

まずやっつく PostgreSQLのチューニング

日本PostgreSQLユーザ会 喜田 紘介

アシスト

「お客様の最高」のために

自己紹介

- 名前
 - 喜田 紘介(きだ こうすけ)
- 所属
 - 日本PostgreSQLユーザ会 広報・企画担当
 - 株式会社アシスト データベース技術本部
- 近況
 - 2014年度より、JPUGの理事になりました。
 - 仕事では、新規構築するシステムのDBをどうすべきか？というRDBMS選択支援や、商用DBからOSSへの移行の前段階として、オブジェクトやSQL差異のレクチャーや、データベースの診断・評価を行う 移行アセスメント支援 を主に担当しています。
 - この夏やりたいこと
PG9.4の検証、マラソンとトライアスロンの練習、歌とギターの練習(初心者)



本日も話すこと

- RDBMSの基本構造に沿いながら、マニュアルレベルでPostgreSQLのチューニングポイントを解説
- PostgreSQLでの実行計画の見方、SQLチューニングの方法を解説
- 最新バージョン9.4betaの話をし



DBチューニングとSQLチューニング

- DBチューニング

構築段階からある程度の設定が可能で、大まかな設定でも効果が得られる。

設定方法: パラメータチューニングが主

評価指標: OS情報(CPU、I/Oなど)

ベンチマークによるTPS測定

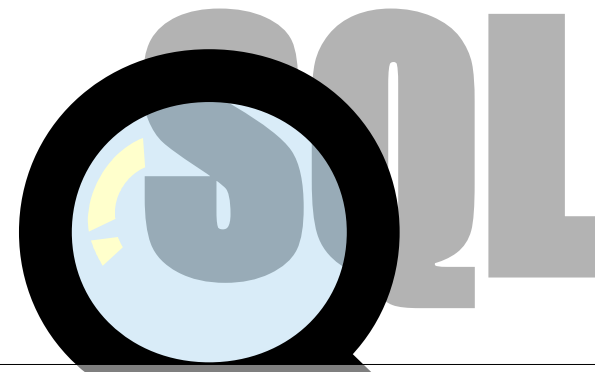
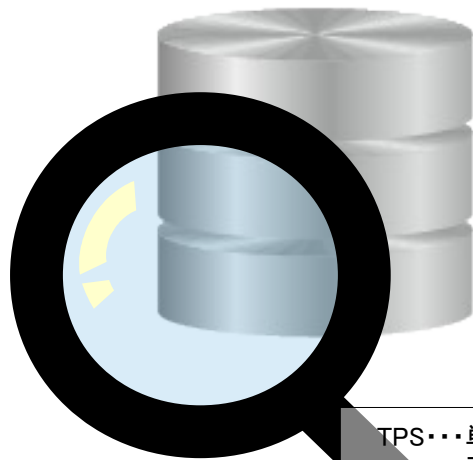
- SQLチューニング

実行時間が長いSQLを対象に、最適な実行計画をとることによる高速化を目指す。正しく行う事で非常に高い効果が期待できる。

設定方法: SQL修正、統計情報の調整

評価指標: 実行計画の確認

該当処理のTAT測定



TPS・・・単位時間あたりに実行されたトランザクション数。トランザクションの内容はベンチマークによって様々である。
設定変更前後で同じベンチマークを実行したTPSを評価指標とするなど、データベース性能を計測する上でしばしば用いる。
TAT・・・システムに処理要求を送ってから、結果の出力が終了するまでの時間。
実行時間が長いSQLではTATを目標時間内に抑えるようチューニングを実施する (参考: IT用語辞典 e-Words)

データベースの チューニングポイント

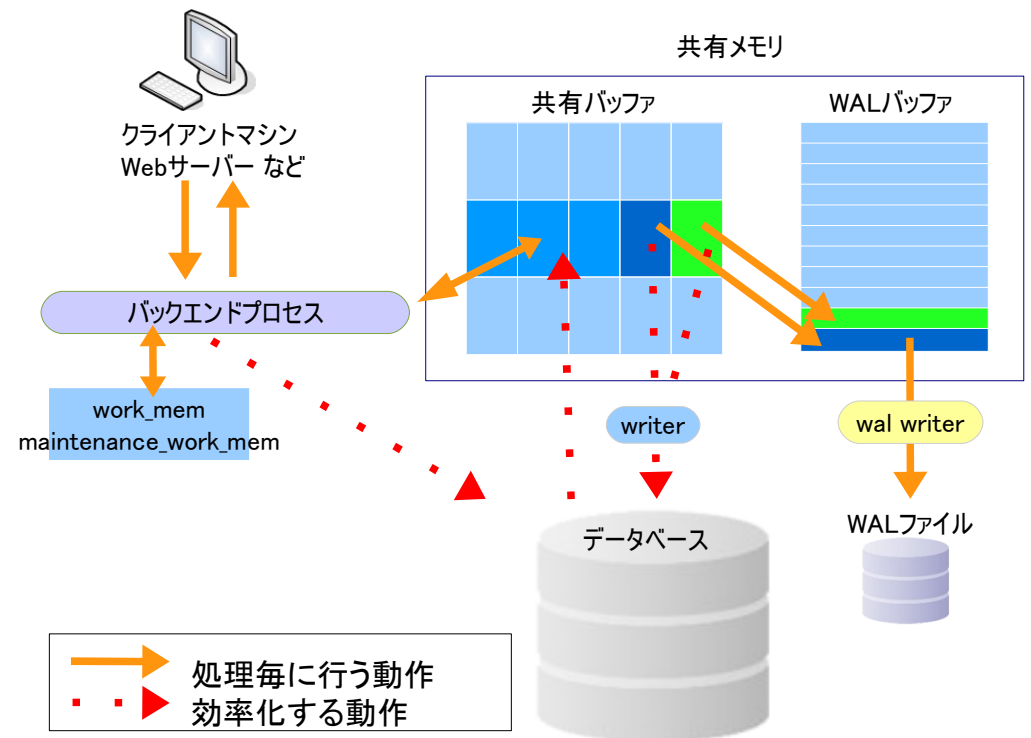


RDBMSのアーキテクチャ PostgreSQL

- データベースのアーキテクチャからチューニングポイントを考える

- メモリ
 - キャッシュヒット率
 - WAL生成タイミング
 - ソート等の一時領域
- ディスクI/O
 - チェックポイント間隔の調整
 - オブジェクトのメンテナンス
- プロセス
 - VACUUM関連
 - writerプロセス

SQL処理におけるPostgreSQLの動作



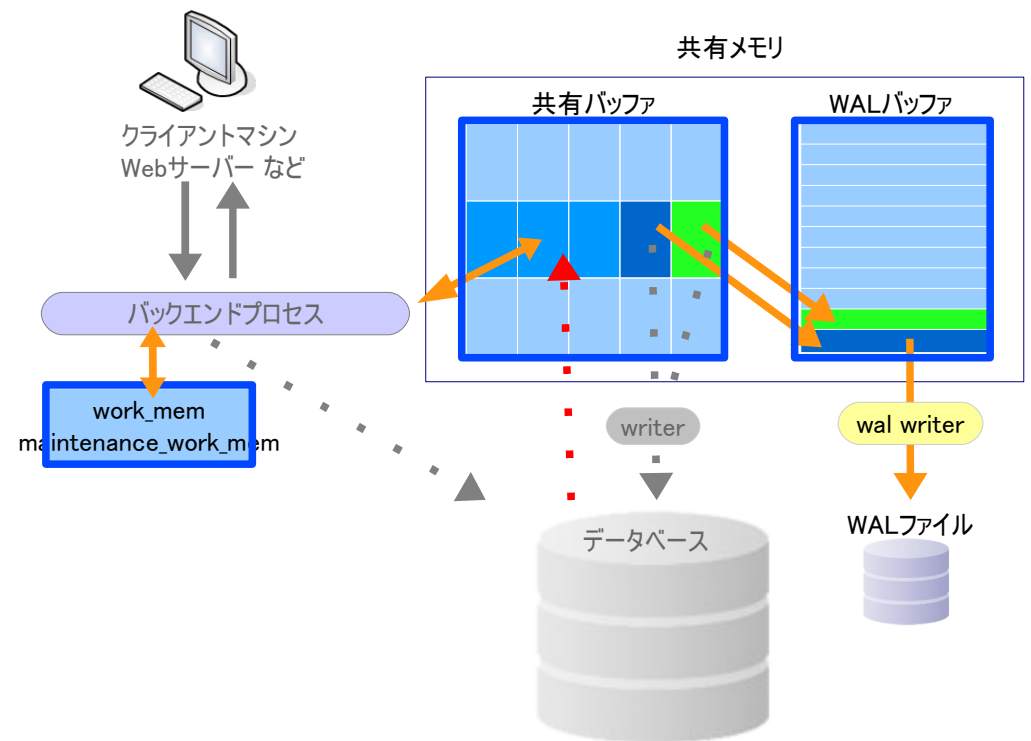
代表的なチューニングポイント(メモリ)

- オンキャッシュで効率よく処理できる状態を目指す
 - 各メモリ領域の割り当てが不適切だと、ディスクI/Oが頻発
 - 最適化は難しいが、大まかな設定でも大きな効果が得られる

- 考えることは 3つ

- ✓ キャッシュヒット率が適切か
- ✓ WAL生成のタイミング
(ただし最近は自動調整で問題なし)
- ✓ ディスクソートの有無を把握する
(SQLチューニング寄り)

SQL処理におけるPostgreSQLの動作



代表的なチューニングポイント(ディスクI/O)

- 性能を引き出す上で一番の要となるポイント

- チェックポイントによる性能影響は非常に大きい
- オブジェクトに対しての適切なメンテナンスも大事

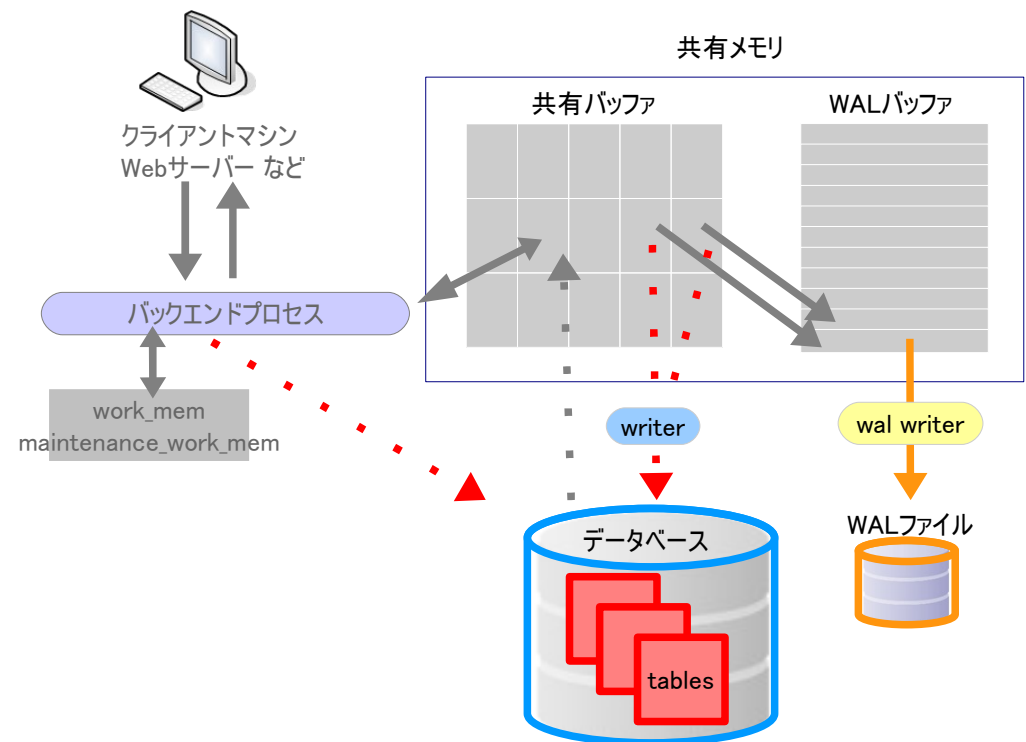
- チェックポイント間隔を適切にする

- ✓ WALとの関係を知っておくこと
- ✓ チェックポイント間隔を調整する

- オブジェクトのメンテナンス(本日は省略)

- ✓ テーブル(ページ)の余裕率を検討
- ✓ テーブルの再作成
- ✓ インデックスの再作成

SQL処理におけるPostgreSQLの動作



代表的なチューニングポイント(プロセス)

- 最適を求める場合、各プロセスの動作を細かく調整(製品固有のことが多い)

- PostgreSQL固有のVACUUM処理
- writerの動作も調整可能

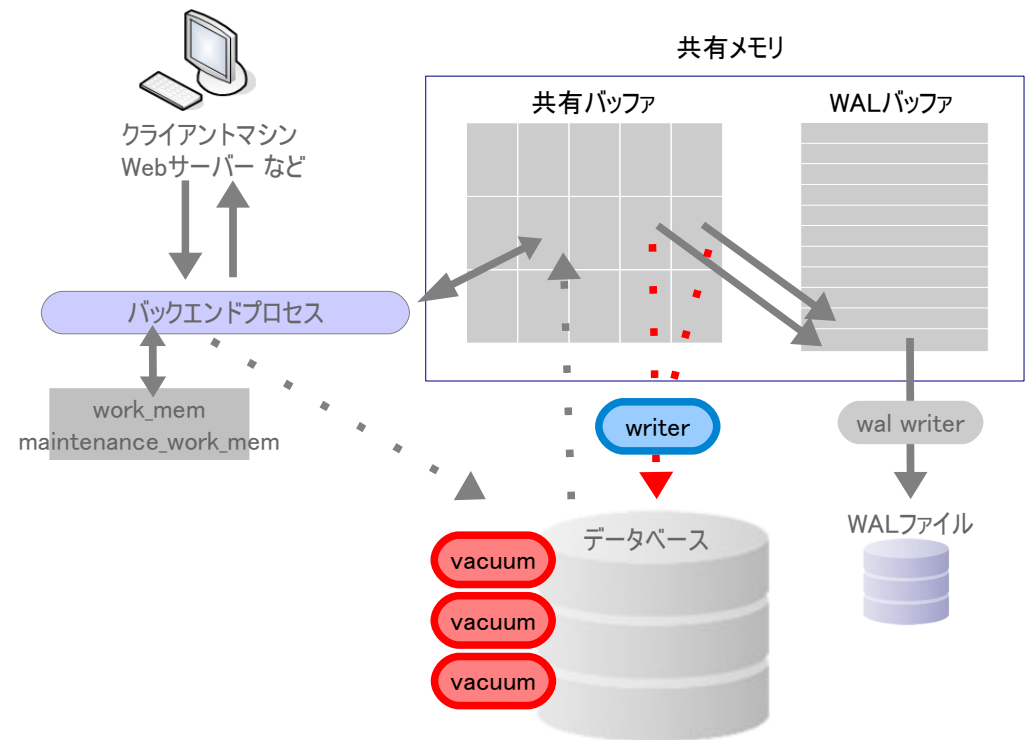
- VACUUM処理の最適化

- ✓ 自動VACUUMがきちんと動くこと
- ✓ VACUUM FULLは不要
- ✓ VACUUM/ANALYZEの閾値

- writerの動作(本日は省略)

- ✓ ダーティバッファの書き出しは常時
(ただし、ここまで調整することは稀)

SQL処理におけるPostgreSQLの動作



設定方法と確認方法の基本

- DBチューニングの基本はパラメータ設定
 - データベースクラスタ(\$PGDATA)配下のpostgresql.confを調整
- 反映されるタイミングを知っておく
 - パラメータ毎にDBの再起動/設定のリロード/即時など、反映されるタイミングが異なる
 - SQLで確認可能（公式のドキュメントでパラメーター一覧が存在しない）

```
/* pg_settingsビューを参照 */  
postgres=# \x  
postgres=# SELECT name, setting, unit, context FROM pg_settings;
```

```
SELECT distinct context FROM pg_settings;  
  
internal    ...変更不可(構築時設定確認用)  
postmaster  ...サーバ起動時  
sighup      ...設定ファイルの再読み込み  
backend     ...セッション確立時に決定  
superuser   ...スーパーユーザ権限で動的変更可能  
user        ...一般ユーザで動的変更可能
```

- 現状を確認するには統計情報ビューを参照
 - pg_stats* や pg_statsio* を参照することで、現状が適切かどうか判断できる
 - 場合によっては、動作させてログを見ることも必要

DBチューニングの実践



キャッシュヒット率を高く保つ

● アクセスデータ範囲を shared_buffers に収める

- postgresql.conf の shared_buffers は物理メモリの25%-40%程度とする
- OSのファイルキャッシュも使うため、厳密な調整でなくても良い
- 物理メモリが数十GBを超える場合や、Windowsの場合などに注意が必要

● キャッシュヒット率の確認

- SQLで累計のキャッシュヒット率を確認する

```
/* データベース単位でキャッシュヒット率を確認 */
postgres=# SELECT datname, ROUND(blks_hit*100/(blks_hit+blks_read), 2) AS cache_hit_ratio
FROM pg_stat_database WHERE blks_read > 0;
```

datname	cache_hit_ratio
template1	99.00
postgres	99.00

```
/* テーブル単位でキャッシュヒット率を確認 */
postgres=# SELECT relname,
ROUND(heap_blks_hit*100/(heap_blks_hit+heap_blks_read), 2) AS cache_hit_ratio
FROM pg_statio_user_tables
WHERE heap_blks_read > 0 ORDER BY cache_hit_ratio;
```

relname	cache_hit_ratio
emp	99.00
dept	99.00

● キャッシュヒット率を高めるための工夫

- 事前に表全体(またはWHERE句で絞って)SELECTし、ウォームアップ
- 索引やパーティションで必要な範囲のみにアクセスする

WAL生成のタイミングを知る

- WAL生成のタイミングを知り、wal_buffersパラメータを調整
 - postgresql.conf の wal_buffers は**最近のバージョンでは自動調整**される
 - 旧バージョンでは、デフォルトのWALファイルサイズの**16MB程度**にしてみる
 - WALがファイルに書かれるタイミングは以下
 - トランザクションがコミットされたとき
 - WALバッファが一杯になったとき ★これを減らしつつ効率的なのは、wal_buffers > WALセグメント(ファイル)サイズ
 - wal_writer_delay パラメータに指定された時間経過(デフォルト200ms)
 - 1トランザクションでの更新が多い場合、コミット以外のタイミングでの書き込み発生に注意
 - 同時接続数が多い場合も、コミット前にwal_buffersを超える可能性が高い
- 明確な確認方法は特に用意されていない
 - WAL生成量はファイル作成時刻などからある程度推測可能
 - しかし、WALがいつ書かれたかは確認する手段がない
 - Postgres Plusは待機イベントの確認が可能で、WAL生成が起因する場合は調整を検討

ディスクソートの発生を避ける

- メモリ上でソートが行えるようwork_memパラメータを調整

- work_memはセッションごとに確保される領域であるため、**SETコマンドで処理に応じて調整するのが望ましい**

```
postgres=# SET work_mem TO '256MB';
postgres=# SELECT ..... ORDER BY .....
```

```
/* バッチ処理などで大量のソートが発生
する場合SETコマンドでパラメータの変更を
行う */
```

- 適切なwork_memが割り当てられていると、実行計画の決定時により良いプランを選択

- ソート処理の詳細を確認

- trace_sortパラメータを有効にし、ログ出力から詳細を確認

```
LOG: internal sort ended, 330 KB used: CPU 0.00s/0.09u sec elapsed 0.15 sec
STATEMENT: SELECT l.logno,e.empname,l.status FROM log_master l,emp e
WHERE e.empno=l.empno;
LOG: external sort ended, 269 disk blocks used: CPU 0.03s/0.20u sec elapsed 0.30 sec
STATEMENT: SELECT l.logno,e.empname,l.status FROM log_master l,emp e
WHERE e.empno=l.empno;
```

←メモリ上でソートができています

←ディスクソートが発生しています

- EXPLAIN ANALYZEコマンドで確認

```
-> Sort (cost=15016.32..15266.32 rows=100000 width=11) (actual time=128.817.. (・・・略・・・)
    Sort Key: l.empno
    Sort Method: external sort Disk: 2152kB
-> Sort (cost=163.66..168.66 rows=2000 width=50) (actual time=1.600..10.439 rows=99951 loops=1)
    Sort Key: e.empno
    Sort Method: quicksort Memory: 330kB
```

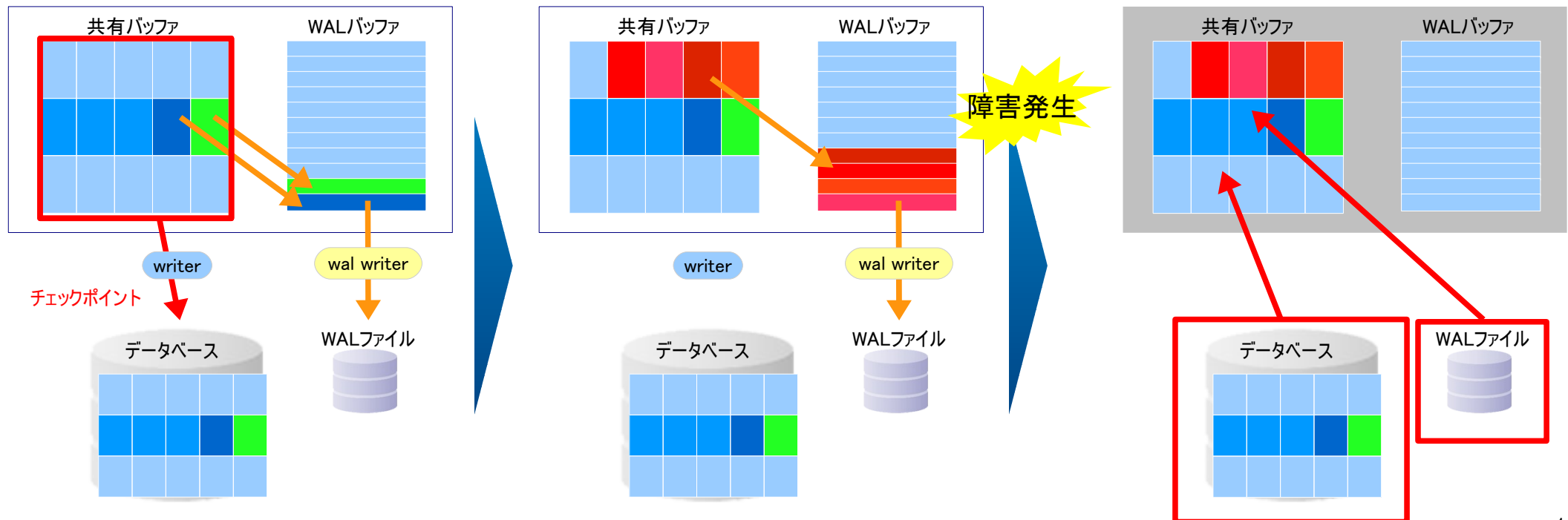
←ディスクソートが発生しています

←メモリ上でソートができています

チェックポイント間隔の調整

● チェックポイントの目的と目指すべき姿

- チェックポイントでは、キャッシュされたデータが漏れなくデータファイルに書き出される
 - クラッシュリカバリ時、チェックポイント以降のWALを適用すれば最新に戻せる
 - 大量のI/Oが発生するため、パフォーマンスの観点ではなるべく避けたい
 - パフォーマンス観点での理想は、「絶対クラッシュしないからチェックポイントもない」システム
- 現実的には、**クラッシュリカバリ時に許容できる時間をもとに**チェックポイント間隔を調整



チェックポイント間隔の調整

● チェックポイント間隔をパラメータで指定

- checkpoint_segmentsを**16個以上**としてみる
 - checkpoint_segmentsは、「WALファイルが何個溜まったらチェックポイントするか」
WALファイル1つで16MB × 16個 = 256MB毎にチェックポイント
- checkpoint_timeoutを**30分以上**としてみる
 - checkpoint_timeoutは、「checkpoint_segmentsの閾値に達しなくてもチェックポイントする時間」
更新が少ない時間帯でも、30分に一回はチェックポイントしておく
- チェックポイントが行われるタイミングは以下
 - checkpoint_segments パラメータで指定した数のWALファイルが生成されたとき
 - checkpoint_timeout パラメータで指定した時間が経過したとき
 - CHECKPOINTコマンドで明示的に実行
 - データベースの正常停止

● その他のチェックポイント関連パラメータ

- checkpoint_complation_targetで、チェックポイントを完了するまでの目標時間を設定
checkpoint_timeoutに対する割合で指定する
0.9程度に設定し、ゆっくりチェックポイントを実施させると、I/O量の安定化が期待できる

チェックポイント間隔の調整

- 実際のチェックポイント頻度を調査する
 - SQLでチェックポイントの回数を確認

```
postgres=# \x
postgres=# SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]-----+-----
checkpoints_timed    | 4120
checkpoints_req      | 0
buffers_checkpoint   | 229843
buffers_clean        | 57368
maxwritten_clean     | 0
buffers_backend      | 2334322
buffers_alloc        | 1145231
```

timeoutでチェックポイントした回数
更新量の閾を超えチェックポイントした回数

/* checkpoint_reqの値が大きい場合、更新が多く、チェックポイントが多発していることが考えられる */

- ログにチェックポイント情報出力する
 - log_checkpoint.....サーバログにチェックポイントの処理の詳細を記録
 - checkpoint_warning....指定時間より短い間隔でチェックポイント処理が事項された場合にログに警告を出力

VACUUM処理

- VACUUMに対する考え方

- 自動VACUUM推し
- VACUUMが適切に行われることで得られるメリット
 - 更新時に同一ページ内に空きがあることでHOTが効く
 - 加えて、HOTによる領域回収ができるため、次回VACUUM負荷が軽減される
 - テーブルファイルの肥大化を防ぎ、検索時のI/O量を適正にする
- (悪名高い)VACUUM FULL
 - 平常運用時に使う必要はない
 - ローカルディスクに余裕があるなら、テーブル再作成のメンテナンスとして
 - ANALYZEを別途実施しなければならない

- 注意点

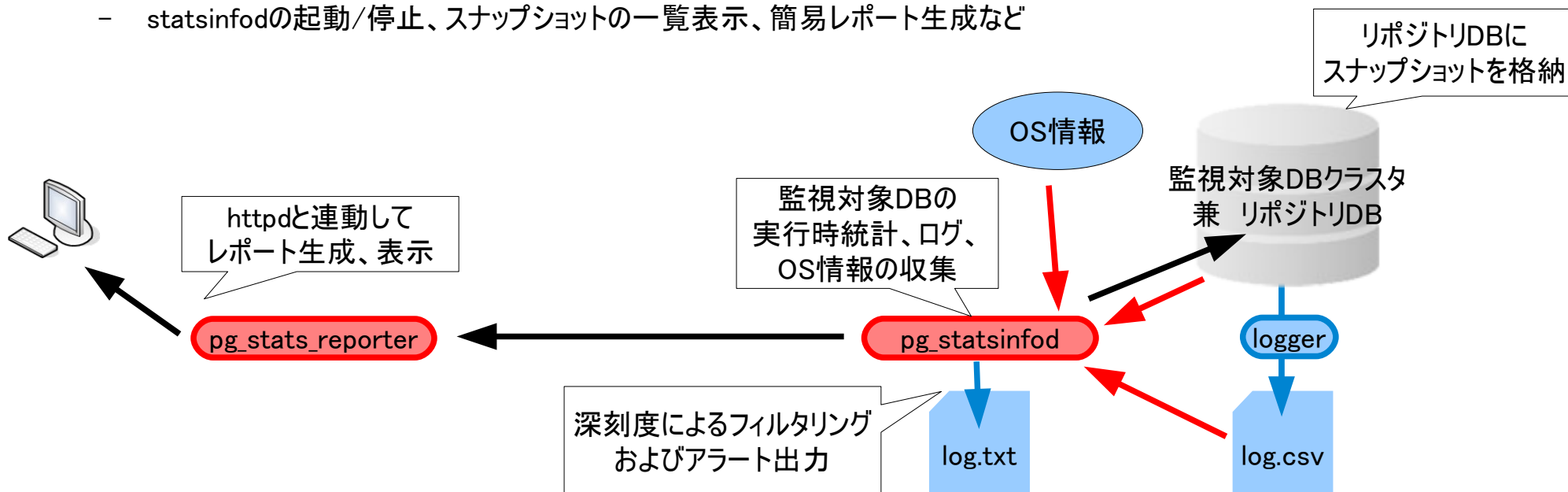
- 自動VACUUMによる性能影響は目安として10%～程度ある
- VACUUMの効果を得るには、必要なテーブルが正しくVACUUMされていること
→特に、同一インスタンスで複数DBが稼働している場合
- pg_stat_all_tablesのn_tup_hot_upd(HOTによる更新行数、多いほど良い)や、n_dead_tup(VACUUM対象の行数、ずっと多いのはダメ)はチェックすること

pg_statsinfoを使ってみよう



pg_statsinfoとは

- pg_statsinfo で、前述の確認項目を含むほぼすべての統計情報を収集
 - スナップショット型で取得し、リポジトリに格納
 - Oracle の Statspack / AWR のような感覚で利用可能
 - 平常時の処理傾向の把握と、性能劣化の予兆認識に活用できる
 - 1つのリポジトリに対して、複数DBインスタンスを登録可能
 - pg_stats_reporter との組み合わせでブラウザでの参照も可能
 - pg_statsinfoコマンドで各種操作を行う
 - statsinfodの起動/停止、スナップショットの一覧表示、簡易レポート生成など



pg_statsinfoを使ってみる

- インストール

```
$ cd <postgresのcontrib配下>/
$ tar zxvf pg_statsinfo-2.5.0.tar.gz
$ cd pg_statsinfo-2.5.0
$ make USE_PGXS=1
$ su
# make USE_PGXS=1 install
```

- PostgreSQLのconfigure時にXMLのサポートを追加しておくこと(rpmならOK)

- pg_statsinfoの設定はpostgresql.confに追記する

```
shared_preload_libraries = 'pg_stat_statements,pg_statsinfo'
pg_statsinfo.snapshot_interval = 3600          # スナップショットの取得間隔
pg_statsinfo.enable_maintenance = 'on'        # 自動メンテナンス設定
pg_statsinfo.maintenance_time = '00:02:00'    # 自動メンテナンス実行時刻設定
pg_statsinfo.repository_keepday = 7           # スナップショットの保持期間設定

log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log' # ログファイル名を指定する
log_min_messages = 'log'                       # ログへ出力するメッセージレベル。
pg_statsinfo.syslog_min_messages = 'error'      # syslogに出力するログレベルを指定する。
pg_statsinfo.textlog_line_prefix = '%t %p %c-%l %x %q(%u, %d, %r, %a) '
# pg_statsinfoがテキストログに出力する際、各行の先頭に追加される書式を指定する。
# log_line_prefixと同じ形式で指定する。
pg_statsinfo.syslog_line_prefix = '%t %p %c-%l %x %q(%u, %d, %r, %a) '
# pg_statsinfoがsyslog経由でログを出力する際、各行の先頭に追加される書式を指定する。
pg_statsinfo.stat_statements_max = 30
```

簡易レポート

- pg_statsinfoで、上述のDBチューニングの確認ポイントが全て確認可能

レポート項目	内容
Summary	環境(ホスト名、バージョン)、スナップショットID(開始と終了)など
DatabaseStatistics	データベース単位の統計情報(キャッシュヒット率、トランザクション数など)
InstanceActivity	WAL出力量やセッション数
OSResourceUsage	CPU(user/system/idle/iowait)の遷移および負荷状況(Load average)
DiskUsage	デバイス毎のI/O遷移
LongTransactions	ロングトランザクション
Lock Conflict	一定時間ロック状態が継続したSQL
NotableTables	注意すべきテーブル(更新が多い/アクセスが多い/断片化)
CheckpointActivity	原因別のチェックポイント回数や処理時間
AutovacuumActivity	自動VACUUMの発生状況
QueryActivity	SQLや関数の情報(所要時間/実行回数など)

- 簡易レポートの生成

```
$ pg_statsinfo -r All -U postgres -d pgdb1
/* -Uや-dでリポジトリDBを指定 */
/* -rでレポートの対象とするスナップショットを指定 */
```

pg_statsinfoのレポート出力例

- SQLではリアルタイムな値のみ取得できていた
- スナップショットを保存し、その差分でレポートを生成するため、有用な情報が得られる

/* Database Statistics */

Database Name : pgdb1
Database Size : 2997 MiB
Database Size Increase : 0 MiB
Commit/s : 7489.429
Rollback/s : 0.000
Cache Hit Ratio : 88.200 %
Block Read/s (disk+cache) : 39085.771
Block Read/s (disk) : 4611.903
Rows Read/s : 90508.565

キャッシュヒット率の確認

/** WAL Statistics **/

WAL Write Total : 1487.714 MiB
WAL Write Speed : 3.759 MiB/s

WAL生成量
→チェックポイント間隔の指標

/* Notable Tables */

/** Heavily Updated Tables **/

Database	Schema	Table	INSERT Rows	UPDATE Rows	DELETE Rows	Total Rows
pgdb1	public	pgbench_accounts	20000000	0	0	20000000

更新量の多いテーブル
→VACUUM間隔の指標

- 簡易レポートの生成

SQLチューニングの基本



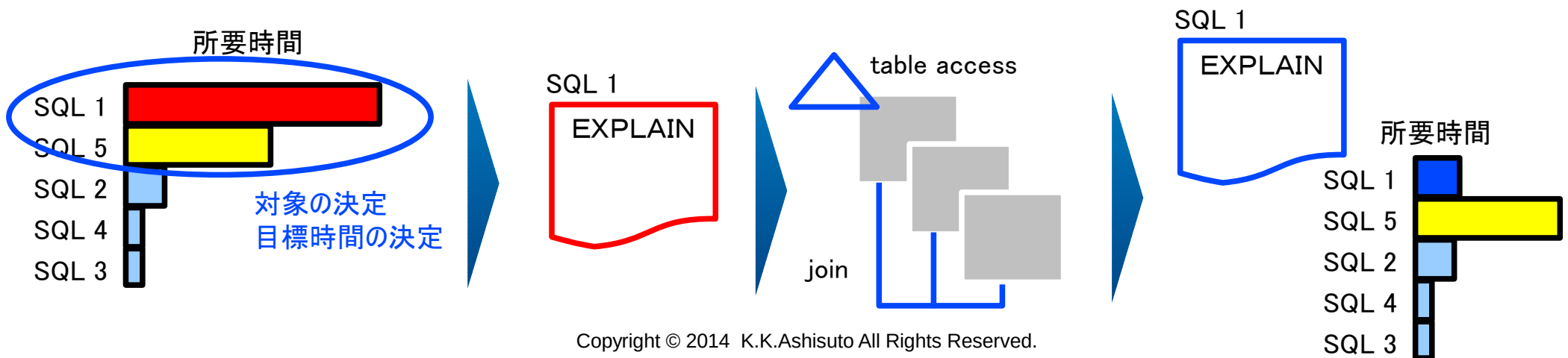
SQLチューニングの考え方

- SQLチューニングとは

- 実行時間が長いSQLを対象に、レスポンス要件を満たすように改善する
- 重い処理が無くなったことで全体のパフォーマンスが向上することもあるが、新たに作成した索引が影響して、他の処理を遅くすることもある

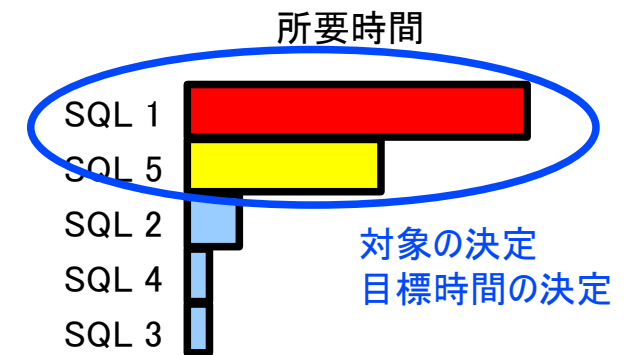
- SQLチューニングのステップ

- どのSQLをどこまで早くするか
- 現状を確認する
- どうやって早くするか
- 効果を測定する



チューニング対象を決定

- 統計情報から実行時間の長いSQLを確認する
 - pg_stat_activity
 - pg_stat_statements(contribツール) ⇒ pg_statsinfoと連携してレポート可能
- エラーログへの出力から実行時間の長いSQLを確認する
 - postgresql.conf の log_min_duration_statementパラメータ
 - パラメータで指定した時間以上を要するSQLをサーバログに出力
 - auto_explain(contribツール)
 - 実行に一定時間を要したSQL文と実行計画をサーバログに出力
- キャッシュヒット率やアクセスブロック数を基準にすることもある
 - 1行を対象にした検索なのに、大量のブロックにアクセスしている場合など
 - pg_stat_all_tables/pg_statio_all_tables



実行計画の確認

EXPLAIN

EXPLAIN

- EXPLAIN または EXPLAIN ANALYZE でSQLの実行計画を取得

```
postgres=# EXPLAIN SELECT b.logno,e.empname,b.log_text FROM log_master m,log_body b,emp e
        WHERE b.logno=m.logno AND e.empno=m.empno AND m.status = 'WI';
```

QUERY PLAN

Hash Join (cost=79.00..1754.21 rows=10000 width=66)

Hash Cond: (m.empno = e.empno)

-> Merge Join (cost=0.00..1525.21 rows=10000 width=24)

Merge Cond: (m.logno = b.logno)

-> Index Scan using log_master_pkey on log_master m (cost=0.00..6907.33 rows=31373 width=8)
Filter: (status = 'WI'::bpchar)

-> Index Scan using log_body_pkey on log_body b (cost=0.00..707.27 rows=10000 width=20)

-> Hash (cost=54.00..54.00 rows=2000 width=50)

-> Seq Scan on emp e (cost=0.00..54.00 rows=2000 width=50)

計画ツリー

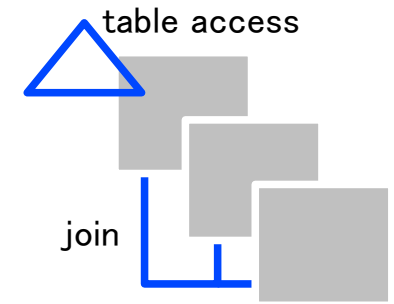
計画タイプ(プラン)

コストの推定値

推定行数や列幅

- 計画ツリーの階層が深いものがから順に実行されている
 - 通常、各テーブルへのアクセスから始まる
- EXPLAIN ANALYZEの場合、実際にSQLが実行される
 - 所要時間、取得した行数、ループ回数が記録される
 - SQLが実行されてしまうため、更新処理の場合は必ずBEGINでトランザクションを開始してから実施する

PostgreSQLの計画タイプ



- 計画タイプを確認し、意図した方法が選択されているかを調査

計画タイプ	内容
Seq Scan	全表スキャン: 表内の大量の行にアクセスする場合に有効
Index Scan	索引スキャン: 表内のごく一部の行にのみアクセスする場合に有効
Index Only Scan	索引のリーフブロックのみにアクセスするスキャン(表へのアクセスをスキップ)
Bitmap Index Scan	ビットマップ使用した索引スキャン
Bitmap Heap Scan	ビットマップスキャンで表にアクセス
Function Scan	ファンクションの実行結果に対するスキャン
Nested Loop	ネステッドループ結合: 片方の表のうち、ごく少数の結果を条件として他方の表からデータを取得
Merge Join	マージ結合: 両方の表の行数が多い場合、ソートし、上から順に値を比較して該当行を抽出
Hash Join	ハッシュ結合: 表の行数に差があり、かつ小さい表の重複が少ない場合に有効

- 必要なIndexが作成されているか
- 意図したプランが選択されるSQLを書いているか

対策：索引が使われないパターン

CATEGORY.....索引利用ルール

Tips

334

▶ Level

索引を使用するため、WHERE句の索引列で関数処理を行わないように書き換える

索引スキャンを実行したいSQLであっても、WHERE句の索引列を関数処理などで変更してしまうと、関数処理された値と索引に格納されている列値が一致しないため、索引が使用されません。

PostgreSQL

※ hiredate 列に索引が作成されている前提

× `SELECT ename FROM emp_huge
WHERE TO_CHAR(hiredate, 'YYYYMMDD') = '19801217';` — ①

QUERY PLAN

Seq Scan on emp_huge
Filter: (to_char((hiredate)::timestamp with time zone,
'YYYYMMDD')::text) = '19801217'::text)

PostgreSQL

○ `SELECT ename FROM emp_huge
WHERE hiredate = TO_DATE('19801217', 'YYYYMMDD');` — ②

QUERY PLAN

Index Scan using idx_huge_hiredate on emp_huge
Index Cond: (hiredate = to_date('19801217'::text,
'YYYYMMDD')::text))

索引列を関数処理で変更してしまうと、索引が使用されません(①)。

索引を使用するには、索引列は変更せず、逆側で関数処理を行うようにします(②)。



- 演算/関数処理
- データ型の暗黙変換
- NOT条件

など、SQLの書き方で索引が使われないパターンを紹介

対策：索引を活用してソートを省略



CATEGORY.....索引活用

Tips

342

ORDER BY 句で索引を使用し、ソート処理を省略する

Level



索引内のエントリはソートされた状態で格納されています。そのため、ORDER BY 句で索引列を使用し、ソート処理を省略することでSQL実行時の負荷を下げることができます。

PostgreSQL

※ empno 列に索引が作成されている前提

```
○ SELECT ename FROM emp_huge
  WHERE empno > 9000
  ORDER BY empno; — ①
```

QUERY PLAN

```
-----
Index Scan using idx_huge_empno on emp_huge
Index Cond: (empno > 9000::numeric)
```

ORDER BY 句に索引列を指定することで、索引からソート済みのデータを取得し、ソート処理を省略することができます(①)。

Oracleとは異なり、索引列に主キー制約やNOT NULL制約、または「IS NOT NULL」によるNULLの排除が指定されていない場合でも索引が使用できます。

- ソート
- 集計

など、索引があることで表へのアクセスをスキップして処理を行えるパターンを紹介

対策: 結合




PostgreSQLでの注意点

● Materialized View

- PostgreSQL9.3でマテリアライズド・ビューが使用可能となった
- 実体をもったビューで、結合や集計結果を保存しておくことで高速にアクセスできる
- ただし、現在は以下の制約があり、今後の機能向上に期待
 - リフレッシュ時に元のテーブルに対して非常に強いロックを取得
 - 自動リフレッシュや差分(高速)リフレッシュがなく、手動の完全リフレッシュのみ

● Index Only Scan

- PostgreSQL9.2でIndex Only Scanが使用可能となった
- インデックスで必要なデータが得られる場合、テーブルへのアクセスをスキップする機能
- ただし、対象のテーブルがVACUUM直後(VACUUM後に更新されていない)ことが選択される条件であり、使用する場合は注意が必要



PostgreSQL9.4betaの話

性能関連でのPostgreSQL 9.4への期待

- 性能に関連するいくつかの変更が明らかになっている
 - 新しいデータ型 JSONB型
 - GINインデックスの軽量・高速化
→この2つが本バージョンの目玉として取り上げられている
 - WALロックの改善
 - hugepageへの対応
→マイナーな変更だが、PostgreSQLの適用領域を拡大する要因かも
- 運用担当者はチェックしておくべき変更点
 - pg_prewarm
 - システムパラメータの動的かつ永続的な変更が可能に

JSONB型とGINインデックス

- JSONデータを解析、整形したバイナリとして格納するJSONB型
 - 従来のJSON型と比べて高速、軽量
 - GINインデックスに対応
- GIN(汎用転置)インデックスのサイズ削減、性能向上
 - 配列、ハッシュ(hstore)、全文検索テキストなど、複数要素を持つデータ型に対して「ある要素を持つもの」を検索するときに使われる
- 互いに進化することで、NoSQL製品に負けない非リレーションにも対応

hugepage対応とWALロックの軽減

- 大規模メモリを扱うRDBMSでは、メモリ管理のオーバーヘッドが懸念される
 - 特に、shared_buffersが8GBを超え、同時接続数が非常に多く、軽量の処理を多数実行するような場合に、CPUが高騰してしまう
 - Linuxの従来のページを扱う場合、RDBMS側で非常に多くのページ数を管理しなければならないためCPU負荷が高くなることが原因
 - PostgreSQL 9.4 から、Linuxのhugepageをサポートする
- RDBMSでは障害に備えて全ての変更を変更履歴として書き出す
 - 避けては通れない処理であり、これがしばしばボトルネックとなる
 - PostgreSQL 9.4 では、更新に対するWAL生成量を減る
 - WAL出力時のロック競合が軽減し、複数ユーザが同時に更新をかけた際の待機が減る
- これらの組み合わせにより、高性能なサーバスペックを活かし、より広い用途でPostgreSQLを活用できる可能性が考えられる

運用の変更

- pg_prewarm
 - 指定したテーブルやインデックスのデータをshared_buffersやOSバッファに載せる
 - contribツールとして9.4で追加される
- EXPLAIN ANALYZE出力が改善
 - プラン作成時間と実行時間が別々に表示されるように変更される
- ALTER SYSTEMコマンド
 - postgresql.confとは別にpostgresql.conf.autoを作成し、パラメータの変更を永続的に反映させることができる

本日のまとめ

- DBチューニング
 - ・ 構築時から考えておくべきポイントを整理
 - ・ パラメータチューニングと、現状の確認
- SQLチューニング
 - ・ 実行計画の確認方法を解説
 - ・ 各プランの特徴を整理
- pg_statsinfoの紹介
 - ・ チューニングに役立つ情報が確認可能
- PostgreSQL 9.4 betaの話
 - ・ 性能関連を簡単に紹介

性能関連パラメーター一覧

パラメータ名	推奨設定	反映タイミング
shared_buffers	物理メモリの25%	サーバ再起動
wal_buffers	自動任せ	サーバ再起動
work_mem	処理ごと	即時
checkpoint_segments	16以上	再読み込み
checkpoint_timeout	30分	再読み込み
checkpoint_completion_target	0.9	再読み込み

SQLチューニングの流れ

ステップ	PostgreSQLでの方法
問題となるSQLを特定	統計情報、ログ出力
実行計画の取得	EXPLAIN ANALYZE
対処の実施	SQL修正、索引の調整
効果の測定	TAT測定、実行計画取得

➤ ハンズオンに続く