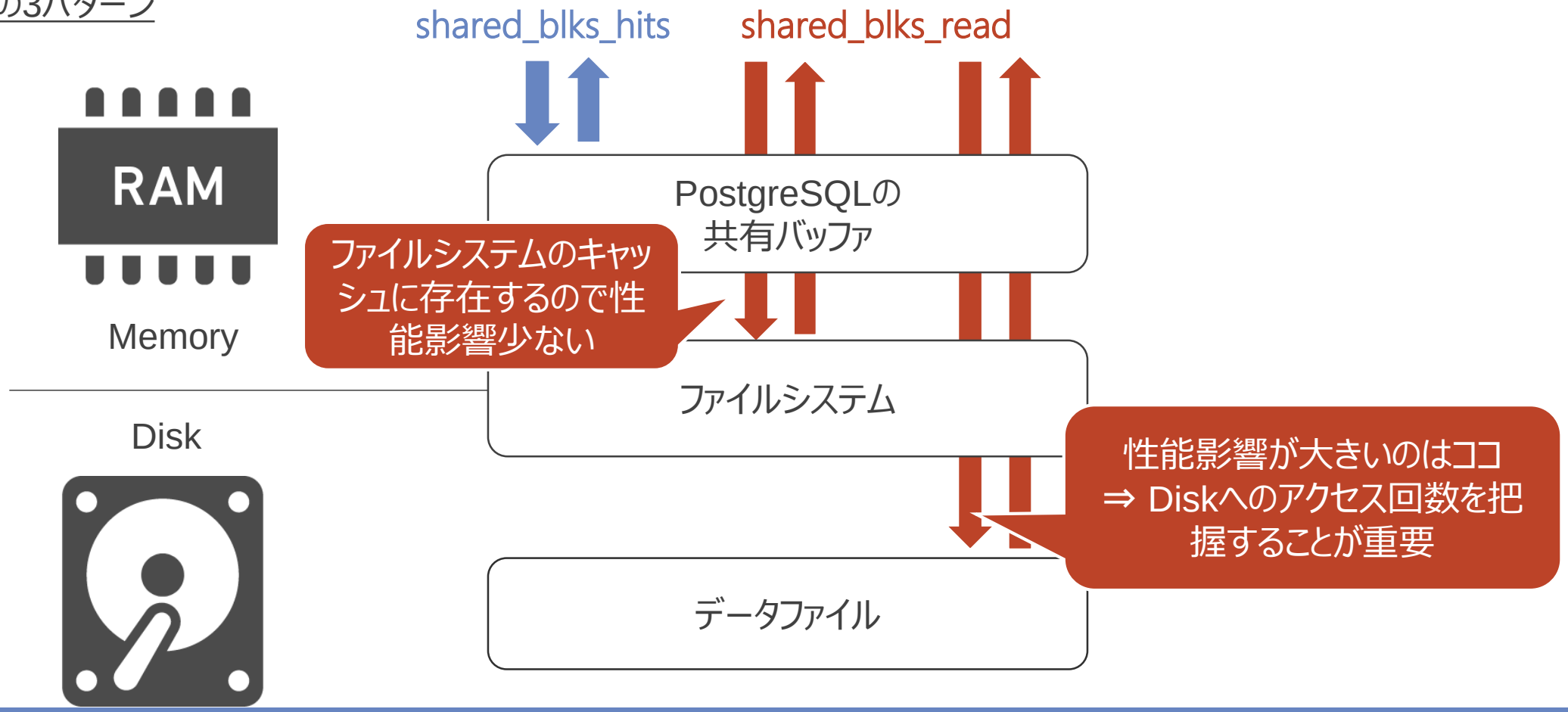


Disk I/Oの統計情報取得に関する課題

性能劣化に大きな影響を与えるDiskへのI/O回数の取得はできない

- shared_blks_hitsの値が低くても、ファイルシステムのキャッシュに存在しているかもしれない

ブロック取得フローの3パターンとキャッシュヒット



NTT DATA

CPU利用時間に関して、v13で取得可能な情報

log_statement_statsパラメータで、実行したクエリごとのCPU使用量をログに出力可能

- log_parser_stats, log_planner_stats, log_executor_statsパラメータにより、より細かいフェーズでもCPU利用時間を取得可能

ログ出力結果

```
2020-10-30 17:09:41 JST [client backend] LOG:  statement: SELECT
md5(clock_timestamp()::text) FROM generate_series(1, 10000);
2020-10-30 17:09:41 JST [client backend] LOG:  QUERY STATISTICS
2020-10-30 17:09:41 JST [client backend] DETAIL:  ! system usage stats:
        !       0.029592 s user, 0.003074 s system, 0.033262 s elapsed
        !       [0.038423 s user, 0.007687 s system total]
        !       6960 kB max resident size
        !       0/0 [0/0] filesystem blocks in/out
        !       0/655 [0/1814] page faults/reclaims, 0 [0] swaps
        !       0 [0] signals rcvd, 0/52 [2/56] messages rcvd/sent
```

user/systemのCPU利用時間

CPU使用時間の統計情報取得に関する課題

クエリを実行するごとに大量のログが生成されるため、運用には適さない

- ログの生成量が大幅に増加するため、Disk領域を圧迫しまう
- 個々の性能ネックとなっているクエリの抽出作業も難しい ⇒ 別に統計処理が必要

右図は、1つのクエリを実行したときに出力されるログ（Parse, Analysis, Rewrite, Plan, Executeのそれぞれフェーズで出力）

1つのクエリに対して、50行程度のログが出力される

pgbenchのデフォルトトランザクションで
1000TPSで処理した場合、1時間で18GB！
⇒ 検証であれば良いが、運用には適さない

```
2020-10-30 16:44:47 JST [client backend] LOG: PARSER STATISTICS
2020-10-30 16:44:47 JST [client backend] DETAIL: ! system usage stats:
! 0.000247 s user, 0.000144 s system, 0.000397 s elapsed
! [0.002514 s user, 0.002025 s system total]
! 3712 kB max resident size
! 0/0 [0/0] filesystem blocks in/out
! 0/81 [0/997] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [2/4] messages rcvd/sent
! 0/1 [0/7] voluntary/involuntary context switches
2020-10-30 16:44:47 JST [client backend] STATEMENT: SELECT md5(clock_timestamp()::text) FROM generate_series(1, 10000);
2020-10-30 16:44:47 JST [client backend] LOG: PARSE ANALYSIS STATISTICS
2020-10-30 16:44:47 JST [client backend] DETAIL: ! system usage stats:
! 0.001655 s user, 0.002520 s system, 0.007257 s elapsed
! [0.004284 s user, 0.004611 s system total]
! 4600 kB max resident size
! 0/0 [0/0] filesystem blocks in/out
! 0/200 [0/1221] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [2/4] messages rcvd/sent
! 11/11 [11/18] voluntary/involuntary context switches
2020-10-30 16:44:47 JST [client backend] STATEMENT: SELECT md5(clock_timestamp()::text) FROM generate_series(1, 10000);
2020-10-30 16:44:47 JST [client backend] LOG: REWRITER STATISTICS
2020-10-30 16:44:47 JST [client backend] DETAIL: ! system usage stats:
! 0.000011 s user, 0.000006 s system, 0.000016 s elapsed
! [0.004333 s user, 0.004635 s system total]
! 4620 kB max resident size
! 0/0 [0/0] filesystem blocks in/out
! 0/5 [0/1226] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [2/4] messages rcvd/sent
! 0/0 [11/18] voluntary/involuntary context switches
2020-10-30 16:44:47 JST [client backend] STATEMENT: SELECT md5(clock_timestamp()::text) FROM generate_series(1, 10000);
2020-10-30 16:44:47 JST [client backend] LOG: PLANNER STATISTICS
2020-10-30 16:44:47 JST [client backend] DETAIL: ! system usage stats:
! 0.000299 s user, 0.000187 s system, 0.000775 s elapsed
! [0.004661 s user, 0.004839 s system total]
! 5012 kB max resident size
! 0/0 [0/0] filesystem blocks in/out
! 0/98 [0/1324] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [2/4] messages rcvd/sent
! 1/0 [13/18] voluntary/involuntary context switches
2020-10-30 16:44:47 JST [client backend] STATEMENT: SELECT md5(clock_timestamp()::text) FROM generate_series(1, 10000);
2020-10-30 16:44:47 JST [client backend] LOG: EXECUTOR STATISTICS
2020-10-30 16:44:47 JST [client backend] DETAIL: ! system usage stats:
! 0.026559 s user, 0.000924 s system, 0.027719 s elapsed
! [0.031387 s user, 0.005850 s system total]
! 6052 kB max resident size
! 0/0 [0/0] filesystem blocks in/out
! 0/235 [0/1592] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/52 [2/56] messages rcvd/sent
```

OSレイヤの統計情報に関連するワークアラウンド

Disk I/O, CPU使用時間を取得できない課題を解決するためのワークアラウンドの2つ

① 既存のpg_stat_statementsビューで提供されている情報を元に推測

本ビュー以外の情報も利用した総合的な判断は必要だが、問題が存在する可能性については把握可能

（ただし、Disk I/Oのみ。CPU使用時間の推測は困難）

② PostgreSQL本体非同梱のエクステンションの利用

OSレイヤの統計情報を取得可能な「pg_stat_kcache」の活用

① 既存のpg_stat_statementsビューで提供されている情報を元に推測

pg_stat_statementsビューには、ブロック操作時間に関するメトリクスが存在

- blk_read_time, blk_write_timeの値が大きい場合、物理的なDisk I/Oが多数発生している可能性がある
- なお、"track_io_timing"パラメータを有効化することで、メトリクスとして、収取可能

pg_stat_statementsビュー（処理時間に関する情報を一部抜粋）

```
query              | INSERT INTO pgbench_history ...（略）

shared_blks_hit    | 157
shared_blks_read   | 7798719

blk_read_time      | 1015.3106780000143 # SQL文がブロック読み取りに費やした総時間（ミリ秒）
blk_write_time     | 1594.9047660000026 # SQL文がブロック書き出しに費やした総時間（ミリ秒）
```

キャッシュヒット小さく、ブロック読み取り時間も長い
⇒ 物理的なDisk readが多数発生している可能性が高い

② PostgreSQL本体非同梱のエクステンションの利用

pg_stat_kcacheを活用することで、OSレイヤの情報（Disk I/O, CPU）の統計情報が取得可能

- PoWA(PostgreSQL Workload Analyzer)と呼ばれるパフォーマンスツールの1つ
- PostgreSQL License
- ログではなく、各情報を統計化したビューが提供されており、性能分析で活用しやすい

pg_stat_kcache

Features

ファイルシステムの物理的なread/writeの情報が収集可能

Gathers statistics about real reads and writes done by the filesystem layer. It is provided in the form of an extension for PostgreSQL >= 9.4., and requires pg_stat_statements extension to be installed. PostgreSQL 9.4 or more is required as previous version of provided pg_stat_statements didn't expose the queryid field.

(※) https://github.com/powa-team/pg_stat_kcache

pg_stat_kcacheで取得可能な統計情報

物理的なDiskへのアクセス回数とCPU使用時間が取得可能

- 2つのビューを提供しており、データベース単位・クエリ単位で把握可能

pg_stat_kcacheビュー（データベース単位）

```
datname      | testdb
user_time    | 18.659182343999177   # CPU利用時間（user）
system_time  | 2.166676000000015    # CPU利用時間（system）
minflts      | 132412
majflts      | 1
reads        | 34181120             # Disk 読み込み量（バイト単位）
reads_blks   | 4172                 # Disk 読み込みブロック数
writes       | 231849984            # Disk 書き込み量（バイト単位）
writes_blks  | 28302                # Disk 書き込みブロック数
nvcsws       | 49816
nivcsws      | 3933
```

(※) Linuxで使用した場合のビューを記載

pg_stat_kcacheで取得可能な統計情報

クエリ単位でも取得可能。role名なども存在するため、role単位での集計なども可能

- リソース消費量が多いクエリを特定し、性能改善につなげることが可能

pg_stat_kcache_detailビュー（クエリ単位）

```
query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
datname     | testdb
rolname     | ikeda
user_time   | 0.961217999999904
system_time | 0.132768000000007
minflts     | 19889
majflts     | 0
reads       | 32243712
reads_blks  | 3936
writes      | 655360
writes_blks | 80
nvcsws      | 3904
nivcsws     | 3
```

pg_stat_kcacheの仕組み

OSレイヤの情報を取得するために大きく2つの仕組みを利用している

1. PostgreSQLのHook機構

- PostgreSQLの高い拡張性を実現するための機構の１つ
- 処理の途中に割り込み、挙動を変更する仕組みで、多様なHookが用意されている

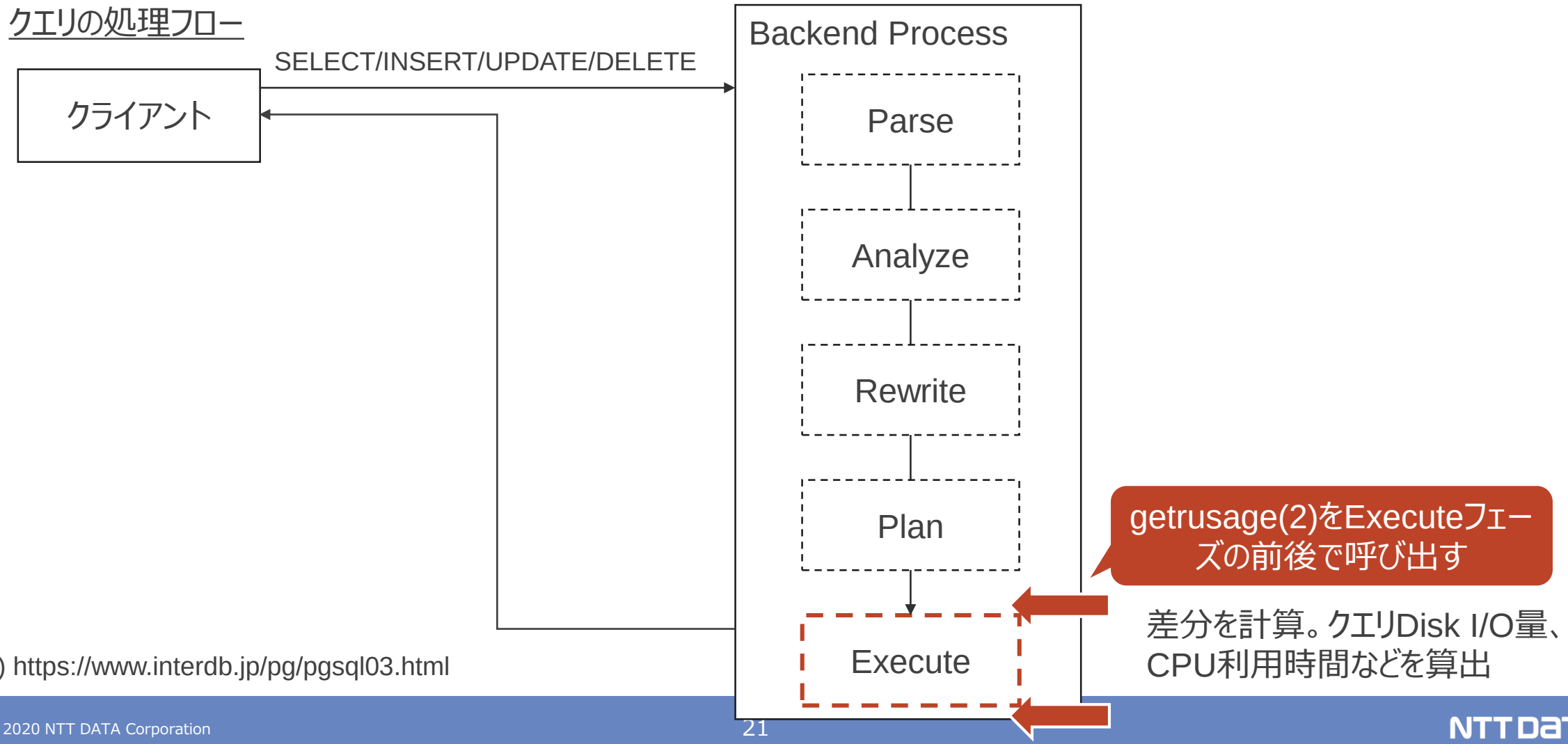
2. getrusage(2)

- 資源の使用量を取得するシステムコール
- バックエンドプロセスが呼び出し、そのプロセスが使用した資源の合計量を取得する

pg_stat_kcacheの仕組み

pg_stat_kcacheでは、クエリ実行前と後に呼ばれるExecuteフェーズのHookを利用している

- したがって、取得できるリソース利用量は、Executeに関する処理のみ



(※ 参考) https://www.interdb.jp/pg/pgsql03.html

現時点でのpg_stat_kcacheの制約や開発状況

実案件への適用の可否を判断するため、調査や検証などを実施しており、確認した制約は下記の通り

（引き続き、開発者へのフィードバックを実施し、改善活動などを続けていく予定です）

① 対応しているコマンドは、DMLのみ

- Executorに関するHookのみを利用しているため、Utilityコマンド（VACUUM, COPY, CREATEなど）については確認できない

② リソース使用量を把握できるフェーズは、Executeフェーズのみ

- ExecutorフェーズのHookのみ利用しているため、その他フェーズ（Planなど）のリソース使用量は把握できない。そのため、実行計画の作成がボトルネックになっているケースなどは、確認できない ⇒ 実装中

③ その他

- パラレルクエリ実行時のworkerリソース使用量が把握できない ⇒ 修正済み
- ネストされているクエリについては、適切なリソース使用量が把握できない ⇒ 実装中

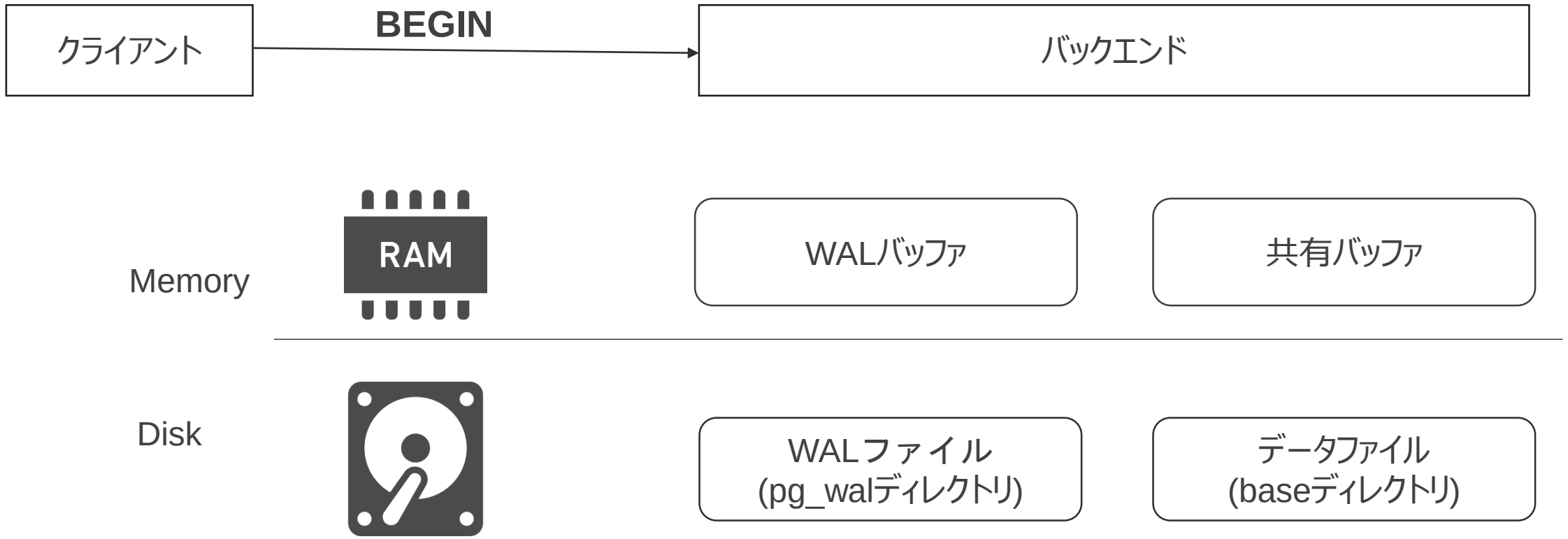
WALに関する統計情報

WALとは

WAL(Write Ahead Logging)は、トランザクションのログを残すための手法

- クラッシュしたときの耐障害性などを保証する事ができる

WALログ書き込みの仕組み

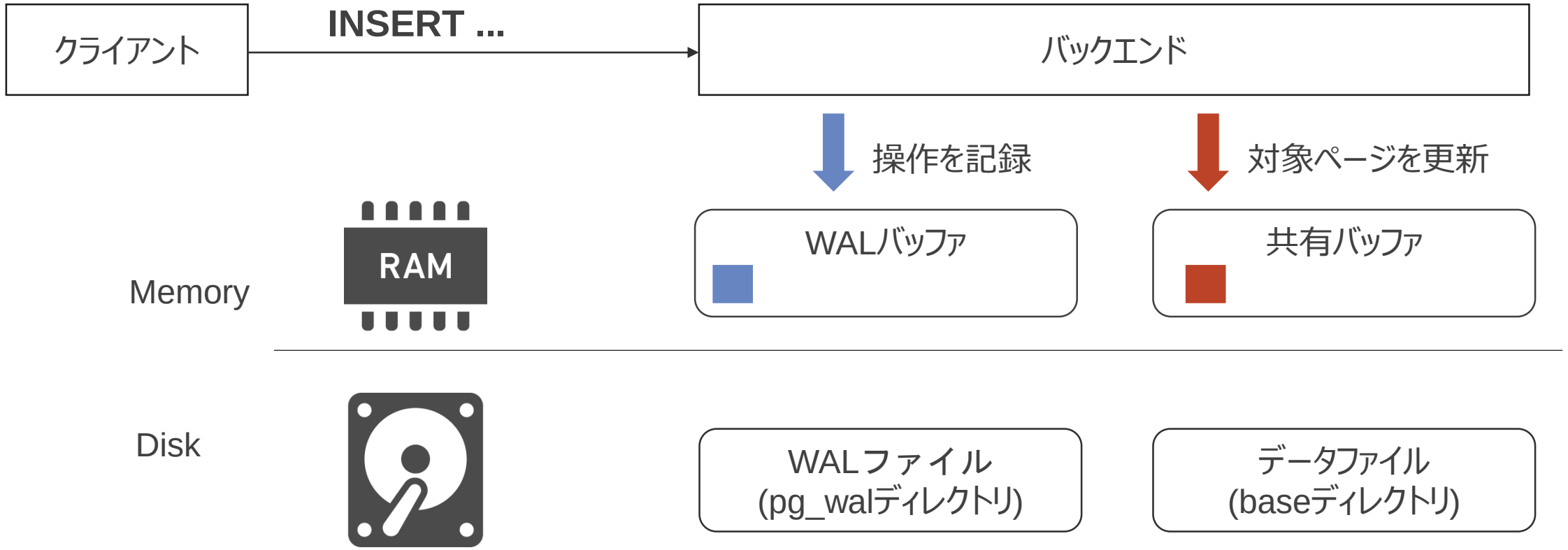


WALとは

INSERTなどのデータ変更を実施するクエリを実行すると、WALバッファに操作を記録する

- 共有バッファも書き換える。この時点で異常停止するとデータが消える可能性があるが、未COMMITなのでOK

WALログ書き込みの仕組み

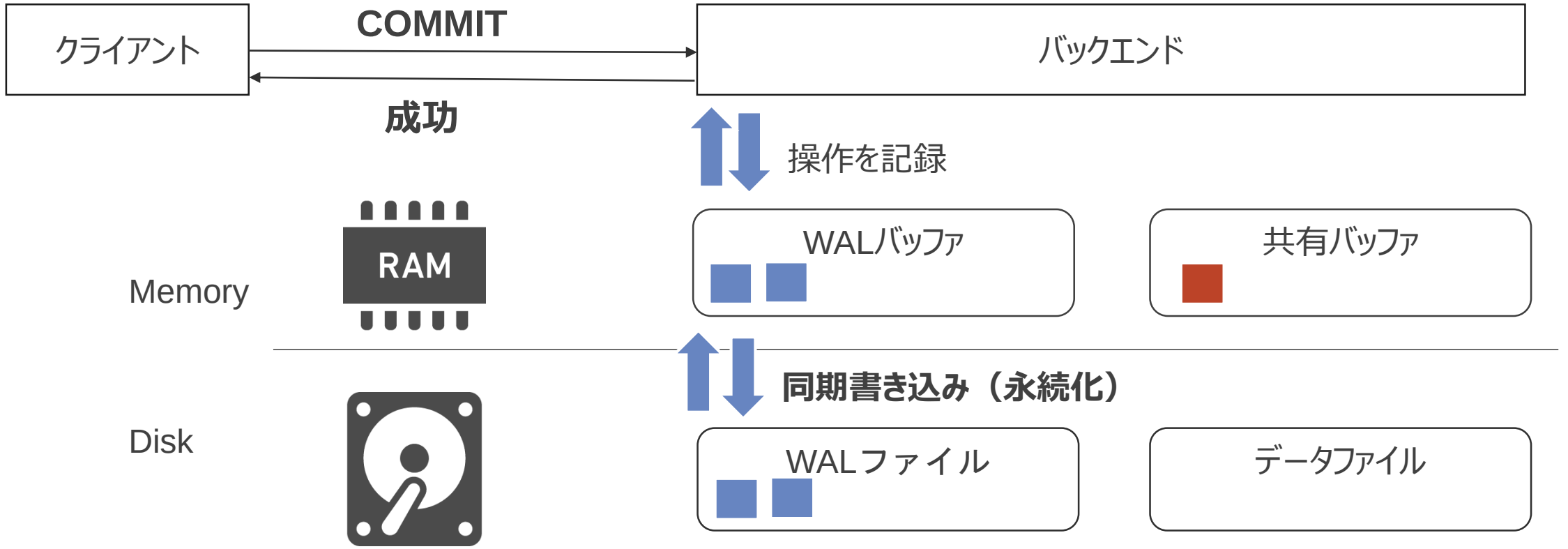


WALとは

COMMITされたデータは永続化されていることを保証する必要がある

- WALファイルにデータをフラッシュすることで、異常停止しても、WALファイルから適切な状態に復帰できる

WALログ書き込みの仕組み



V13で確認できるWALの統計情報

更新が多いワークロードなどの場合、WAL書き出しがDisk I/Oのボトルネックになる可能性がある。

そこで、PostgreSQL v13では、WALに関するメトリクスが取得可能に。

- Autovacuum: バキューム時に生成したWALレコード数など
- EXPLAN, auto_explain, pg_stat_statements: クエリ単位で生成したWALレコード数など

pg_stat_statementsビュー

```
postgres=# SELECT query, wal_records, wal_fpi, wal_bytes FROM pg_stat_statements ;

query       | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
wal_records | 206328      # 本クエリによって、生成された合計WAL数
wal_fpi     | 3807        # 本クエリによって、生成された合計のfull page images数(※)
wal_bytes   | 43960467    # 本クエリによって、生成された合計WALバイト数
```

(※) 通常書き出されるWALと比較して、サイズが大きく、Disk書き込みの負荷が高いWALレコード。鈴木啓修さんのslideshare「PostgreSQLのリカバリ超入門(もしくはWAL、CHECKPOINT、オンラインバックアップの仕組み)」が詳しいので、ご興味のある方はご参考ください。

v14～に向けて、追加の提案している統計情報

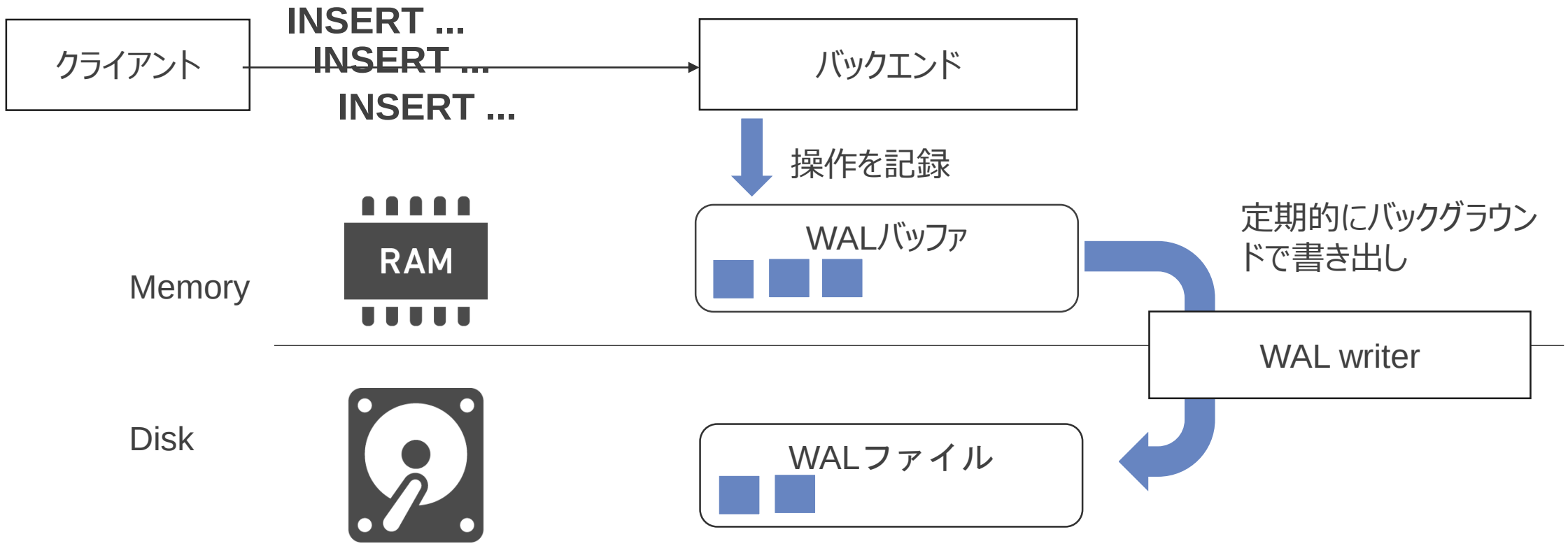
WALに関して、性能問題の予兆検知や解決をするためには、下記の統計情報も必要になると考えており、PostgreSQL本体に取り組むため、コミュニティに提案中

A. WALバッファが一杯になったときに同期書き込みが発生した回数（v14でリリース予定）
B. WALファイルを新規に作成した回数　（未定）
C. WALのDiskへの書き込み回数とレイテンシ　（未定）

A. WALバッファが一杯になったときに同期書き込みが発生した回数
～ WALバッファのDiskへの書き込み方法 ～

WALバッファ上のデータは、通常はWAL writerが定期的にDiskに書き出す

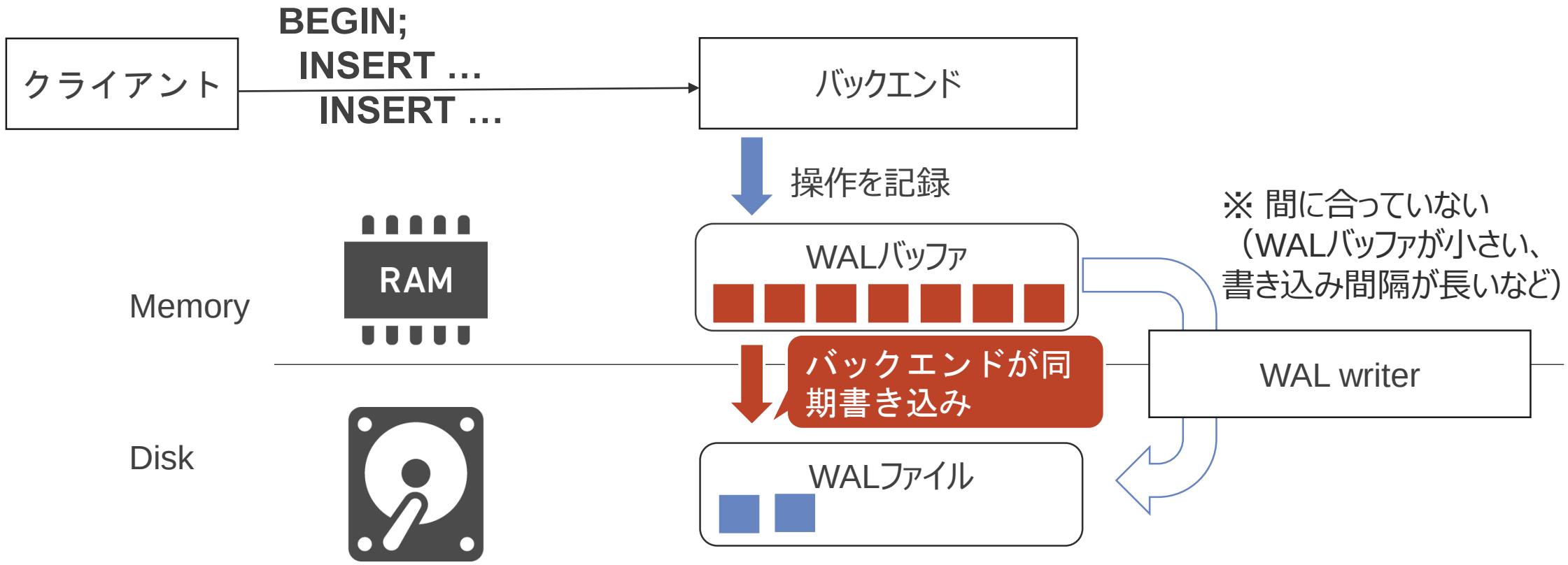
- バックエンドはメモリ上の操作のみなので高速
- ただし、WAL writerのDisk書き出しが間に合わない場合は…



A. WALバッファが一杯になったときに同期書き込みが発生した回数
～ WALバッファが一杯になったときの挙動 ～

WALバッファが一杯になっていると、バックエンドがWALバッファを書き出さざるを得ない

- Diskへの同期書き込みが発生するため、クエリ処理の性能劣化を生み出す



A. WALバッファが一杯になったときに同期書き込みが発生した回数 ～新たなviewの追加～

pg_stat_walビューを導入し、発生回数を情報提供する機能を追加（v14でリリース予定）

- 頻繁に発生していることが分かれば、WALバッファのサイズを増やすなどのチューニングにつなげることができる
- pg_stat_walビューは、現在2カラムのみ
  - クラスタ内で発生したWAL数、FPI数、WALバイト数についても追加予定
  - また、後述する情報も追加していく予定

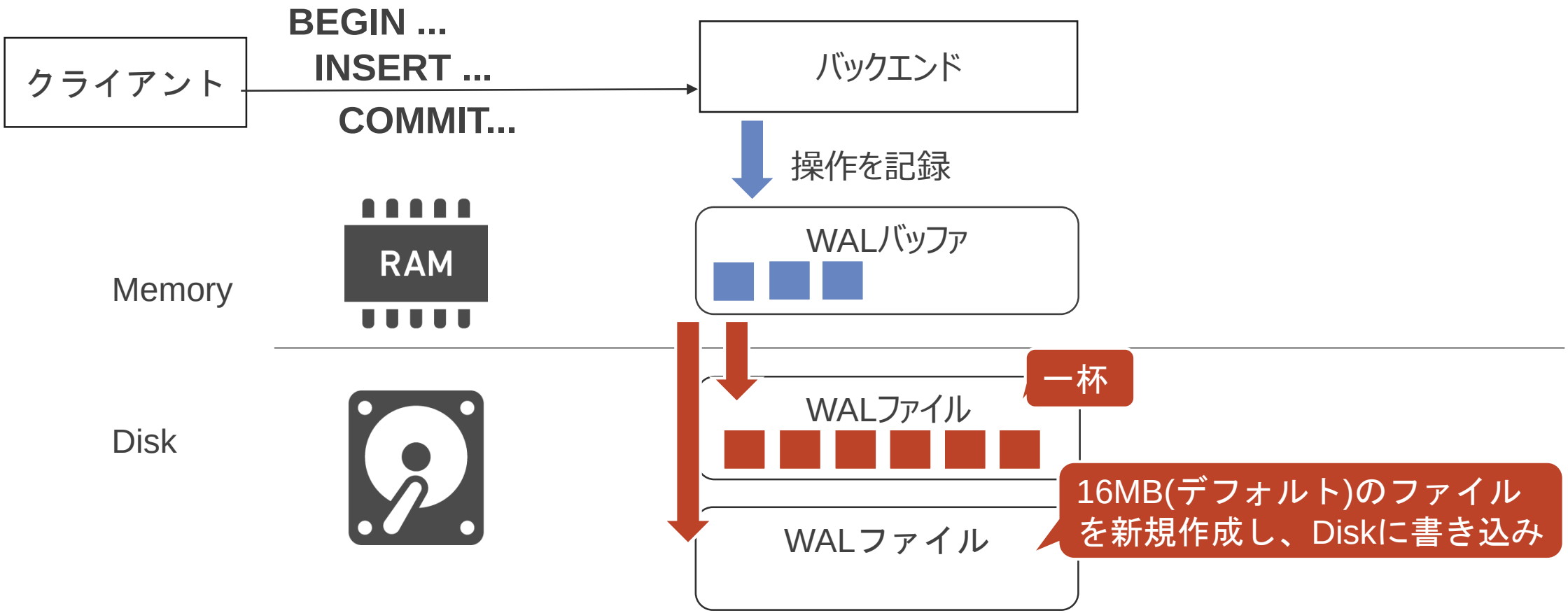
pg_stat_walビュー

```
wal_buffers_full | 15631      # WALバッファが一杯でDiskへの書き込みが発生した回数
stats_reset      | 2020-11-04 16:18:34.703078+09
```

B. WALファイルを新規に作成した回数

大量にWALを生成するワークロードの場合、Disk上に新規WALファイルを作成する処理が処理性能に影響する可能性がある

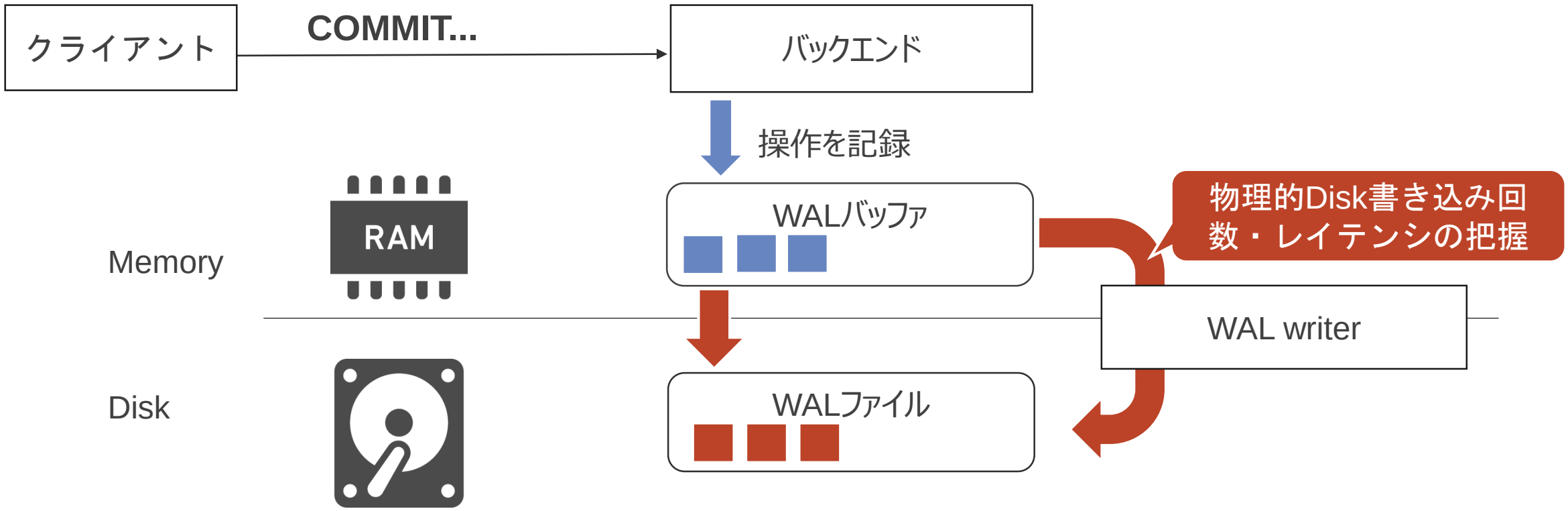
- 高頻度で発生している場合は、リサイクルされるWALファイル数を増やすなどの対応に繋げられる



C. WALのDiskへの書き込み回数とレイテンシ

WAL書き出しの基本的な統計情報の1つとして、性能に直結しやすい物理Diskへの書き込み状況を把握できるようにする必要があると考えている

- ワークロードの変化などの検知につなげられると考えている

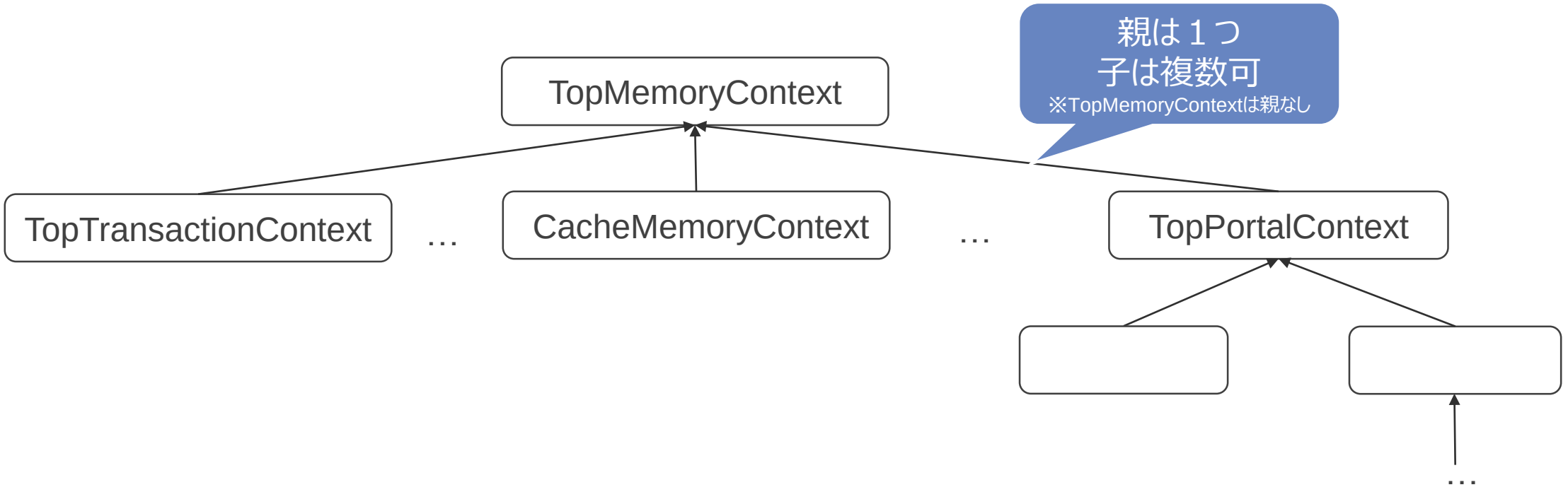


メモリ利用状況に関する統計情報
～MemoryContextの確認～

MemoryContextとは

PostgreSQLでは、MemoryContextと呼ばれる単位を使って、Tree構造でメモリを管理

- 用途に応じたMemoryContextを作成、MemoryContext内にメモリを割り当てる(palloc)

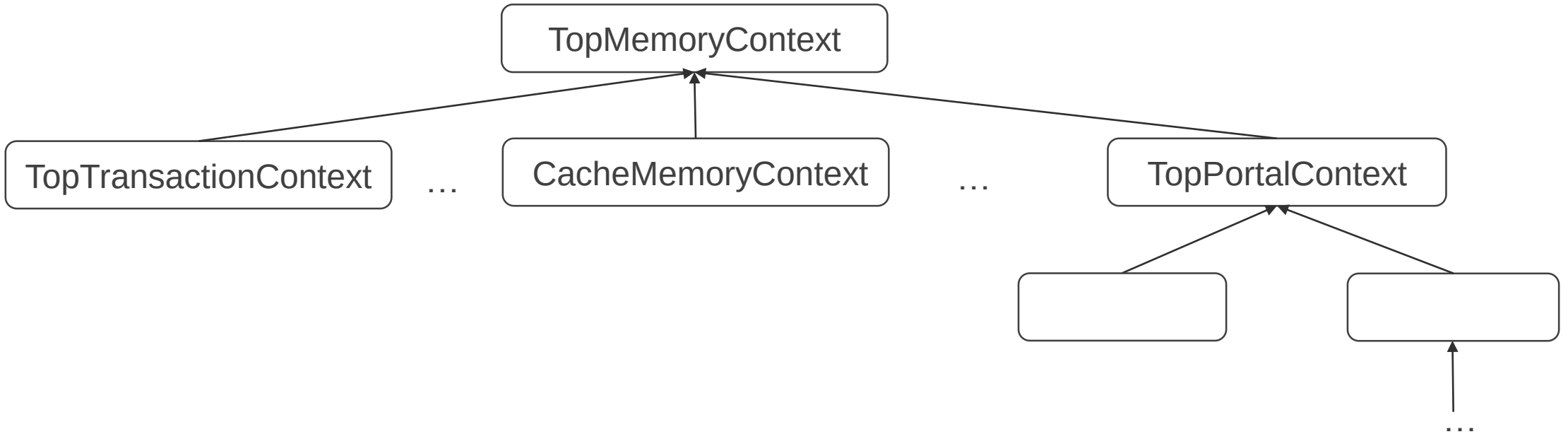


MemoryContextとは

PostgreSQLでは、MemoryContextと呼ばれる単位を使って、Tree構造でメモリを管理

- 用途に応じたMemoryContextを作成、MemoryContext内にメモリを割り当てる(palloc)

PREPARE文実行時の例

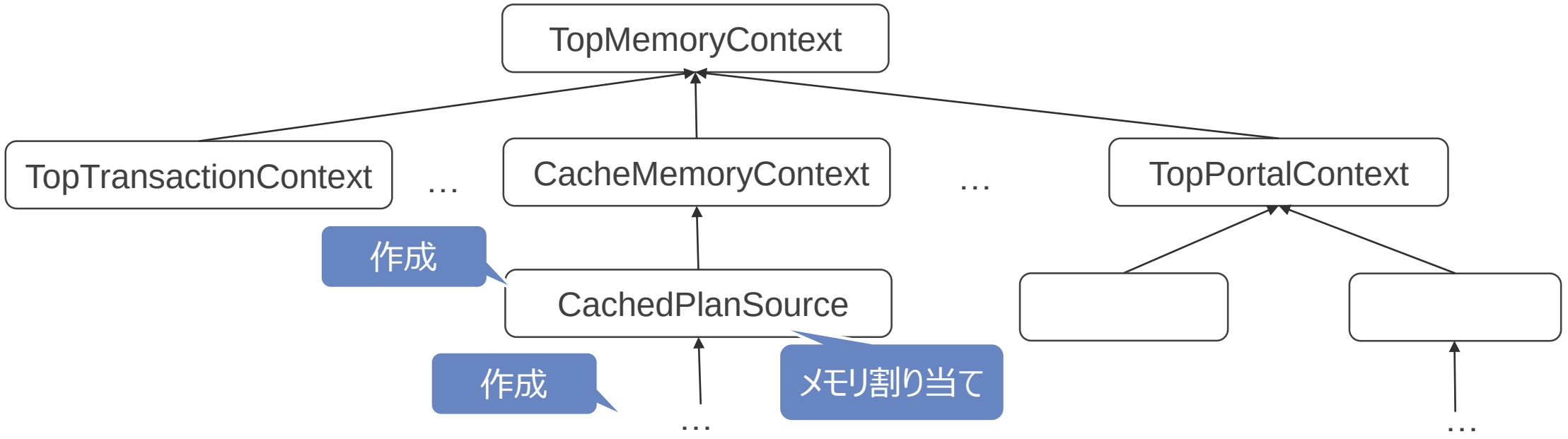


MemoryContextとは

PostgreSQLでは、MemoryContextと呼ばれる単位を使って、Tree構造でメモリを管理

- 用途に応じたMemoryContextを作成、MemoryContext内にメモリを割り当てる(palloc)

PREPARE文実行時の例

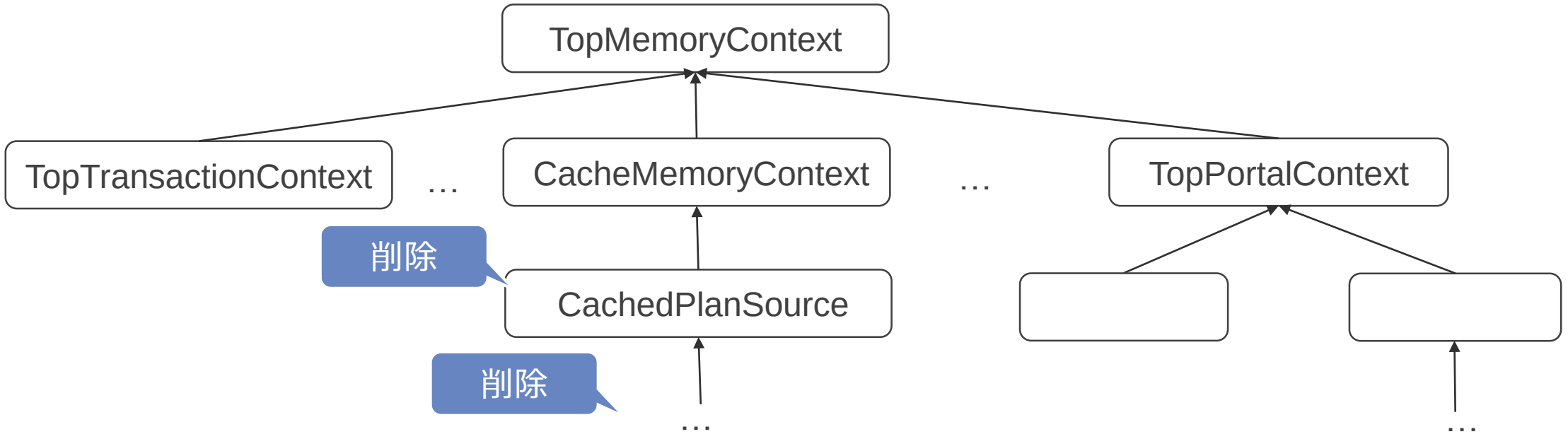


MemoryContextとは

PostgreSQLでは、MemoryContextと呼ばれる単位を使って、Tree構造でメモリを管理

- 用途に応じたMemoryContextを作成、MemoryContext内にメモリを割り当てる(palloc)
- 親のMemoryContextが削除されると、その子孫のMemoryContextも全て削除される

DEALLOC文実行時の例



MemoryContextの情報を取得する必要性

特定のContextが肥大化したり、大量のContextが作成されることで、メモリを逼迫することがある

例)

- 大量のPreparedStatement
- PostgreSQL内部のキャッシュ(relcacheなど)増加
- メモリリーク

OSからはPostgreSQLプロセスのメモリが肥大化していることはわかるが、PostgreSQLが何にメモリを利用しているかまではわからない

V13でのMemoryContextの確認

デバッグ用関数MemoryContextStats()で指定したContext以下の全Contextの状況を把握可能

- 使い方

  デバッガから呼び出す

```
(gdb) call MemoryContextStats(TopMemoryContext)
```

- 出力例

階層の深さはインデントで表現

```
TopMemoryContext: 68720 total in 5 blocks; 15592 free (17 chunks); 53128 used
  TopTransactionContext: 8192 total in 1 blocks; 7720 free (0 chunks); 472 used
  Prepared Queries: 16384 total in 2 blocks; 6616 free (4 chunks); 9768 used
  Operator lookup cache: 24576 total in 2 blocks; 10712 free (4 chunks); 13864 used
  Record information cache: 8192 total in 1 blocks; 1536 free (0 chunks); 6656 used
  RowDescriptionContext: 8192 total in 1 blocks; 6880 free (0 chunks); 1312 used
  MessageContext: 8192 total in 1 blocks; 4416 free (1 chunks); 3776 used
  Operator class cache: 8192 total in 1 blocks; 512 free (0 chunks); 7680 used
  smgr relation table: 16384 total in 2 blocks; 4544 free (3 chunks); 11840 used
  TransactionAbortContext: 32768 total in 1 blocks; 32504 free (0 chunks); 264 used
  Portal hash: 8192 total in 1 blocks; 512 free (0 chunks); 7680 used
  TopPortalContext: 8192 total in 1 blocks; 7648 free (0 chunks); 544 used
    PortalContext: 1024 total in 1 blocks; 480 free (0 chunks); 544 used: <unnamed>
  Relcache by OID: 16384 total in 2 blocks; 3424 free (3 chunks); 12960 used
  CacheMemoryContext: 524288 total in 7 blocks; 91744 free (0 chunks); 432544 used
    ...
Grand total: 1129888 bytes in 201 blocks; 287048 free (162 chunks); 842840 used
```

出力先はstderr
⇒ “logging_collector”がonの場合、ログファイルに出力

総計も出力

V13でのMemoryContextの確認

- 注意点

  MemoryContextStats()の出力コンテキスト数は100にハードコードされている

  もっと見たいときは、 MemoryContextStatsDetail()を出力行数を指定して実行

```
(gdb) call MemoryContextStatsDetail(TopMemoryContext, 500)
```

V13でのMemoryContextの確認(のつらみ)

- 商用環境で利用しづらい
  - デバッグシンボルがインストールされていること(ソースからビルドの場合configure --enable-debug rpmなどのパッケージの場合、デバッグ用パッケージが必要)
  - デバッガがインストールされていること
- 出力が集計しずらい、見にくい
  - 出力例(再掲)

```
TopMemoryContext: 68720 total in 5 blocks; 15592 free (17 chunks); 53128 used
  TopTransactionContext: 8192 total in 1 blocks; 7720 free (0 chunks); 472 used
  Prepared Queries: 16384 total in 2 blocks; 6616 free (4 chunks); 9768 used
  Operator lookup cache: 24576 total in 2 blocks; 10712 free (4 chunks); 13864 used
  Record information cache: 8192 total in 1 blocks; 1536 free (0 chunks); 6656 used
  RowDescriptionContext: 8192 total in 1 blocks; 6880 free (0 chunks); 1312 used
  MessageContext: 8192 total in 1 blocks; 4416 free (1 chunks); 3776 used
  Operator class cache: 8192 total in 1 blocks; 512 free (0 chunks); 7680 used
  smgr relation table: 16384 total in 2 blocks; 4544 free (3 chunks); 11840 used
  TransactionAbortContext: 32768 total in 1 blocks; 32504 free (0 chunks); 264 used
  Portal hash: 8192 total in 1 blocks; 512 free (0 chunks); 7680 used
  TopPortalContext: 8192 total in 1 blocks; 7648 free (0 chunks); 544 used
    PortalContext: 1024 total in 1 blocks; 480 free (0 chunks); 544 used: <unnamed>
  Relcache by OI                                                    ed
  CacheMemoryCon                                                  544 used
    ...
Grand total: 1129                                                   0 used
~
```

起動直後idle状態でも80行くらい省略している

V14でできるようになること(予定)

pg_backend_memory_contextsビューから、SQLで確認できる

- 内容はMemoryContextStatsDetail()と同等

```
postgres=# SELECT * FROM  pg_backend_memory_contexts LIMIT 15;
          name          |   ident   |      parent       | level | total_bytes | total_nblocks | free_bytes | free_chunks | used_bytes
------------------------+-----------+-------------------+-------+-------------+---------------+------------+-------------+------------
 TopMemoryContext       |           |                   |     0 |       68720 |             5 |      14832 |          24 |      53888
 TopTransactionContext  |           | TopMemoryContext  |     1 |        8192 |             1 |       7720 |           0 |        472
 Record information cache |         | TopMemoryContext  |     1 |        8192 |             1 |       1536 |           0 |       6656
 TableSpace cache       |           | TopMemoryContext  |     1 |        8192 |             1 |       2048 |           0 |       6144
 Type information cache |           | TopMemoryContext  |     1 |        8192 |             1 |       2584 |           0 |      22040
 Operator lookup cache  |           | TopMemoryContext  |     1 |       32768 |             3 |      10712 |           4 |      13864
 RowDescriptionContext  |           | TopMemoryContext  |     1 |        8192 |             1 |       6880 |           0 |       1312
 MessageContext         |           | TopMemoryContext  |     1 |       65536 |             4 |      24320 |           0 |      41216
 Operator class cache   |           | TopMemoryContext  |     1 |        8192 |             1 |        512 |           0 |       7680
 smgr relation table    |           | TopMemoryContext  |     1 |       32768 |             3 |      16768 |           8 |      16000
 TransactionAbortContext |          | TopMemoryContext  |     1 |       32768 |             1 |      32504 |           0 |        264
 Portal hash            |           | TopMemoryContext  |     1 |        8192 |             1 |        512 |           0 |       7680
 TopPortalContext       |           | TopMemoryContext  |     1 |        8192 |             1 |       7648 |           0 |        544
 PortalContext          | <unnamed> | PortalContext     |     2 |        1024 |             1 |        552 |           0 |        472
 ExecutorState          |           |                   |     3 |       65536 |             4 |      24376 |           4 |      39840
(15 rows)
```

階層の深さはlevelで表現

Contextを特定する情報があれば、identに出力される
例) PREPARE文を実行した場合

```
       name       |                                ident
------------------+-------------------------------------------------------
 CachedPlanSource | PREPARE q2(text) AS                                  +
                  |         SELECT datname, datistemplate, datallowconn+
                  |         FROM pg_database WHERE datname = $1;
```

V14でできるようになること(未定)

- ただし、pg_backend_memory_contextsビューは、実行したプロセスのContextしか見れない
- 任意のバックエンドプロセスのメモリコンテキストを確認する機能を提案中！

(参考)V13以前でSQLによるMemeoryContext確認

エクステンションpg_cheat_funcsを利用すれば、同様のことがV9.6～でも可能

- pg_stat_get_memory_context()
- というかほぼこの機能をPostgreSQL本体に移植
- “ident”相当の情報はない
- pg_cheat_funcsでは、ほかにもPostgreSQLで取り扱い可能な全UTF-８文字を返す関数(pg_all_utf8())、トランザクションIDを変更する関数など、いろいろな機能が提供されている

(※) https://github.com/MasaoFujii/pg_cheat_funcs

実行計画に関する統計情報
～汎用プラン・カスタムプランの生成状況の把握～

汎用プラン・カスタムプランについて

PREPARE文などPrepared Statementを利用した場合に生成されるプラン

- PREPARE文の例

```
=# PREPARE prep(INT) AS SELECT * FROM pgbench_accounts WHERE aid = $1;
```

- カスタムプラン: バインド変数の値を考慮して生成したプラン
- 汎用プラン: バインド変数の値を考慮せずに生成したプラン

汎用プランはプランを再利用するので、プラン作成にかかる時間を節約できる

一方で、バインド変数によっては不適切なプランになる

…

PostgreSQLが汎用・カスタムプランを選択する仕組みと影響

- パラメータがない場合、汎用プランを選択
- パラメータがある場合、デフォルト(※1)では、サーバ側(※2)は以下のようにプランを選択
  - 最初の5回は必ずカスタムプラン
  - 6回目移行はカスタムプランの平均コストと汎用プランのコストを比較し、小さい方を選択
  - 汎用プランのコストが小さくなった場合は、それ以降汎用プランが選ばれる
  - ただし、統計情報が更新されたり、クエリが操作するDBオブジェクトが変更された場合、汎用プランを再作成し、再度プラン選択をする

つまり、繰り返し実行しているクエリでも、プランがある時突然変わることもある

場合によっては、性能が大幅に劣化してしまうことも

※1 V12以降では、 plan_cache_modeにより、カスタム/汎用プランいずれかを強制することも可能

※2 JDBCドライバでは、まずJDBCドライバ側で最適化処理が動作する。デフォルトでは最初の5回はサーバ側でのPREPAREを実行しないので注意

…

V13での汎用・カスタムプランの確認

実行計画を確認する。

```
postgres=# PREPARE prep(INT) AS SELECT * FROM pgbench_accounts WHERE aid = $1;
```

- バインド変数が具体的な値 ⇒ カスタムプラン

```
postgres=# EXPLAIN EXECUTE prep(0);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97)
   Index Cond: (aid = 0)
(2 rows)
```

- バインド変数が$1など汎用的な表現 ⇒ 汎用プラン

```
postgres=# EXPLAIN EXECUTE prep(0);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97)
   Index Cond: (aid = $1)
(2 rows)
```

…

V13での汎用・カスタムプランの確認 (のつらみ)

- 基本的に都度EXPLAINを実行しないとわからない

- プランの確認が面倒

…

V14でできるようになること(予定)

- 汎用プラン・カスタムプランそれぞれの実行回数を記録したビューの提供

汎用プランが選択された回数

```
postgres=# SELECT * FROM pg_prepared_statements ;
 name |              statement                |         prepare_time          | parameter_types | from_sql | generic_plans | custom_plans
------+---------------------------------------+-------------------------------+-----------------+----------+---------------+--------------
 prep | PREPARE prep(INT) AS SELECT * FROM+| 2020-09-12 15:25:40.596691-04 | {integer}       | t        |             2 |            5
      | pgbench_accounts WHERE aid = $1;      |                               |                 |          |               |
(1 row)
```

カスタムプランが選択された回数

…

V14でできるようになること(未定)

- pg_prepared_statementsビューは、現在準備されているPrepared Statementしか確認できない。対象も現在のセッション文のみ
- pg_stat_statementsと連携して、全セッションの累積情報を記録することを提案中
    - ..汎用プラン・カスタムプランだけではなく、プランそのものの情報を保存するという議論もあるので、そちらに統合するかも。。

…

NTT DATA

Trusted Global Innovator