OSS-DB Gold
技術解説セミナー

db tech showcase 東京 2014

LPI-JAPAN
Linux Professional Institute Japan

2014/11/12
株式会社メトロシステムズ
花田 茂

自己紹介

- 氏名
  - 花田　茂（はなだ　しげる）
- 所属
  - 株式会社メトロシステムズ（東京・池袋・サンシャイン６０）
- 略歴
  - １９９９年：株式会社メトロシステムズに入社
  - ２００３年：オープンソースデータベースのＲ＆Ｄを担当
  - ２０１０年：PostgreSQLの開発に参加（主に外部データ連携）
  - ２０１３年：OSS-DB Gold取得
  - 現在：
    - OSS関連の構築やコンサルティング、トレーニング
    - PostgreSQL開発

主に外部テーブル/FDWまわり

本セミナーの流れ

- 運用管理の基本
  - PostgreSQLのアーキテクチャ
  - データ構造
- 性能監視
  - 基本的な監視項目
  - 便利な外部ツール
- パフォーマンスチューニング
  - パラメータチューニング
  - クエリチューニング
  - 便利な外部ツール

OSS-DB試験の紹介

- 試験の概要
- PostgreSQLの特徴

OSS-DB試験とは

- オープンソースデータベース（OSS-DB）に関する技術と知識を認定するＩＴ技術者認定です

データベースシステムの設計・導入・運用ができる技術者

大規模データベースシステムの
改善・運用管理・コンサルティングができる技術者

Linux技術者認定制度のLPICと同じく、
LPI-Japanが実施

OSS-DB Gold試験の出題範囲

- 運用管理（３０%）
  - データベースサーバ構築
  - 運用管理コマンド全般
  - データベースの構造
  - ホットスタンバイ運用
- 性能監視（３０%）
  - アクセス統計情報
  - テーブル/カラム統計情報
  - クエリ実行計画
  - スロークエリの検出
  - 付属ツールによる解析
- パフォーマンスチューニング（２０%）
  - 性能に関係するパラメータ
  - チューニングの実施
- 障害対応（２０%）
  - 起こりうる障害のパターン
  - 破損クラスタ復旧
  - ホットスタンバイ復旧

試験時間	：９０分※
問題数	：３０問
合格点	：７０点

※アンケート時間等を含む

OSS-DB試験の注意事項

- 基本となるPostgreSQLバージョンは「9.0」以降
  - 一部、新しい機能に関する問題も出題
  - ２０１４年11月時点の最新バージョンは「9.3.5」
  - 間もなく9.4がリリース！？
  - 最新の試験範囲はWebで確認！
    - http://www.oss-db.jp/outline/examarea.shtml

- OSに依存しない内容だが、表記はLinuxベース
  - シェルのコマンドプロンプトは「$」
  - ディレクトリ区切り文字は「￥」や「\」でなく「/」

運用管理の基本

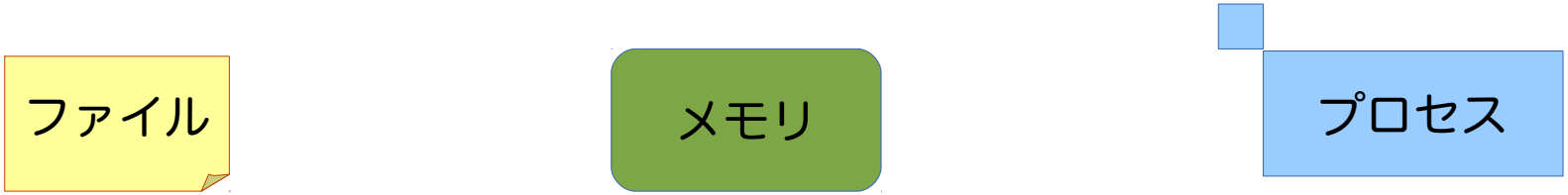
- PostgreSQLのアーキテクチャ
- PostgreSQLのデータ構造

PostgreSQLのアーキテクチャ（1）

- アーキテクチャにおける特徴
  - 多数のクライアントに高性能を提供
    - マルチプロセス構成（スレッド未使用）
    - 共有メモリによるデータ共有とバッファリング
    - 追記型によるMVCC
  - 賢くクエリを実行
    - コストベースオプティマイザ
    - 各種インデックス（B-Treeだけでなく GINやGiSTなど）
  - 任意のタイミングまでのリカバリ
    - WALによるリカバリ（クラッシュリカバリやPITR）
  - 柔軟な構成が可能
    - 同期・非同期を選べるレプリケーション
    - スタンバイからのオンラインバックアップ取得

PostgreSQLのアーキテクチャ（2）

PostgreSQLを構成する要素



PostgreSQLのアーキテクチャ（2）

PostgreSQLを構成する要素

ファイル

メモリ

プロセス

○設定ファイル
- ▲サーバ設定
- ▲クライアント認証設定

○データファイル
- ▲テーブル
  - ◇実データ
  - ◇空き領域マップ
  - ◇可視性マップ
- ▲インデックス

○ログファイル
- ▲サーバログ
- ▲トランザクションログ
  - ◇オンライン
  - ◇アーカイブ
- ▲コミットログ

PostgreSQLのアーキテクチャ（2）

PostgreSQLを構成する要素

ファイル

メモリ

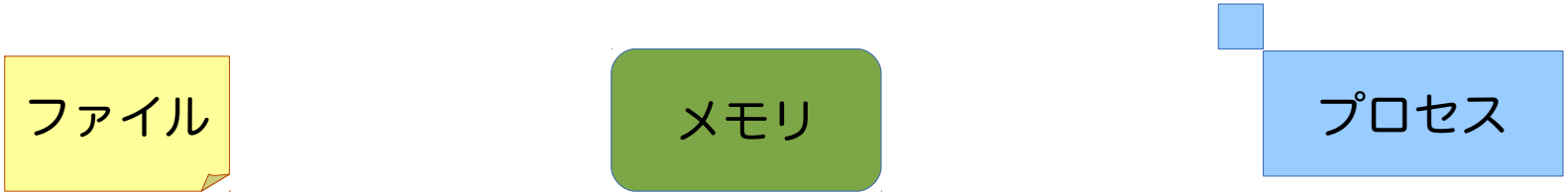
プロセス

○共有メモリ
- ▲セッション情報
- ▲プロセス情報
- ▲トランザクション情報
- ▲共有バッファ
- ▲WALバッファ

○ヒープメモリ
- ▲プロセスコード
- ▲スタック
- ▲ソート領域
- ▲一時バッファ

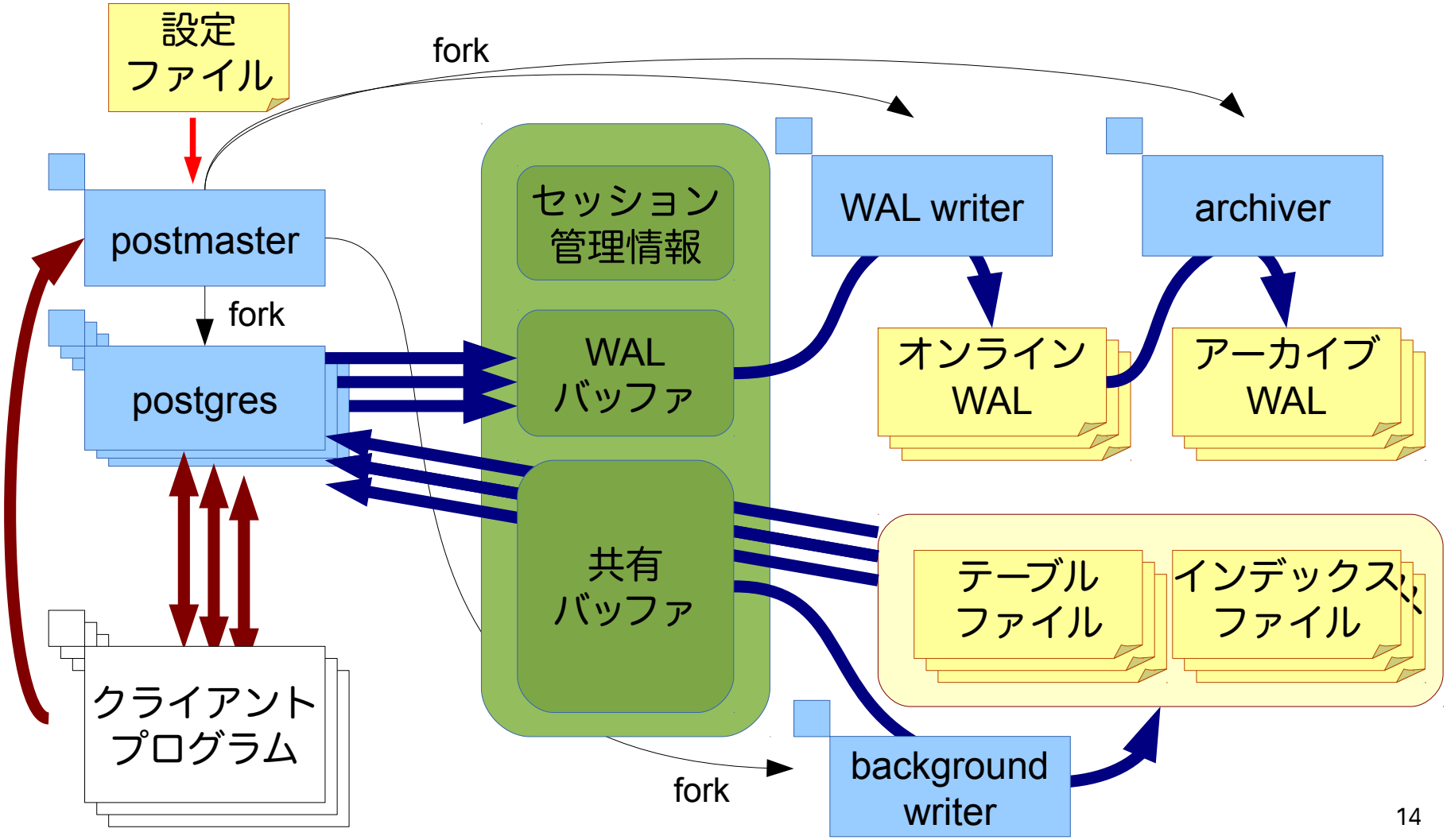
PostgreSQLのアーキテクチャ（2）

PostgreSQLを構成する要素



○postmaster	リスナ
○postgres	クエリ実行等、クライアント対応
○logger	サーバログ記録
○checkpointer	チェックポイント実行（9.2～）
○writer	ダーティバッファのフラッシュ
○wal writer	トランザクションログ書き込み
○autovacuum launcher	自動VACUUM（worker起動）
○autovacuum worker	自動VACUUM（実処理）
○archiver	トランザクションログアーカイブ
○stats collector	統計情報収集
○wal sender	レプリケーション（マスタ側）
○wal receiver	レプリケーション（スレーブ側）

PostgreSQLのクエリ処理



PostgreSQLのデータ構造（1）

- 階層構造
  - データベースクラスタ（$PGDATA）
    - デフォルトテーブル空間（base）
      - データベース
        - テーブル
        - インデックス
    - グローバルテーブル空間（global）
    - ユーザ定義テーブル空間（pg_tblspc）
    - 設定ファイル（*.conf）
    - トランザクションログ（pg_xlog）
    - コミットログ（pg_clog）
    - サーバログ（pg_log）
    - Etc.

PostgreSQLのデータ構造（2）

データベースクラスタの内容

```
% ls -l
-rw-------   1 postgres  postgres      4  7  3  2013 PG_VERSION
drwx------   7 postgres  postgres    238  4  1 13:57 base/
drwx------  42 postgres  postgres   1428  4  1 13:58 global/
drwx------   3 postgres  postgres    102  7  3  2013 pg_clog/
-rw-------   1 postgres  postgres   4219  7  3  2013 pg_hba.conf
-rw-------   1 postgres  postgres   1636  7  3  2013 pg_ident.conf
drwx------   6 postgres  postgres    204  4  1 13:40 pg_log/
drwx------   4 postgres  postgres    136  7  3  2013 pg_multixact/
drwx------   3 postgres  postgres    102  3 31 17:43 pg_notify/
drwx------   3 postgres  postgres    102  4  1 14:05 pg_stat_tmp/
drwx------   3 postgres  postgres    102  3 31 18:34 pg_subtrans/
drwx------   3 postgres  postgres    102  4  1 13:55 pg_tblspc/
drwx------   2 postgres  postgres     68  7  3  2013 pg_twophase/
drwx------  11 postgres  postgres    374  4  1 13:55 pg_xlog/
-rw-------   1 postgres  postgres  20337  3 31 17:45 postgresql.conf
-rw-------   1 postgres  postgres     36  3 31 17:43 postmaster.opts
-rw-------   1 postgres  postgres     50  3 31 17:43 postmaster.pid
```

PostgreSQLのデータ構造（３）

- テーブルファイルの構成
  - テーブルデータは以下の三種類の「フォーク」で管理
    - 実データ
    - FSM（Free Space Map:空き領域マップ）
    - VM（Visibility Map:可視性マップ）
  - ファイル名はpg_class.relfilenode（OID）で管理
    - 実データは<relfile_node>（例11716）
    - FSMは<relfilenode>_fsm（例:11716_fsm）
    - VMは<relfilenode>_vm（例:11716_vm）
    - oid2nameコマンドで名称取得可能
  - ファイル・ブロックで分割して管理
    - 8KB単位のブロックで管理
    - 1GB単位でファイルを分割
      - ２つ目以降のファイルには11716.1, 11716.2のように枝番がふられる

PostgreSQLのデータ構造（3）

テーブルファイルの構成

```
postgres=# SELECT relname, oid, relfilenode
postgres-# FROM pg_class
postgres-# WHERE relkind = 'r'
postgres-# AND relnamespace = (SELECT oid FROM pg_namespace WHERE nspname = 'public')
postgres-# ORDER BY relname;
     relname      |  oid  | relfilenode
------------------+-------+-------------
 pgbench_accounts | 16390 |       16438
 pgbench_branches | 16393 |       16393
 pgbench_history  | 16384 |       16384
 pgbench_tellers  | 16387 |       16387
(4 rows)
```

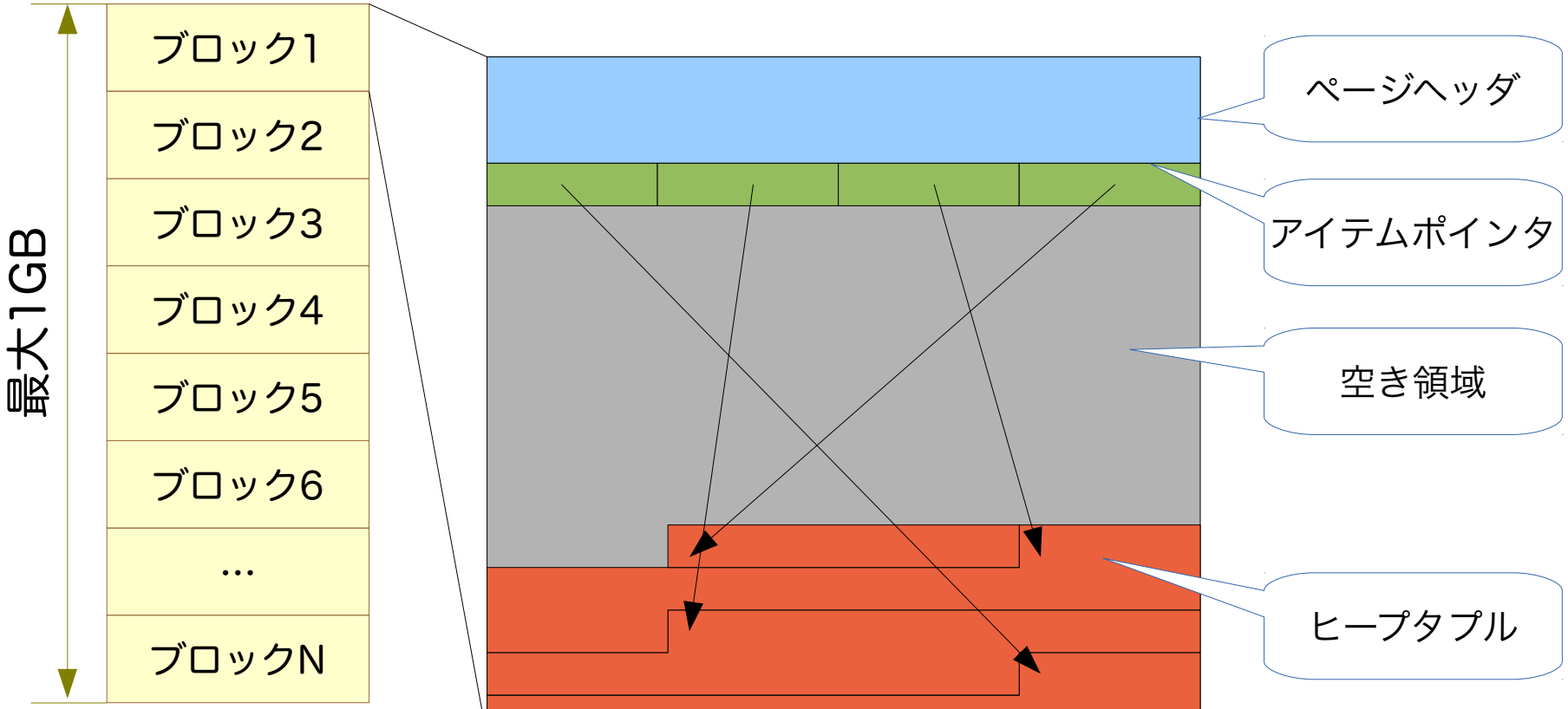
通常テーブルのみ

publicテーブルスペースにある（≒ユーザ定義の）もののみ

oidと同じとは限らない
→TRUNCATEやCLUSTERなどで変化

```
$ cd $PGDATA/base/12403
$ ls -l 16438*
-rw-------  1 hanada  staff  1073741824  6 12 16:25 16438
-rw-------  1 hanada  staff   269213696  6 12 16:25 16438.1
-rw-------  1 hanada  staff      352256  6 12 16:25 16438_fsm
-rw-------  1 hanada  staff       24576  6 12 16:25 16438_vm
```

テーブルファイルのブロック構成

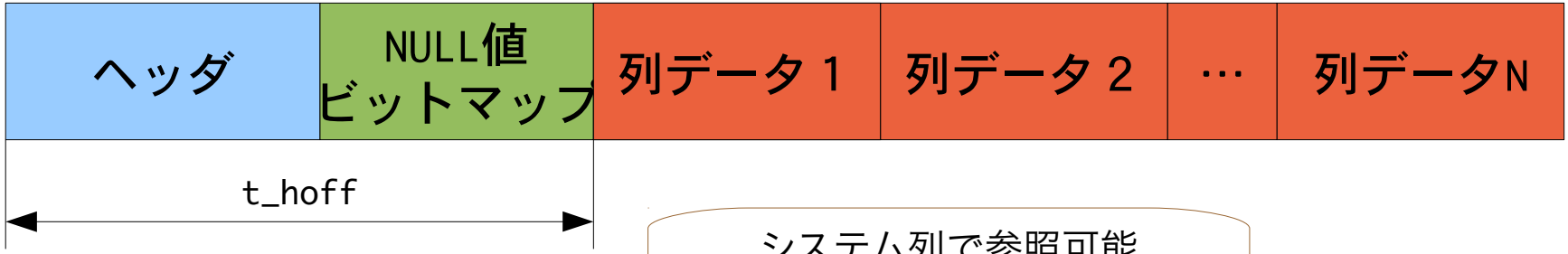


PostgreSQLのデータ構造（５）

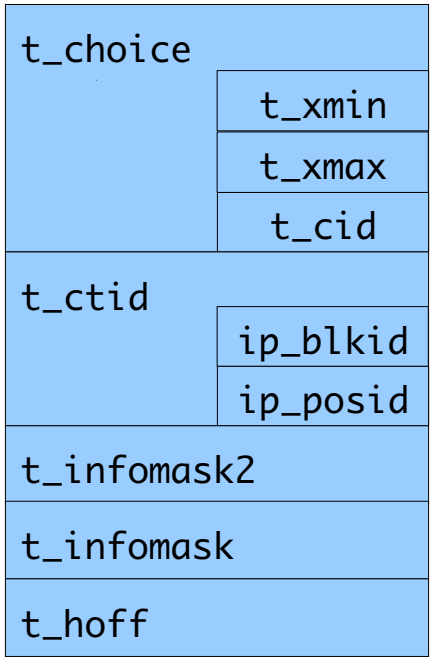
- ヒープタプル
  - 論理的なレコードを構成する物理的な要素
    - １レコード＝Nタプル（N=更新世代数）
  - PostgreSQLは、更新を削除＋挿入で実現
    - INSERT
      - 最新バージョンのタプルが追加される
    - UPDATE
      - 更新対象バージョンのタプルに削除フラグが立つ
      - 最新バージョンのタプルが追加される
    - DELETE
      - 削除対象バージョンのタプルに削除フラグが立つ

「追記型」と呼ばれる所以

タプルの構造



システム列で参照可能
xmin,xmax,cmax,cmin,ctid



可視性情報
- 作成トランザクションID
- 削除トランザクションID（未削除の場合は０）
- コマンドID

タプル位置
- ブロック番号
- アイテムポインタ番号

属性数+情報ビットマップ

情報ビットマップ（NULL値の有無、行ロック情報など）

列データオフセット

PostgreSQLのデータ構造（５）

MVCCに基づくタプルの参照

- トランザクションA(XID=10)
  - トランザクション開始
  - INSERT(ID=1、VAL=10)
  - COMMIT
- トランザクションC(XID=13)
  - UPDATE(ID=1、VAL=20)
  - SELECT→VAL=20
  - COMMIT

- タプルの状態
  - 1/10[xmin:10 xmax:0]
  - 1/10[xmin:10 xmax:13]
  - 1/20[xmin:13 xmax:0]

タプルは２つあるが
レコードは１件

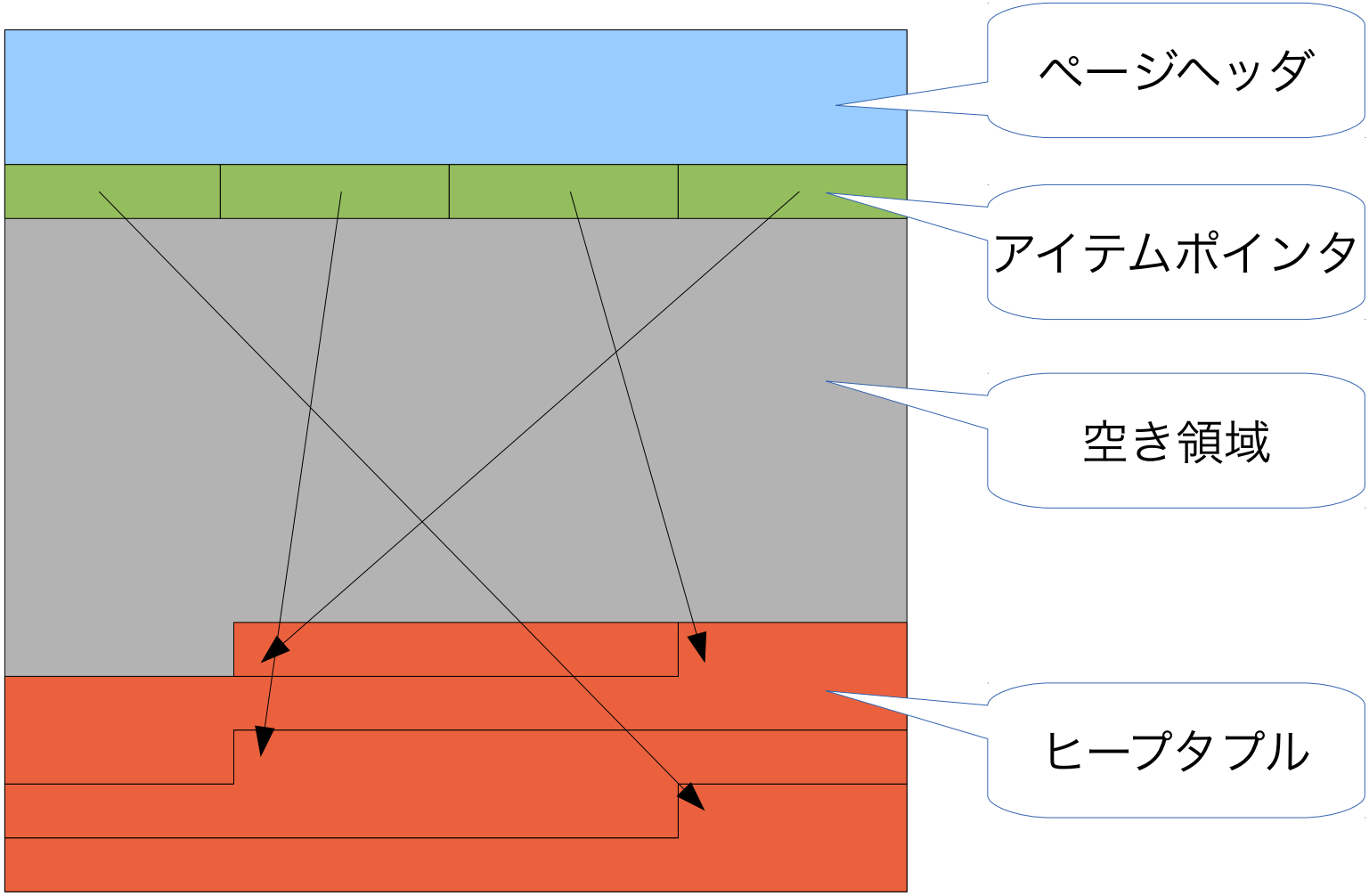
MVCCに基づくタプルの参照

トランザクションA(XID=10)	トランザクションB(XID=12)
トランザクション開始	
INSERT(ID=1、VAL=10)	トランザクション開始
COMMIT	
	SELECT→VAL=10
トランザクションC(XID=13)	
UPDATE(ID=1、VAL=20)	
SELECT→VAL=20	SELECT→VAL=10
COMMIT	
	SELECT→VAL=20

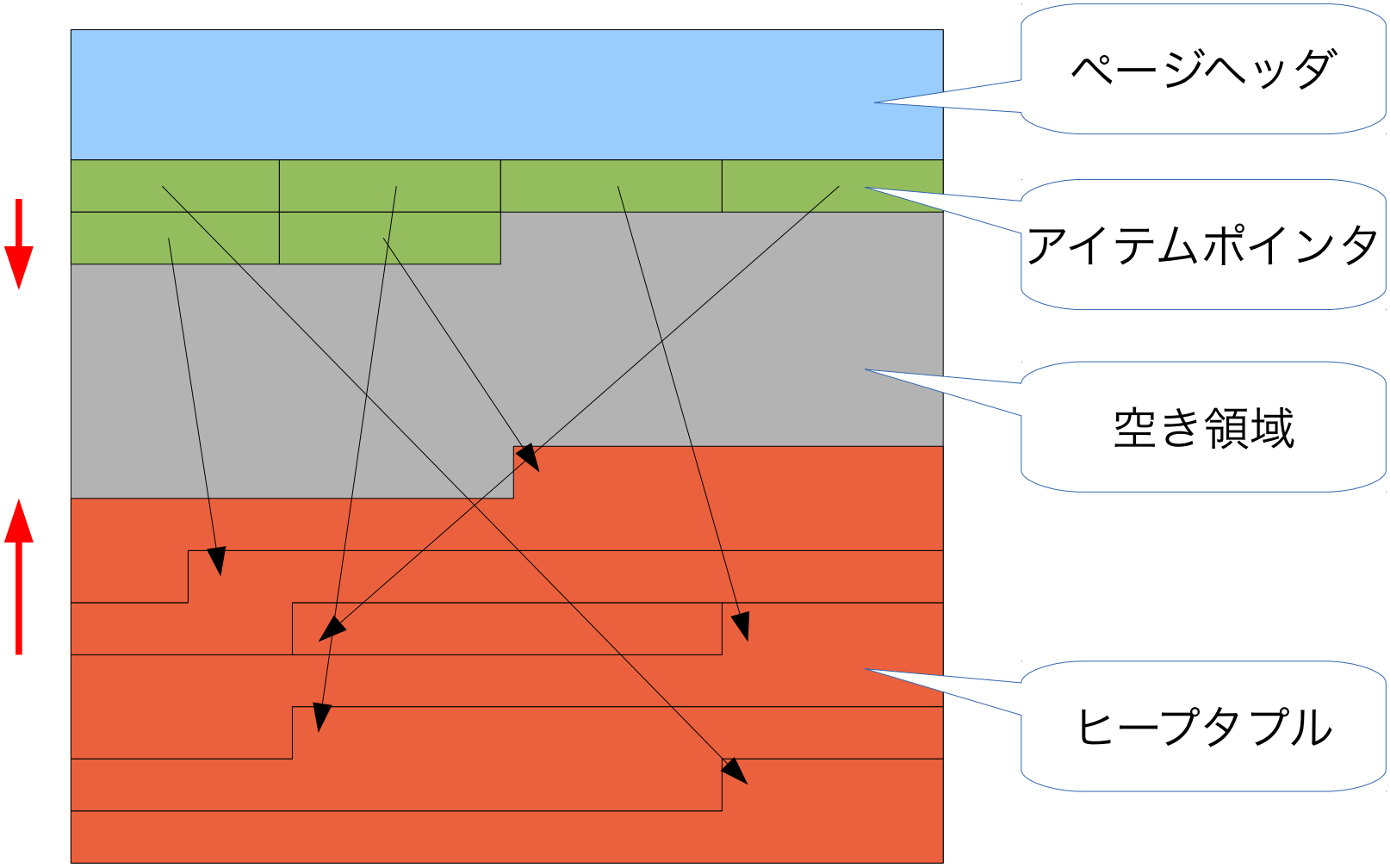
PostgreSQLのデータ構造（6）

- VACUUM
  - VACUUMは、不要領域を回収し断片化を解消する処理
  - 不要領域とは「削除・更新によりどのトランザクションからも参照されなくなった領域」のこと
  - VACUUMをしないとテーブルファイルが肥大化
- VACUUMとVACUUM FULL
  - この二つは全くの別物で、ほとんどのケースではVACUUM FULLは不要
    - VACUUM FULLは可能な限りタプルを詰めるので、更新が発生すると
  - それぞれのページに空き領域がある程度ある状態がPostgreSQLとしてはベストの状態

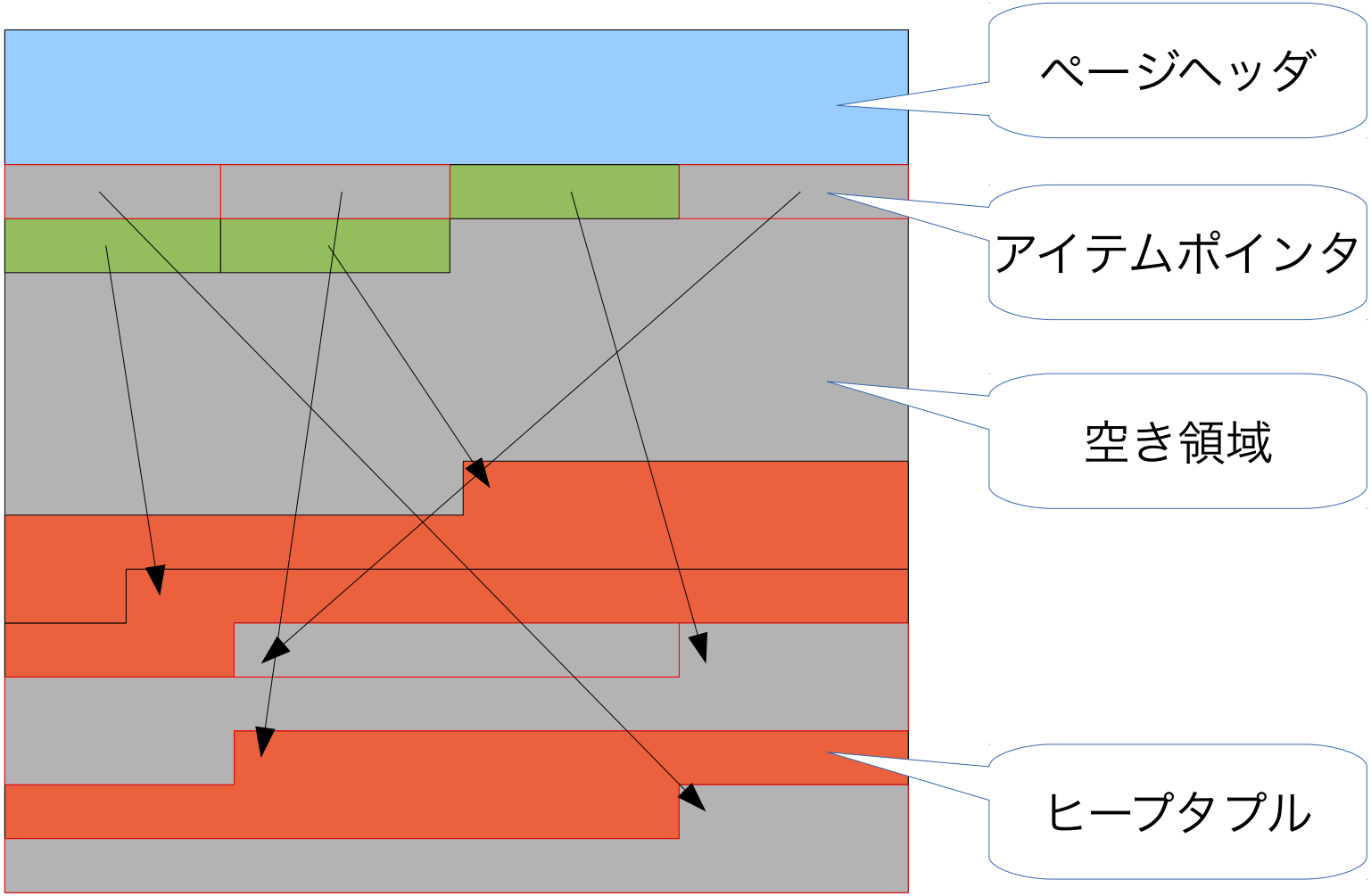
VACUUMの流れ（初期状態）



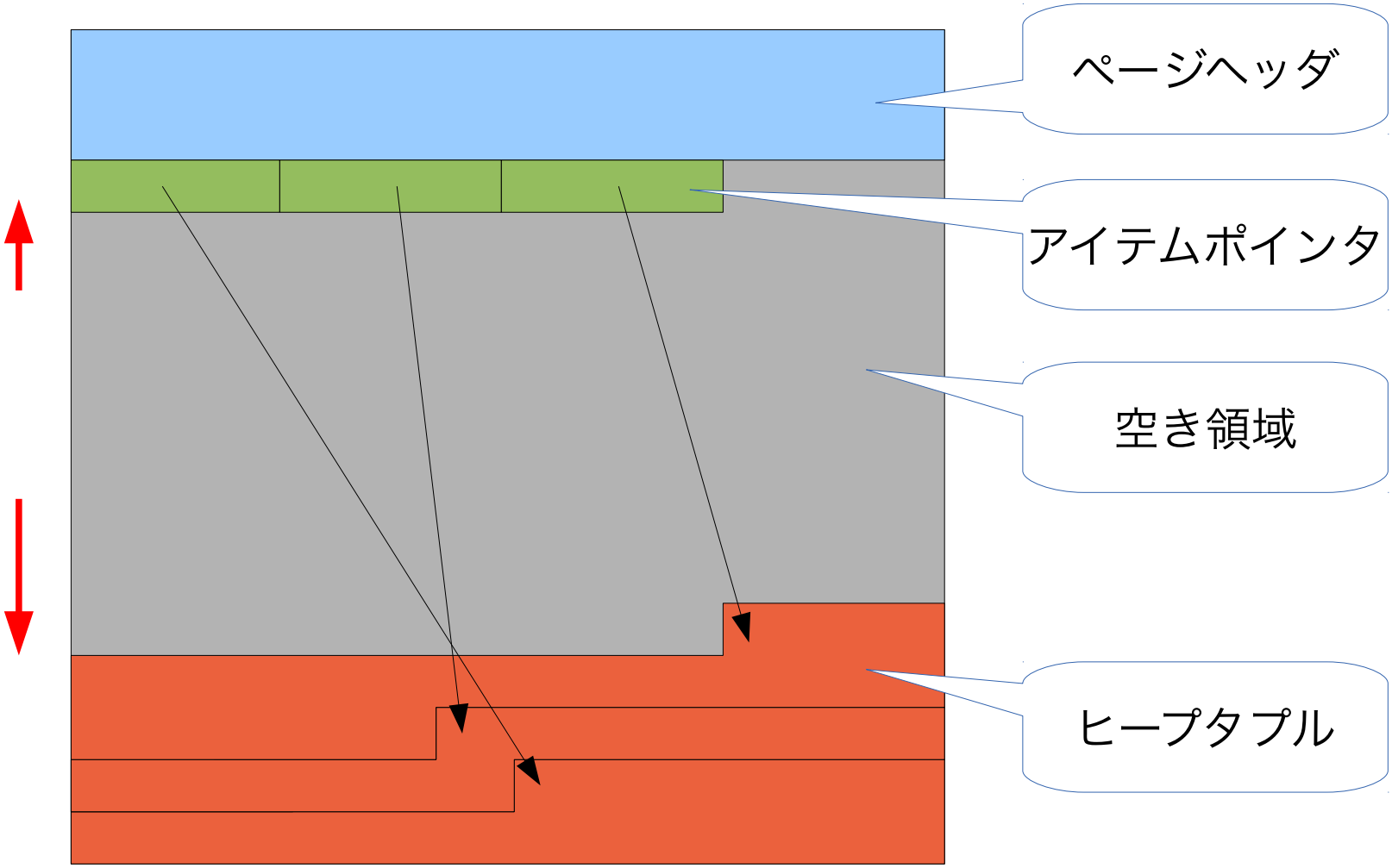
VACUUMの流れ（データ追加）



VACUUMの流れ（更新/削除による断片化）



VACUUMの流れ（VACUUM実施）



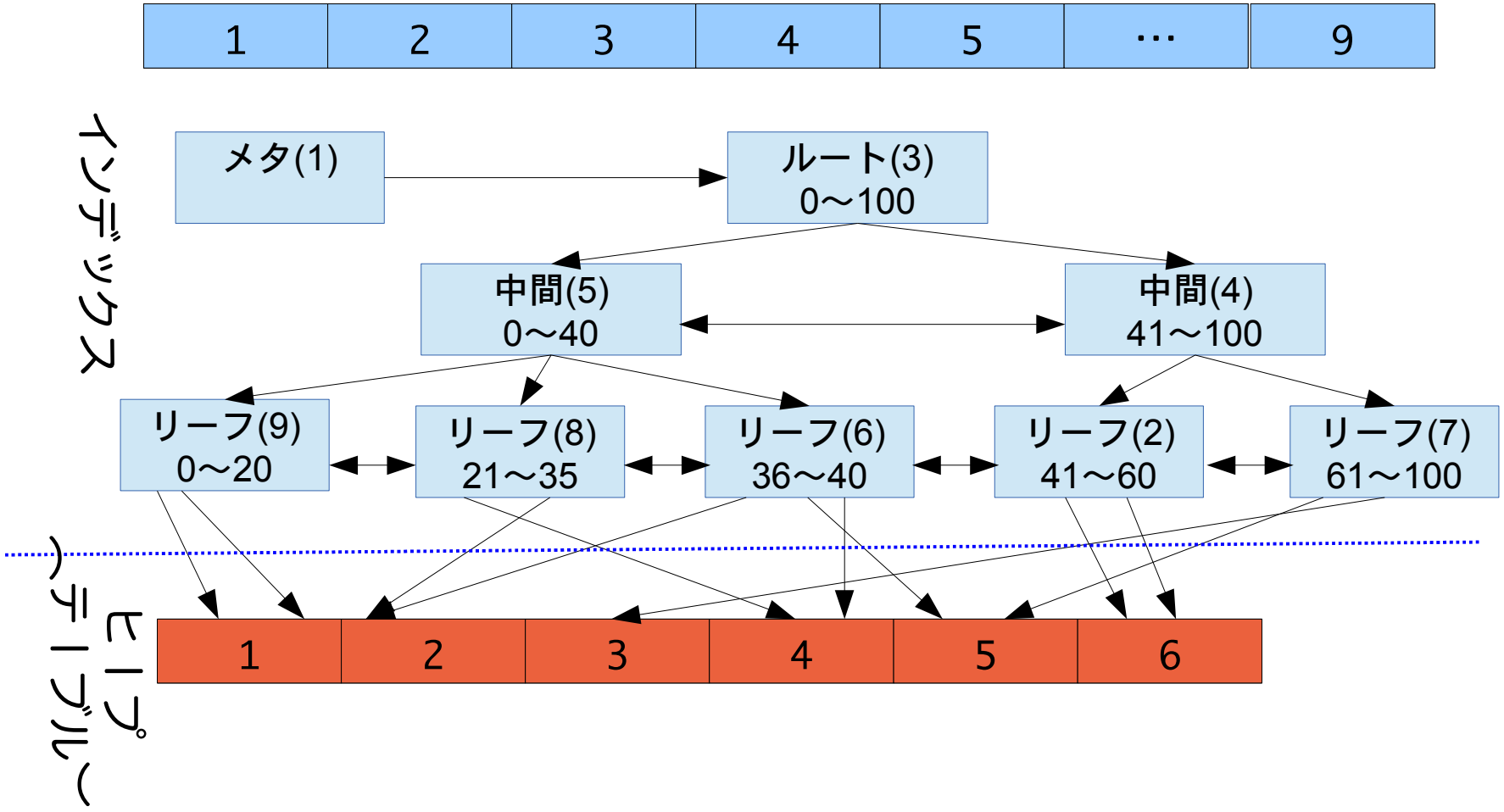
PostgreSQLのデータ構造（7）

- B-Treeインデックスのブロック構造
  - テーブルと同様に8KBブロックで管理
    - ブロック先頭にページヘッダ
    - ページヘッダの後ろにアイテムポインタ配列
    - ブロック末尾に管理情報（左右のページへのリンクなど）
    - 管理情報の前にインデックスタプル
  - テーブルと異なりFSMやVMは存在しない

PostgreSQLのデータ構造（7）

- B-Treeインデックスのブロック構造
  - メタページ、ルートページ、中間ページ、リーフページからなる
    - メタページは、ルートページの位置等を保持
    - ルートページと中間ページは、下位のページの最小値・最大値とそのページ番号を保持
    - リーフページでは、キー値とタプル（レコードデータ）へのポインタ（ブロック番号とオフセット）のペアをインデックスタプルが保持

B-Treeインデックスのブロック構造



PostgreSQLのデータ構造（7）

- PostgreSQLのB-Treeの特徴
  - ヒープタプルごとにインデックスタプルができる
    - 更新を繰り返すとレコードが少なくてもインデックスが肥大化する
    - VACUUMではインデックスからも不要領域を回収
  - 可視性判断材料(xmin/xmax)はヒープタプルにしかない
    - 自トランザクションが参照すべきヒープタプルを見つけるために複数のインデックスタプルをスキャンする必要がある

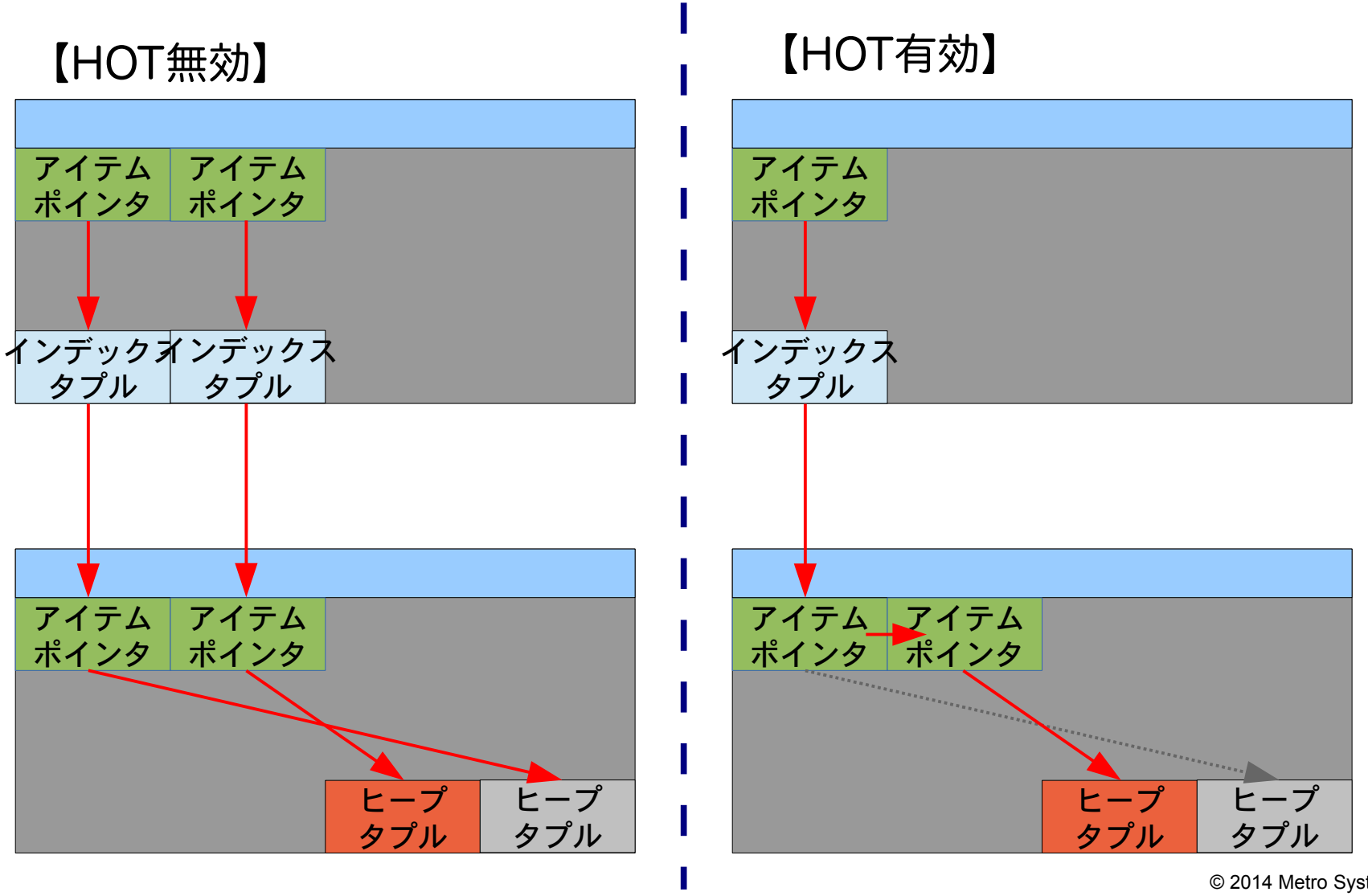
PostgreSQLのデータ構造（7）

- PostgreSQLのB-Treeの特徴
  - 更新を繰り返すとインデックスが汚れる
    - ツリー形状の不均衡やインデックスページ内の断片化
  - 解消にはREINDEX
    - REINDEXは排他ロックを取り（そのインデックスを使うものだけだが）検索もブロック！対策は…
      - 通常のインデックスは、CREATE INDEX CONCURRENTLYで作成し、旧インデックスを削除後に新インデックスをリネーム
      - 主キーインデックスは、CREATE INDEX CONCURRENTLYで作成し、主キー制約を作り直す（ALTER TABLE DROP/ADD CONSTRAINTはトランザクション内で！）

PostgreSQLのデータ構造（８）

- HOT(Heap Only Tuple)
  - 元タプルからリンクされたヒープタプルのみを追加し、インデックスタプルは追加しない
  - 更新対象列にインデックスがなく、同一ページ内に空き領域がある場合のみ可能
  - 参照すべきインデックスタプルが減るので参照も高速に

HOT(Heap Only Tuple)



PostgreSQLのデータ構造（9）

- FILLFACTORによる空き領域確保
  - INSERT時にFILLFACTOR%以上は埋めない
    - デフォルトは100%
    - ALTER TABLEで変更可能、ただし現在の空き領域が変わる訳ではない
  - ページ内に空き領域があると…
    - 更新時に同一ページに新バージョンタプルを配置しやすい→HOTになりやすい→インデックスが汚れにくい
    - 格納効率が下がるので、ディスク領域はより多く使う

性能監視

- 基本的な監視項目
- クエリチューニング
- 便利な外部ツール

基本的な監視項目（1）

- 障害やパフォーマンス劣化が起きてからでは遅い！
  - 運用開始前からどのような項目を監視するか決めておく
    - サービスレベル（目標）を事前に定義
    - 定常的な監視で許容範囲のうちに対策を！
  - 監視によるオーバーヘッドを見込んだサイジング
    - データベース単体で性能が出ても、いざという時に対策が打てないのでは実用にはならない！
    - アプリケーションやネットワークも含めて監視しましょう
  - 「監視するだけ」にならないように、フィードバックサイクルを作る
    - 「いつの間にか遅くなっていた」とならないように

基本的な監視項目（2）

- OSレベルの監視
  - CPU
    - SQLパース
    - プラン作成
    - フィルタリング
    - ソート
    - VACUUM
  - メモリ
    - ソート
    - ハッシュ結合
    - VACUUM
  - プロセス
    - 対象セッション
    - 自動VACUUM
  - ディスクI/O
    - テーブルスキャン
    - 一時ファイル
    - WAL書き込み
    - VACUUM
  - ディスク使用量
    - データベースサイズ
    - WALサイズ
    - アーカイブWALサイズ
    - 一時ファイルサイズ
  - ネットワークI/O
    - 結果データ転送
    - レプリケーション

基本的な監視項目（3）

- PostgreSQLレベルの監視
  - クエリのパフォーマンス
    - pg_stat_activityビュー
      - セッション数（接続/切断頻度なども）
      - クエリ所要時間
    - pg_stat_all_(tables|indexes)ビュー
      - テーブルやインデックスへのアクセス数や方式（全件/インデックス）
      - VACUUM/ANALYZE状況
    - pg_statio_all_(tables|indexes)ビュー
      - テーブルやインデックスのキャッシュヒット率
    - pg_xlog_location()/pg_xlog_insert_location()/pg_xlog_location_diff()関数
      - WAL書き込み量

基本的な監視項目（3）

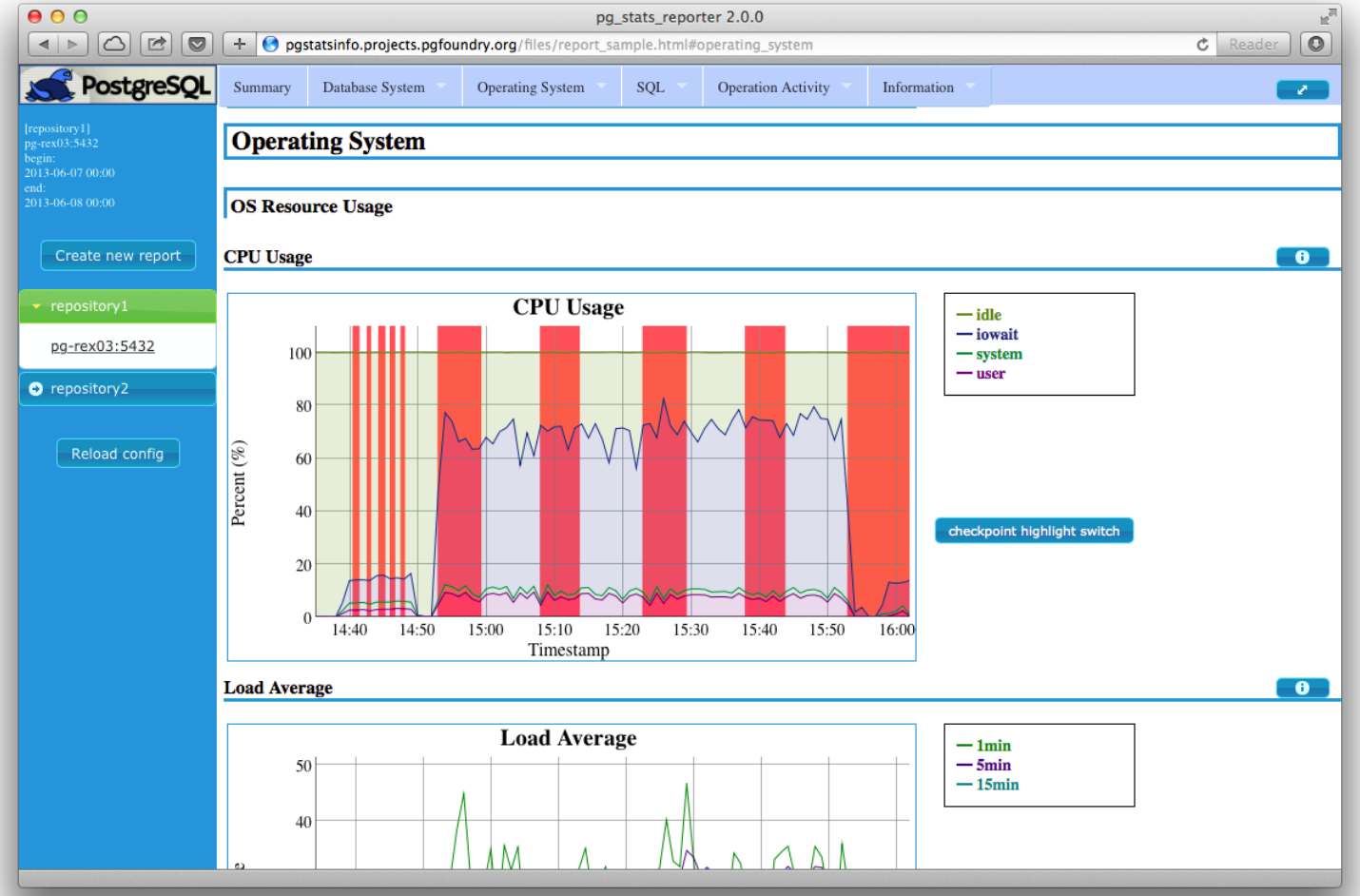
- PostgreSQLレベルの監視（続き）
  - その他の処理のパフォーマンス
    - サーバログ
      - チェックポイントや自動VACUUMの頻度・所要時間
      - WALアーカイブ
    - pg_stat_replicationビュー
    - レプリケーション遅延
  - ディスク領域
    - pg_(database|table|relation|indexes)_size()関数
      - データ領域（データベース/テーブル空間/テーブルやインデックス）
    - duコマンド
      - オンラインWAL領域・アーカイブWAL領域
  - サーバログ
    - ログ監視（FATAL/PANIC/ERRORが出ていないか？）

便利な外部ツール

- pg_statsinfo/pg_stat_reporter
  - 定期的に統計情報のスナプショットを取得しグラフィカルに表示
  - http://pgstatsinfo.projects.pgfoundry.org/index_ja.html
- Zabbix+pg_monz
  - pg_monzはZabbixでPostgreSQLを監視するためのテンプレート
  - http://pg-monz.github.io/pg_monz/
- Hinemos
  - 統合運用管理ソフト
  - http://www.hinemos.info

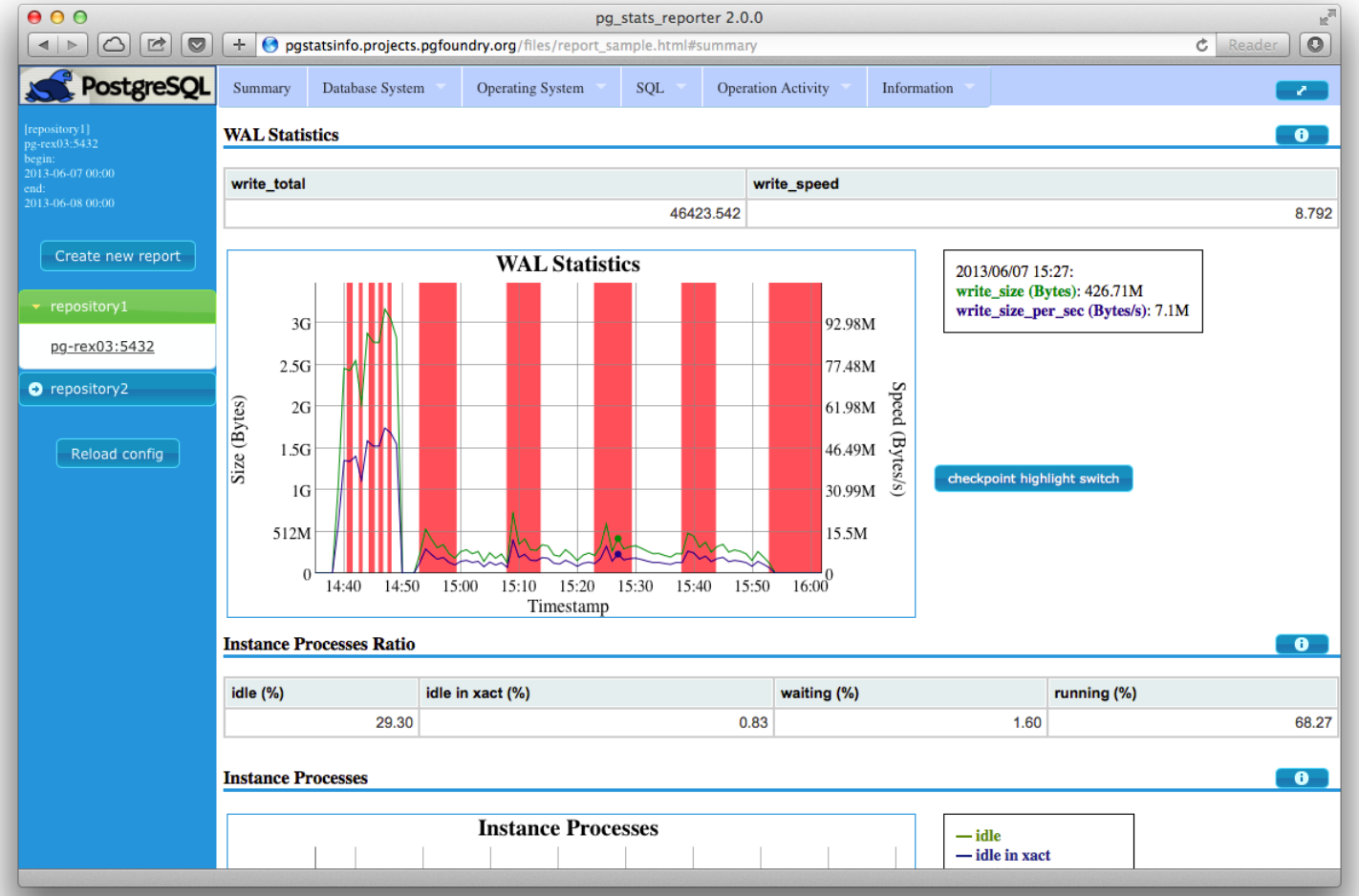
便利な外部ツール

- pg_statsinfo/pg_stat_reporterのサンプル画面
  - CPU使用率とチェックポイントの関係



便利な外部ツール

- pg_statsinfo/pg_stat_reporterのサンプル画面
  - WAL書き出し量とチェックポイントの関係



パフォーマンスチューニング

- パラメータチューニング
- クエリチューニング
- 便利な外部ツール

パフォーマンスの基本（1）

- パフォーマンスとは？
  - 業務モデルによって詳細は異なるが「単位時間でさばける処理量」
- パフォーマンスを決定する要因は様々
  - CPU処理
    - SQL解析、ソート処理、暗号化、Etc.
  - ディスクI/O処理
    - データファイル読み書き、WAL記録、VACUUM、ログ記録、Etc.
  - ロック競合
    - 同時アクセス、デッドロック、Etc.
  - ネットワーク転送
    - クエリ結果返却、レプリケーション、Etc.
- どこかの要因がボトルネックになり性能が決まる

パフォーマンスの基本（2）

- パフォーマンスをあげるには？
  - クエリ所要時間（レスポンス）を短縮する
    - I/O量を下げる
    - CPU処理を減らす
    - メモリを増やす
    - 検索対象データ量を減らす
    - 結果データ量を減らす
  - 並列度を上げる
    - ロックの強度を下げる
    - ロックの期間を短くする
    - CPUを増やす

パラメータチューニング（1）

- 構築したら変えましょう！のパラメータ
  - shared_buffers
    - テーブルやインデックスの内容をPostgreSQLがキャッシュする量
    - 実メモリの25%程度に設定
  - work_mem
    - ソート処理やハッシュ結合を高速化するが、プロセス単位の設定なのであまり大きくするとスワップする
    - バッチセッションやバッチユーザのみ別に設定するのも一案

```
postgres=# ALTER USER batch SET work_mem = '100MB';
ALTER ROLE
postgres=# \c - batch
You are now connected to database "postgres" as user "batch".
postgres=> show work_mem;
 work_mem
----------
 100MB
(1 row)
```

パラメータチューニング（2）

- 構築したら変えましょう！のパラメータ（つづき）
  - checkpoint_segments/checkpoint_timeout
    - 更新量が多いシステムではデフォルトの3/5minは小さいので、クラッシュリカバリの時間との兼ね合いで設定
    - 大きくするとpg_xlogディレクトリが肥大化する
    - 小さすぎると、このようなログが出ます

```
LOG:  checkpoints are occurring too frequently (2 seconds apart)
HINT:  Consider increasing the configuration parameter "checkpoint_segments".
```

パラメータチューニング（4）

- 様子を見て変えましょう！のパラメータ
  - random_page_cost
    - 値を小さくするとインデックススキャンが選択されやすくなる
    - SSDや高速なストレージを使用している場合は小さくしてみるとよいかも
  - effective_cache_size
    - OSのバッファキャッシュも含めたキャッシュサイズで、増やすとインデックススキャンになりやすい
    - 実メモリの25%～50%程度に設定

「速くする設定」も重要ですが、
「何が起きているか知るための設定」
の方が重要です！

パラメータチューニング（3）

- 構築したら変えましょう！のパラメータ（つづき）
  - wal_level
    - 運用要件上許される最小のレベルに
  - wal_buffers
    - デフォルトの「-1」だとshared_buffersの1/32を使用
    - 大きくしすぎるとコミット時の待ちが長くなる

クエリチューニング

- 実際のチューニングは…
  - 最近、画面の表示が遅くなってきた
  - 調べたらデータベースアクセスが遅いようだ
  - 一覧画面の表示に時間がかかっている
  - どのSQLが遅いんだろう？
  - 分からない…

EXPLAINの見方（1）

- EXPLAINとは
  - PostgreSQLがクエリをどのように実行するか/したかを知るためのツール
  - PostgreSQLは様々なクエリを以下のような「プランノード」をツリー上に組み合わせて実現
    - スキャン
      - Seq Scan/Index Scan/Index Only Scan/Bitmap Heap Scan/Etc.
    - 結合
      - Nested Loop/Merge Join/Hash Join
    - その他
      - Sort/Append/Aggregate/Limit/Etc.
  - 各プランノード毎に以下の情報を出力
    - プラン種別・推定行数・推定コスト・推定レコード長
    - 所要時間・ヒット行数・繰り返し回数（ANALYZE指定時のみ）

EXPLAINの見方（2）

実行計画の例

小さいテーブル（1ブロック）

```
postgres=# explain select * from pgbench_branches where bid < 3;
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on pgbench_branches  (cost=0.00..1.06 rows=2 width=364)
   Filter: (bid < 3)
(2 rows)
```

肥大化したテーブル（167ブロック）

```
postgres=# explain select * from pgbench_branches where bid < 3;
                                      QUERY PLAN
---------------------------------------------------------------------------------
-
 Bitmap Heap Scan on pgbench_branches  (cost=4.27..11.67 rows=2 width=364)
   Recheck Cond: (bid < 3)
   ->  Bitmap Index Scan on pgbench_branches_pkey  (cost=0.00..4.27 rows=2 width=0)
         Index Cond: (bid < 3)
(4 rows)
```

EXPLAINの見方（3）

- 実行計画ノードの例
  - スキャン
    - Seq Scan：先頭ブロックから順にヒープ全体をスキャン
      - 大きいテーブルでは致命的にディスクI/Oが出る
    - Index Scan：インデックスに基づいてヒープをスキャン
      - ランダムアクセスになる
      - 大量にヒットすると逆に遅い
    - Index Only Scan：インデックスのみを用いてスキャン
      - こまめにVACUUMをしないと選ばれない
    - Bitmap Index Scan：複数のインデックスから作成したビットマップに基づいてヒープをスキャン
      - ビットマップがメモリに収まれば高速

EXPLAINの見方（3）

- 実行計画ノードの例
  - 結合
    - Nest Loop：Outer 1 行につきInner全体をスキャンし結合
      - Inner側の件数が多いと極端に遅い
    - Merge Join：結合キーでソートされた結果同士を結合
      - 事前にソートする必要があるが、大きい結果同士でもある程度高速
    - Hash Join：Innerの結合キーでハッシュテーブルを作成し、それに基づいてOuterをスキャン
      - ハッシュテーブルがメモリに収まればかなり高速

EXPLAINの見方（3）

- 実行計画ノードの例
  - その他
    - Sort：結果を並べ替え
      - 結果が多いと一時ファイルを使い始める
      - 大量データのソートが必要なケースではwork_memを上げる
    - Limit：結果の行方向の部分集合を取得
      - 最終的に必要な件数が少なくて済む場合があり結果セットごとに性能が変わり易い
    - Materialize：結果セットを一時領域に保存
      - 結果セットがwork_mem以上なら一時ファイルに書き出す

EXPLAINの見方（４）

- EXPLAINでの注意点
  - 見積もり件数
    - 見積もり件数と実際の件数に乖離がある場合は、最適でない実行計画が選ばれがち
  - 統計情報は最新に
    - 定期的にANALYZEを実行して統計情報を更新
    - 自動VACUUMで十分か、確認を！
    - 大量更新バッチの後は手動ANALYZEを！
  - 動作中のアプリの実行計画はauto_explainで
    - contribモジュールのauto_explainを用いると、サーバログに実行されたSQLの実行計画を出力可能
    - 負荷が高くなるので、経過時間設定やon/offの切り替えなどでなるべく出力対象を限定的に

EXPLAINの見方（５）

- 詳しくは
  - 「生き残るデータベース管理者/アプリケーション開発者のための PostgreSQL SQLチューニング入門～Explaining Explain～」
    - OSC 2012 Tokyo/Springにて株式会社アシストの田中氏が講演
    - http://www.postgresql.jp/events/osc12tk_spring_folder

便利な外部ツール

- pg_dbms_stats
  - PostgreSQLの持つ統計情報を管理
    - 統計情報を固定して実行計画の変化を抑止
    - 本番環境の統計情報を検証環境に移植してチューニング
  - http://sourceforge.jp/projects/pgdbmsstats/
- pg_hint_plan
  - PostgreSQLでヒント句を利用可能にする
  - http://sourceforge.jp/projects/pghintplan/
- pgAdmin-III
  - 実行計画をグラフィカルに表示
  - クエリ書き換えのトライ＆エラーに
  - http://www.pgadmin.org

ディスクI/O分散

- ディスクI/OはRDBMSの最大の敵
  - PostgreSQLで発生するディスクI/O
    - データファイル（チェックポイント、VACUUM）
    - WAL（更新処理、VACUUM、CHECKPOINT）
    - バックアップ
  - ディスクI/Oボトルネックであれば、分散化で高速化
    - オンラインWALを分離
      - サーバ停止状態で$PGDATA/pg_xlogを別ボリュームに移動しシンボリックリンクでつなぐ
    - テーブルスペースを分離
      - アクセスの競合するテーブルを異なるテーブルスペースに配置
    - 注意点
      - バックアップ時に$PGDATA以外の場所を取り忘れないように！
      - pg_basebackupを使えば、全体を容易にバックアップ可能

参考資料

- PostgreSQL文書
  - 基本的には正しい情報はここから！
  - http://www.postgresql.jp/document/9.3/html/index.html
- PostgreSQL全機能バイブル
  - 非常に細かく内部構造や詳細動作が記述されています
  - 鈴木啓修・技術評論社
- Let's Postgres
  - 日本語のPostgreSQL技術情報ポータル
  - http://lets.postgresql.jp
- PostgreSQL Internals
  - 体系的にまとまっているのでアーキテクチャ概要などの理解に
  - http://www.postgresqlinternals.org/

最近の開発動向（1）

- 次のバージョンの9.4は…
  - 今秋リリースの見込み
    - 現在、ベータテスト中→ガンガン試してください！
  - 個人的には、GINインデックス高速化やJSONB（バイナリJSON）が熱いかと
  - 6/19に本イベントで講演された宗近さんの講演資料が詳しいです
    - http://www.slideshare.net/munetika/dbts-osaka2014-pg94

最近の開発動向（2）

- その次のバージョンの9.5は…
  - 6/15に開発サイクルが始まりました
    - 自律トランザクション
    - B-Treeインデックス構築高速化
    - 共有バッファのNUMA対応
    - 共有バッファのHibernation
    - 外部テーブルで継承や結合をサポート←宣伝

ご清聴ありがとうございました。

■お問い合わせ■
株式会社メトロシステムズ
花田　茂
Mail: hanada@metrosystems.co.jp
Twitter: @s87