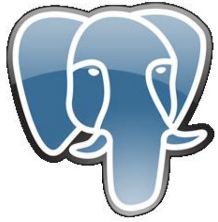


PostgreSQL運用テクニック

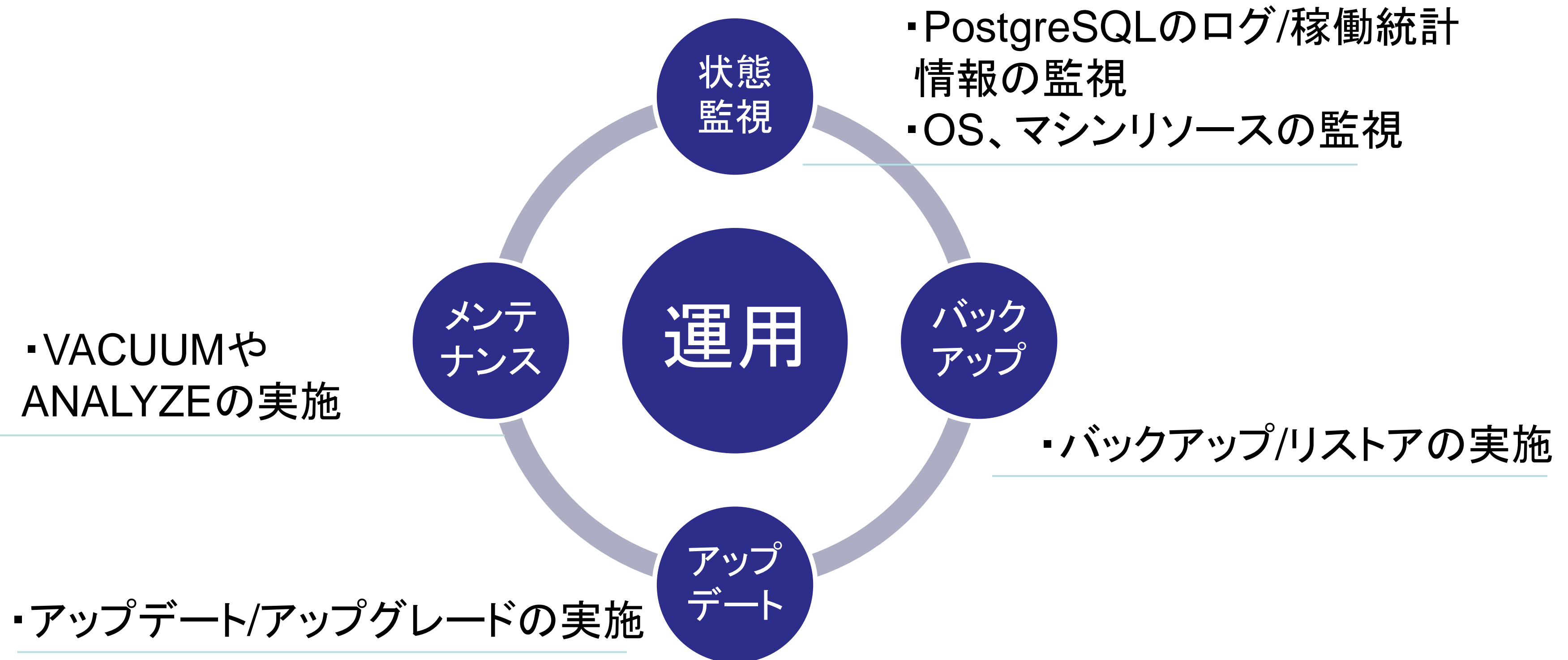
JPUG 2011夏セミナ
2011.6.25

PostgreSQLしくみ分科会
笠原 辰仁,坂本昌彦

アジェンダ

- 運用には何がある？
  - 状態監視のテクニック
  - 異常発生時のテクニック
  - バックアップ・リストアのテクニック
  - アップデート・アップグレードに関するテクニック
  - メンテナンスに関するテクニック

運用にはどんな作業がある？



・本日は、運用にまつわる、知っておくと有用ないくつかのテクニックをご紹介します。
・PostgreSQLの9.0、9.1では様々な運用性の向上が図られていますので、そこにも少し触れながらお話します。

状態監視に関する運用テクニックを紹介します

- ログの出力情報をlog_line_prefixで設定する。
- application_nameを使うと原因を識別しやすくなる。
- log_filemodeを使ってログの監視を行いやすくする。
- log_temp_filesで一時ファイル書き出しを捕捉する。
- スロークエリの実行計画を捕捉する。
- テーブルの稼動状況を把握する。
- レプリケーションの状況を把握する。

ログの出力情報をlog_line_prefixで設定する

監視

- log_line_prefixとは？
  - ログの接頭にどんな情報を付与するかを設定するパラメータ
- どう使えばよいの？
  - **(9.0 ～) log_line_prefix=‘[%t][%p][%c-%l][%x][%e]%q (%u, %d, %r, %a) ‘**
  - **(～8.4) log_line_prefix=‘[%t][%p][%c-%l][%x]%q (%u, %d, %r) '**
  - **上記ともに、最後に半角空白を入れると良い (可視性向上のため)**

prefix	説明	対応ver
%t	タイムスタンプ	ALL
%p	プロセスID	ALL
%c	セッションID	ALL
%l	各セッションまたは各プロセスのログ行番号	ALL
%x	トランザクションID（参照処理などXID未割り当ての場合は0）	ALL
%e	SQLSTATE エラーコード	9.0～
%q	セッションプロセス(クライアントからの通常処理)のみ、これより後ろのprefix情報を出力する。	ALL
%u	ユーザ名	ALL
%d	データベース名	ALL
%r	処理元のホスト名、またはIPアドレス、およびポート番号	ALL
%a	アプリケーション名	9.0～

ログの出力情報をlog_line_prefixで設定する

監視

- **ログの読み方**
- **log_line_prefix='[%t][%p][%c-%l][%x][%e]%q (%u, %d, %r, %a) '**

```
[2011-06-03 19:24:27 JST]  <- タイムスタンプ
[30979]  <- プロセスID
[4de8b60f.7903-7]  <- セッションIDとログ行番号
[87571]  <- トランザクションID
[23505]  <- エラーコード
(postgres, test, 127.0.0.1(34990), testAP) <- ユーザ名と処理元のIP、ポート、およびAP名
ERROR:  duplicate key value violates unique constraint  "test_pkey "  <- エラーメッセージ


/* autovacuumなど、バックグラウンドプロセスによる処理 */
[2011-06-03 19:26:49 JST][30981][4de8b635.7905-1][87595][00000]
LOG:  automatic vacuum of table "test.public.pgbench_branches": index scans: 1
        pages: 5 removed, 5 remain
        tuples: 456 removed, 2 remain
        system usage: CPU 0.00s/0.00u sec elapsed 0.27 sec
```

application_nameを使うと原因を識別しやすくなる

監視

- application_nameとは？
  - 個々のアプリケーションごとに識別子を設けるためのパラメータ
- どう使うの？
  - libpqやpsqlの接続パラメータで指定
  - postgresql.confでグローバルに指定
  - JDBCの接続パラメータで指定(JDBC9.1-xxxから)
    - JDBC9.0以前はAP側のSET文で指定
  - PGAPPNAME環境変数で指定
  - レプリケーション時にスタンバイ側で設定するprimary_conninfoで指定
- メリットは？
  - エラーなどの発生時に、ログやpg_stat_activityの情報から、どのアプリケーションに異常なのかを判別できる

```
# libpqの接続パラメータ
conn = PQconnectdb(“application_name=‘MyApName’”);

# psql の接続パラメータで
psql “application_name=MyApName”

# SET文で指定
SET application_name=MyApName;
```

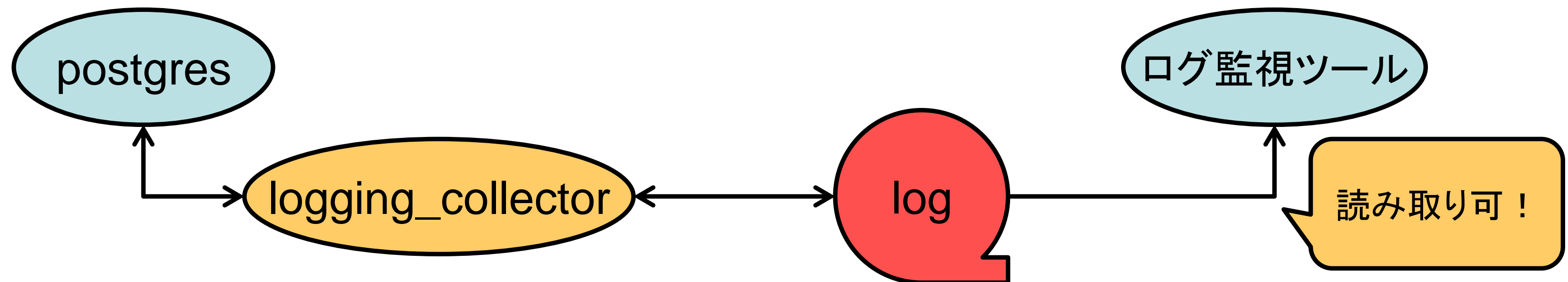
log_file_modeでログ監視を柔軟に行う

監視

- log_file_modeとは？
  - ログのパーミッションを指定するパラメータ
- どう使うの？
  - ログのパーミッションを指定するだけです
  - log_file_mode='644' など
- メリットは？
  - 0644 にしておき、PostgreSQLを操作するユーザ以外からも読み取り可能にすることで、サードパーティ製のログ監視プロダクトと上手に連携できる

従来は、0600 (postgresの操作ユーザのみ読み書き可能)固定
- 外部からのログ監視をしたい場合はsyslogにすることが多くなる
- syslogへの大量のログ出力がネックになることが･･
- rsyslogなどが一般になれば、問題ではなくなるが、使いづらい
- stderr + logging_collector = on の組み合わせが、性能を考慮するとベスト

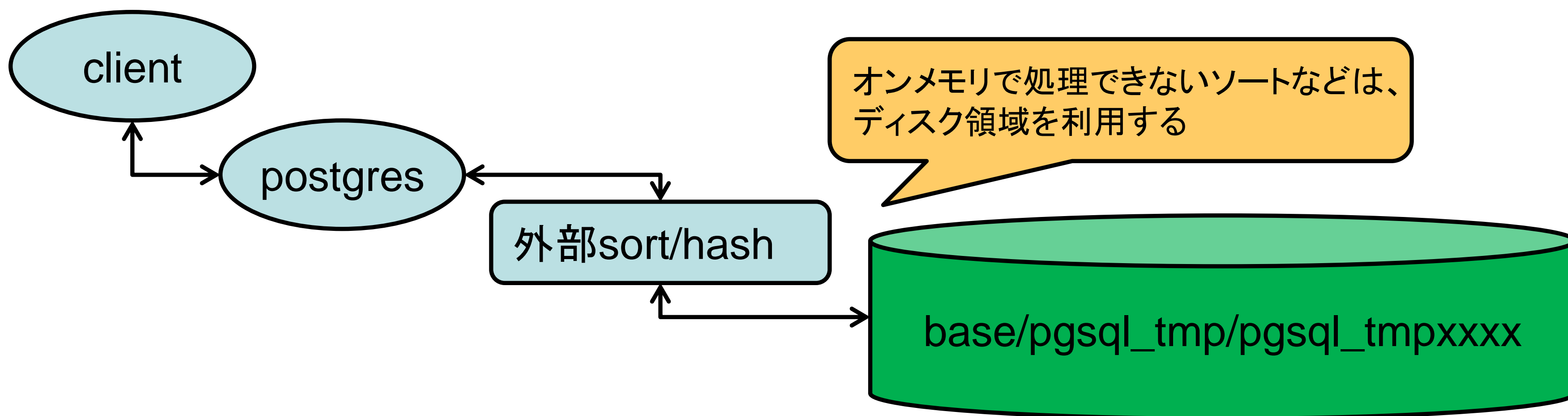
log_file_modeを活用することで、監視がしやすくなります



log_temp_filesで一時ファイルの書き出しを捕捉する

監視

- log_temp_filesとは？
  - ソートやハッシュなどで一時ファイル書き出しが発生した場合にロギングするかどうかを設定するパラメータ
- どう使うの？
  - サイズを指定し、指定サイズ以上の一時ファイルが書き出されたらログに出します
- メリットは？
  - データ量の増加や突然のプラン変更などにともなう一時ファイル書き出しの大量発生を検知し、ディスクの空き領域の確保やプランチューニングを行えます




```
[2011-06-03 20:19:13 JST][31291][4de8c330.7a3b-3][0][00000]
(postgres, postgres, [local], psql)
LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp31291.0", size 475136
```

スロークエリの実行計画を捕捉する

監視

- auto_explainモジュールを活用する
  - 遅いSQLに対してのみ、実行計画をロギングしてくれるcontribモジュール
- どう使うの？
  - SQLの許容レスポンスタイム値を設定し、設定値以上時間のかかったSQLの実行計画をログに出力します
  - 出力形式は単純なテキストのほか、XMLやJSON、YAMLをサポートしてます
- メリットは？
  - SQLの性能劣化の原因解析用に実行計画と実際のレスポンス値を得ることができます
  - 商用環境などでスロークエリの問題切り分けのために実行計画を取りたくても、なかなか試行するのが難しい、といったケースでは特に有効です

```
[2011-06-03 21:35:27 JST][31758][4de8d50e.7c0e-3][0][00000](postgres, postgres,
127.0.0.1(36553), psql) LOG:  duration: 16.464 ms  plan:
        Query Text: SELECT * FROM test  ORDER BY name LIMIT 1000;
        Limit  (cost=2927.16..2929.66 rows=1000 width=37)
          ->  Sort  (cost=2927.16..3027.16 rows=40000 width=37)
                Sort Key: name
                ->  Seq Scan on test  (cost=0.00..734.00 rows=40000 width=37)
```

スロークエリの実行計画を捕捉する

監視

- 使い方は？
  - インストール(make install or contrib RPM導入)後、postgresql.confに設定するだけです
- どんな設定ができる？
  - log_min_duration <時間>
    - どれくらい時間のかかったSQLの実行計画をログに出すか？
  - log_fromat <フォーマット名>
    - どの形式でログに出すか？デフォルトのtextの他、XML、JSON、YAMLがある
  - log_verbose <boolean>
    - やや詳しい実行計画(出力する列名やソート対象のキー名など)を出すか？
  - log_analyze <boolean>
    - EXPLAIN ANALYZE相当の情報を出すか？負荷の高いシステムで有効にする場合は事前によく検討/試験をしてから！
  - log_buffers <boolean>
    - 上記のANALYZEに加えてBUFFER情報も出すか？注意点は上記と同じ

```
# -----------------------------
# PostgreSQL configuration file
# -----------------------------
(略)
 shared_preload_libraries =  ‘auto_explain ‘
(略)
custom_variable_classes = 'auto_explain’
auto_explain.log_min_duration =  ‘10s ‘ # 10秒以上かかったSQLの実行計画をログ出力
auto_explain.log_format =  ‘json ‘      # ログをJSONフォーマットで出力
auto_explain.log_verbose = on           # Verbose情報を出力
```

その他の有用なログ関連のパラメータ

監視

- log_min_duration_statement <時間>
  - 設定した時間以上かかったSQL文を、その時間とともにログに出力します
  - スロークエリの把握に非常に便利です
- log_lock_waits
  - dead_lock_timeout(def. 1sec)以上の時間、ロック待ち状態が発生したSQLをその時間とともにログに出力します
  - ロック待ちの多発の把握に使え、システムのネックになっているかどうかの切り分けに便利です
- log_connections/log_disconnections
  - クライアントからの接続/接続断の情報をログに出力します
  - disconnectionsを有効にすると、そのクライアントが接続していた累積時間を出力してくれるので、監視などに有効です

テーブルの稼働状況を監視する

監視

- pg_stat_all_tablesを活用しよう
  - 各テーブルの稼働統計情報を格納しているビュー
- 何がわかるの？
  - 各テーブルに実施されたDMLや表/インデックススキャンの回数、現在格納している有効/無効行数ならびにVACUUMやANALYZEの実施情報

idx_scan	7169
idx_tup_fetch	7169
n_tup_ins	20
n_tup_upd	48819
n_tup_del	0
n_tup_hot_upd	48346
n_live_tup	20
n_dead_tup	0
last_vacuum	2011-06-01 05:10:11.445285+09
last_autovacuum	2011-06-01 05:13:04.566691+09
last_analyze	2011-06-01 04:47:29.450732+09
last_autoanalyze	2011-06-01 05:13:04.575302+09
vacuum_count	4
autovacuum_count	8
analyze_count	1
autoanalyze_count	10

9.1からはVACUUMとANALYZEの実施回数も記録されるようになりました！

レプリケーションの状況を把握する

監視

- pg_stat_replicationを活用しよう
  - レプリケーションに関するアクティビティ情報を格納しているビュー
  - 9.1から使用可能です
- どう使うの？
  - レプリケーションを実施している各プロセスの情報を俯瞰でき、遅延や状態を把握できます

procpid	19398	21602
usesysid	10	10
usename	postgres	postgres
application_name	sync_slave	async_slave
client_addr	127.0.0.1	127.0.0.1
client_hostname		
client_port	53690	45785
backend_start	2011-06-06 23:29:48.900515+09	2011-06-07 05:26:34.784452+09
state	STREAMING	STREAMING
sent_location	0/21D65AE0	0/21D65AE0
write_location	0/21D65AE0	0/21D65AE0
flush_location	0/21D65AE0	0/21D65878
replay_location	0/21D65878	0/21D65878
sync_priority	1	0
sync_state	SYNC	ASYNC

レプリケーションの状況を把握する

監視

- pg_stat_replicationを活用しよう
  - レプリケーションに関するアクティビティ情報を格納しているビュー
  - 9.1から使用可能です
- どう使うの？
  - レプリケーションを実施している各プロセスの情報を俯瞰でき、遅延や状態を把握できます

state : 現在のレプリケーション状態
STARTUP : receiverとsenderのコネクション開始
BACKUP : pg_basebackupの実施中
CATCHUP : 古いWALの受信/再生中
STREAMING : 最新のWALの受信/再生中

*_location : sender/receiverの扱っているWALの位置
sent_location : マスタがどこまでのWALを送信したか
write_location : スタンバイがどこまでのWALを受信したか
flush_location : スタンバイがどこまでのWALをディスクに書いたか
replay_location : スタンバイがどこまでのWALを再生したか

back[illegible]art	2011-06-06 23:29:48.900515+09	2011[illegible]:26:34.784452+09
state	STREAMING	STREAMING
sent_location	0/21D65AE0	0/21D65AE0
write_location	0/21D65AE0	0/21D65AE0
flush_location	0/21D65AE0	0/21D65878
replay_location	0/21D65878	0/21D65878
sync_priority	1	0
sync_state	SYNC	ASYNC

- 異常発生時に知っておくと有用な運用テクニックを紹介します
  - 異常/残存プロセスを安全に削除する
  - クラッシュリカバリ後に稼働統計情報を修復する
  - 残存プロセスを早めに除去する

異常/残存プロセスを安全に削除する

監視

- 予想以上に長時間処理を実施している/終わらないプロセスを停止したい
- うっかり仕掛けてしまったVACUUMをキャンセルしたい
- こんな時、間違っても kill –SIGKILL や kill -9はやらないように！
  - サーバプロセス全体が停止してしまいます
  - クラッシュリカバリ扱いとなります

- どうすれば良い？
  - クエリのキャンセルは･･
    - kill –SIGINT 対象のPID
もしくは
SELECT pg_cancel_backend(対象のPID);
  - プロセスを落とすには
    - kill –SIGTERM 対象のPID
もしくは
SELECT pg_terminate_backend(対象のPID);

クラッシュリカバリ後に稼働統計情報を修復する

監視

- どうして修復が必要なのか？
  - PostgreSQLは、クラッシュした場合に稼働統計情報をクリアします
    - 稼働統計情報はトランザクショナルな情報ではないので、クラッシュ時は念のためPostgreSQLが内部でクリアしてます
  - 稼働統計情報は、autovacuumの処理実施のトリガとなるため、早めの回復が望ましいです
    - 稼働統計情報から、ANALYZEやVACUUMが必要かどうかを判断しています
- どうやって修復するの？
  - ANALYZEを実施します
    - ただし、8.4以前のバージョンではANALYZEを2回実施するか、ANALYZE前にテーブルにアクセス(簡単なSELECTでも良い)をしておく必要があります
      - ANALYZEに時間のかかる巨大なテーブルがある場合には有効なTIPSです
  - ANALYZEにより、テーブルの有効行数(pg_stat_all_tables.n_live_tup)と不要行数(pg_stat_all_tables.n_dead_tup)が最新化されます

残存プロセスを早めに除去する

監視

- 残存プロセスとは？
  - よくあるのは、AP側が落ちたり、APとのNWが断絶された際、APからの応答をずっと待っているDB側のプロセスがいつまでも残ってしまう
  - コネクション数がmaxに達してしまったり、ロングトランザクション化してしまったり、という問題が起こる
  - この場合、タイムアウトを短めに設定することで残存プロセスを早めに除去すると良いです
- どうすれば良いの？
  - tcp_keep_*パラメータを活用しよう
  - TCPコネクションのタイムアウトを制御するパラメータ
- どう使うの？
  - APサーバのダウン時に、NW等が断絶してしまいDBサーバ側に残存してしまったプロセスを早めに除去したい時に有用です。
  - 例えば、残存プロセスを1分半程度で除去したい場合は下記の様にします

パラメータ名	設定値	Linuxのデフォルト値	説明
tcp_keepalives_idle	60 秒	7200 秒	KeepAliveパケット送出までの待機時間
tcp_keepalives_interval	5 秒	75 秒	KeepAliveパケット送出に応答がなかった場合の再送間隔
tcp_keepalives_count	5	9	KeepAliveパケット送出回数の上限

バックアップ・リストアに関する運用テクニックを紹介します

- バックアップ・リストア方式を知って、選ぶ。
- きちんとベースバックアップを取得する。
- アーカイブログを管理する。
- バックアップコマンドpg_basebackupを使う。
- PITRに必要なものを用意する。

バックアップ・リストア方式を知って、選ぶ。

バックアップ

- どのようなバックアップ方法があるの？
  - これまで、好まれてきたのは「物理バックアップ（オンライン）」
    - ベースバックアップと、WALアーカイブを取得する方式。PITRに対応している。
    - スナップショット機能（LVMやストレージの機能）との連携例も。
  - 今後は「pg_basebackup コマンド」（Ver9.1～）が本命
    - ～Ver. 9.0 ではpg_rman がよい代替となります。
- 番外編
  - レプリケーションもバックアップ？ただし、オペミスは戻せません。

定番　9.1～

	論理バックアップ	物理バックアップ（オフライン）	物理バックアップ（オンライン）	物理バックアップ（オンライン libpq）	外部ツール	レプリケーション
バックアップコマンド	pg_dump / pg_dumpall	cp, tar rsyncなど	pg_start_backup() cp, tar,rsyncなど	pg_basebackup	pg_rman	(ストリーミングレプリケーション)
リストアコマンド	pg_restore / psql	特に無し	（PITR機能）	（PITR機能）	pg_rman	-
売り文句	・手順が容易 ・リモートで実施可 ・DB単位でバックアップ	・手順が容易	・標準的な方法	・手順が容易 ・リモートで実施可	・手順が容易 ・複数世代管理	・簡単
どこまで戻せる？	バックアップ時点	バックアップ時点	任意の時点	任意の時点	任意の時点	最新の時点のみ
レプリカになれる？	-	-	なれる	なれる	（なれる）	なれる
リモートで実施可能？	リモート可	-	-	リモート可	-	リモート可
スナップショット機能と連携可？	-	可能	可能	-	-	-
注意点など	設定ファイルなど、別途バックアップが必要	・サーバを停止させる必要がある				厳密にはバックアップではない。オペミス/改竄は防げない

きちんとベースバックアップを取得する。

バックアップ

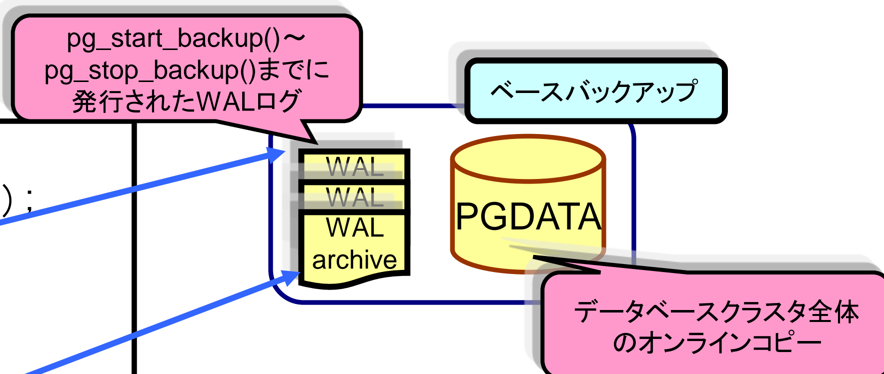
- 独立したバックアップセットとは？
  - データベースクラスタ全体のオンラインコピー
  - バックアップ取得中でのWALログ

- どうするの？
  - pg_xlogfile_name_offset() を pg_start_backup(), pg_stop_backup() と組み合わせると、必要なWALアーカイブ名がわかります。

WALアーカイブ名の取得方法

```
SELECT * FROM
pg_xlogfile_name_offset(pg_start_backup('test','t'));
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000012 |          32

SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 000000010000000000000014 |     4571128
```



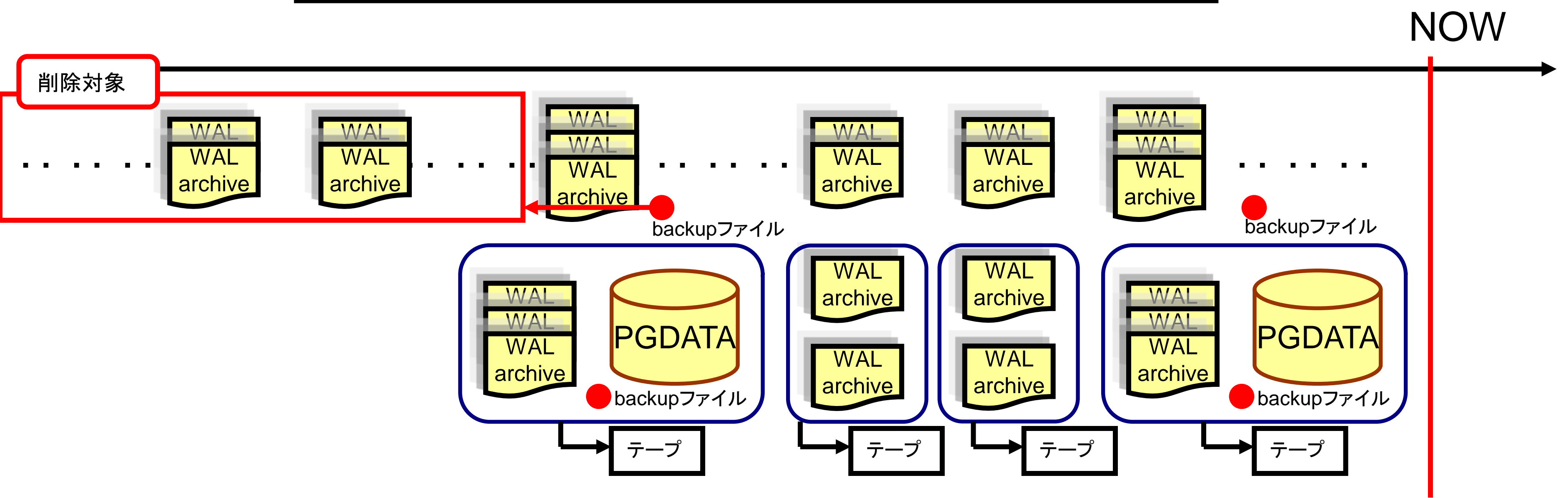
ベースバックアップ取得のスクリプト作成フロー

1. pg_start_backupを発行
2. rsyncでPGDATAをコピー
3. pg_stop_backupを発行
4. 手順1～手順3で得たアーカイブをコピー

アーカイブログを管理する。

- アーカイブログはたまり続けます
  - 消さないと、確実にディスクフルになります。
  - 消してよいアーカイブログはどれでしょうか？
  - ベースバックアップ以前のアーカイブログは不要ですが、具体的にどのファイルでしょうか？
- 解決法
  - pg_archivecleanup が有用です。
    - pg_archivecleanup は、contribモジュールです（ Ver. 9.0～）。
- どうやって使うの？
  - バックアップ履歴ファイルを指定すると、リストアするのに必要なログより前のログを全て削除します。

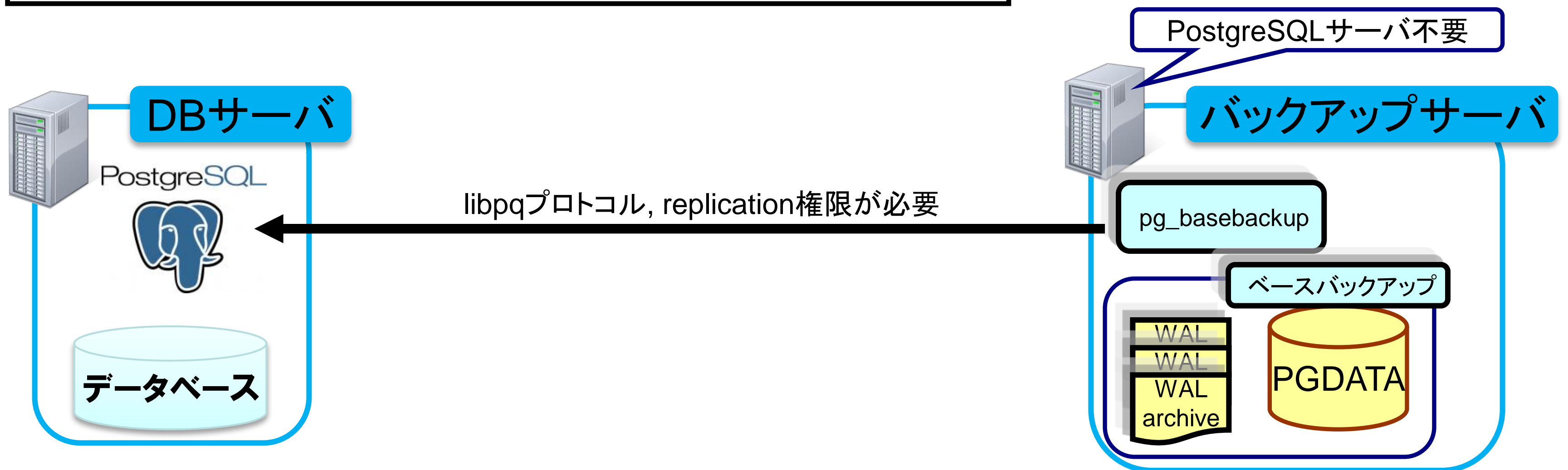
```
$ pg_archivecleanup -d <arc_dir> xxxx.yyyy.backup
```



バックアップコマンドpg_basebackupを使う。

- pg_basebackupの特徴
  - 簡単にベースバックアップが取れる
    - PITRに必要な資材を一式、自動で取得してくれる
    - ベースバックアップ中のWALログも取得できる
  - クライアントアプリなので、リモートからベースバックアップが取れる
    - libpqプロトコルでリモートからも実行可能
    - レプリケーション構築時に重宝
  - アーカイブログの管理等は依然として必要です。

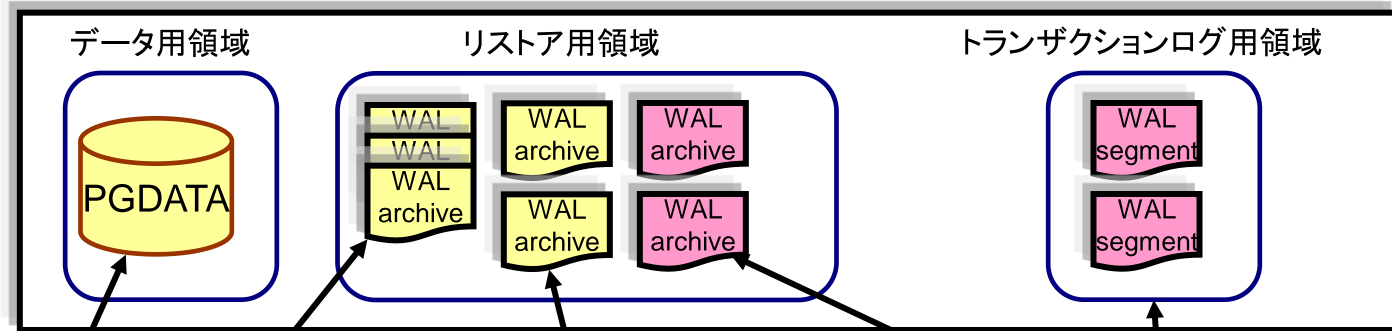
```
pg_basebackup -h db01 -D /bkup/basebkup_1 -x
```



PITRに必要なものを用意する。

バック
アップ

- PITR（Point Time in Recovery）はむずかしい
  - PITRでは、必要なものが不足していても、リストア処理自体は正常に完了してしまう場合があります。
  - 当然データは、障害直前まで戻っていませんが、DBは正常に起動します。
- どうするの？
  - 不足しているものがないか、あらかじめ、きちんとチェックしましょう。特に以下の二つはよくありがちです。
    - 障害発生時におけるトランザクションログの配置を忘れてしまい、障害直近まで戻らずに起動する
    - アーカイブログに抜けがあり、最後までログを当てきれない
  - 特に障害発生時のpg_xlog配下のデータは最重要データです。誤って消すことが無いようにコピーをとりましょう。



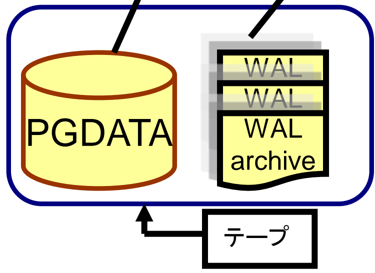
手順1：
・ベースバックアップからデータ用領域をリストア
・リストア用領域にWALアーカイブをリストア

手順2：
全てのアーカイブログバックアップから
WALアーカイブをリストア

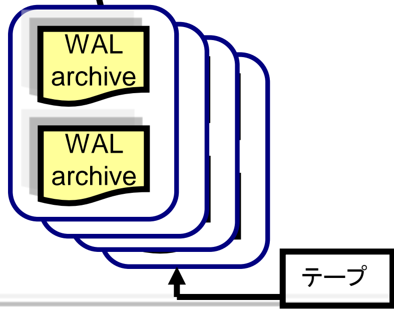
手順3：
・リストア用領域に障害発生時におけるWALアーカイブを
コピーして持ってくる。

手順4：
・障害発生時におけるトランザクションログをリストア用の
PGDATAのpg_xlogにコピーする。これが無い場合、
直近までデータを戻すことができない。

手順5：
・recovery.confを作成し、PostgreSQLを起動する。



テープ



テープ

アップデート

アップデート・アップグレードに関する運用テクニックを紹介します

- アップデートは、バイナリ変更だけで大丈夫です。
- アップグレードはデータの変換も必要です。
- pg_dumpでアップグレードする。
- pg_upgradeでアップグレードする。

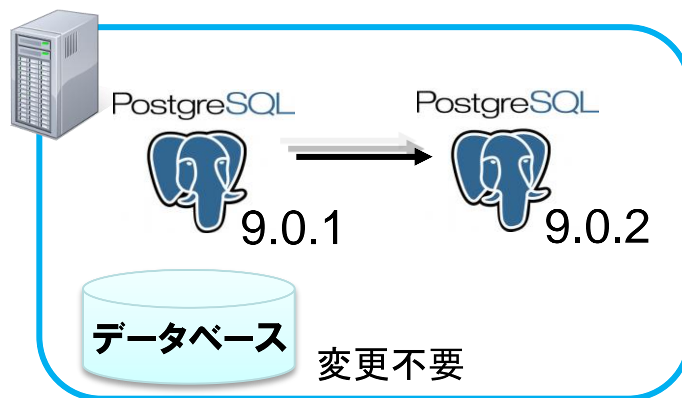
アップデートは、バイナリ変更だけで大丈夫です。

アップデート

- アップデートって？
  - マイナーバージョンアップとも呼ばれます。
  - バージョン番号の上3桁目の変更 (9.0.x から 9.0.y) を行うことです。
  - 実行ファイルの入れ替えでアップデートできます。
    - データファイルはそのまま利用可能です。
    - PostgreSQLを利用するアプリケーション（プログラム）も変更する必要がありません。

- 注意点
  - データ構造に関するバグ修正が行われた場合には、アップデート後にデータの再編成コマンドの投入が必要になる場合があります（過去に事例あり）。リリースノートをよく読みましょう。

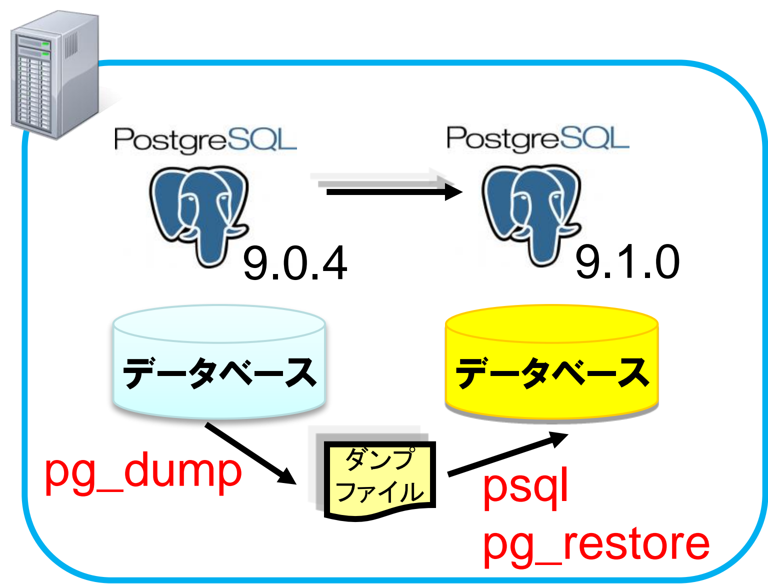
- 手順
  - 念のため、データファイルのバックアップは取得しておくべきでしょう。
  - ソースからバイナリをコンパイルする場合は、configureオプションはそろえましょう。
  - RPMなどのパッケージ管理システムを使う場合は、設定ファイルが変更されるかもしれません。（例：/var/lib/pgsql/.bash_profile , /etc/init.d/postgresql , /etc/pam.d/postgresql …)

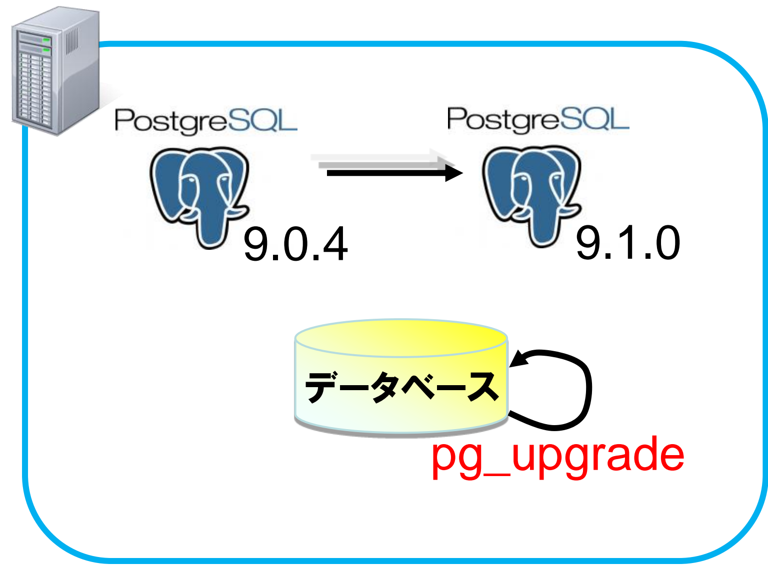


アップグレードはデータの変換も必要です。

アップデート

- アップグレードって？
  - メジャーバージョンアップとも呼ばれ、バージョン番号の上2桁目の変更 (9.x.x から 9.y.y) を行うことです。
  - 実行ファイルの入れ替えだけでなく、データファイルの移行が必要です。
  - PostgreSQLを利用するアプリケーション（プログラム）を変更する必要があるかもしれません。
- 手順
  - データファイル大きく分けて二つの方法があります。
    - pg_dump/pg_dumpall でダンプファイルを作ってリストアする
    - pg_upgradeでインプレイスなアップグレードを行う
  - 手順が厄介ですが、ローリングアップグレードも可能です
    - Slony-I を利用します。

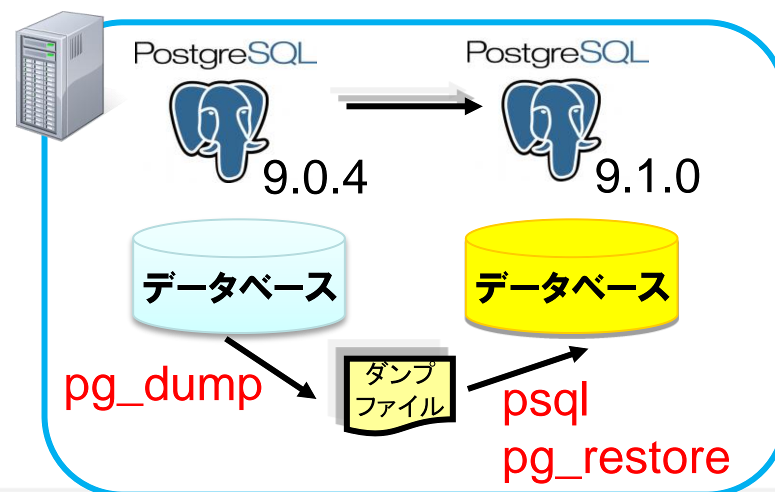




pg_dumpでアップグレードする。

アップデート

- pg_dump/ pg_dumpallをつかったアップグレードとは？
  - 従来からのアップグレード方法です。
  - ダンプファイルを作成し、新しいPostgreSQLにロードします。
- 注意点
  - ダンプ編
    - 管理者権限でdumpしましたか？（ダンプできていないデータはありませんか？）
    - 新しいpg_dumpを利用しましたか？
    - バックアップはとりましたか？
  - ロード編
    - テーブルスペースの移行はしましたか？
    - 設定ファイル(pg_hba.confなど)はリストアしましたか？
    - 新しいpsqlでリストアしましたか？



pg_upgradeでアップグレードする。

アップデート

- pg_upgradeって何？
  - 本体に同梱されているアップグレードツール（Ver.9.0～）です。
    - 昔はpg_migratorと呼ばれていました。
- どのようなアップグレードに使うのか？
  - 巨大なDB≒おおきなサイズのDBクラスタの移行時です。
    - インプレースなアップグレードが実施可能（コピーモードもあります）。
    - 時間短縮・ディスク容量節約につながります。
- 注意点
  - 一部制約（ロケール・コンパイルオプション）があります
    - 詳細は公式ドキュメントを参考のこと

バージョン別のツール選択

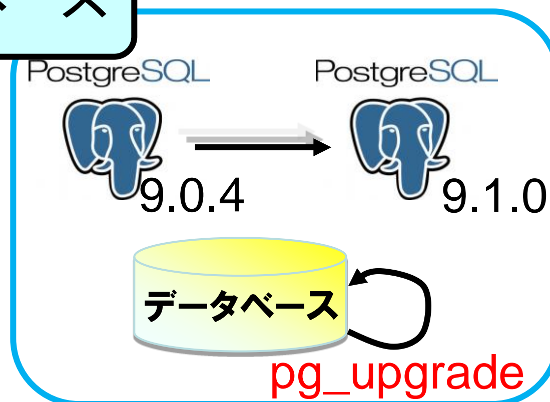
古＼新	8.3	8.4	9.0～
8.3	-	pg_migrator	pg_upgrade
8.4	-	-	pg_upgrade
9.0～	-	-	pg_upgrade

早い

150GB、850テーブルの移行時

移行方法	所要時間（分）
dump + restore(通常の方法)	300
dump + パラレルrestore	180
pg_upgrade (コピーモード)	44
pg_upgrade (リンクモード)	0.7

省スペース



- メンテナンスに関する運用テクニックを紹介します
  - メンテナンスを知る。
  - 自動VACUUM不足にならないようにする。
  - VACUUMを手早く終わらせる。
  - 統計情報を正確に取得する。

メンテナンスを知る。

メンテナンス

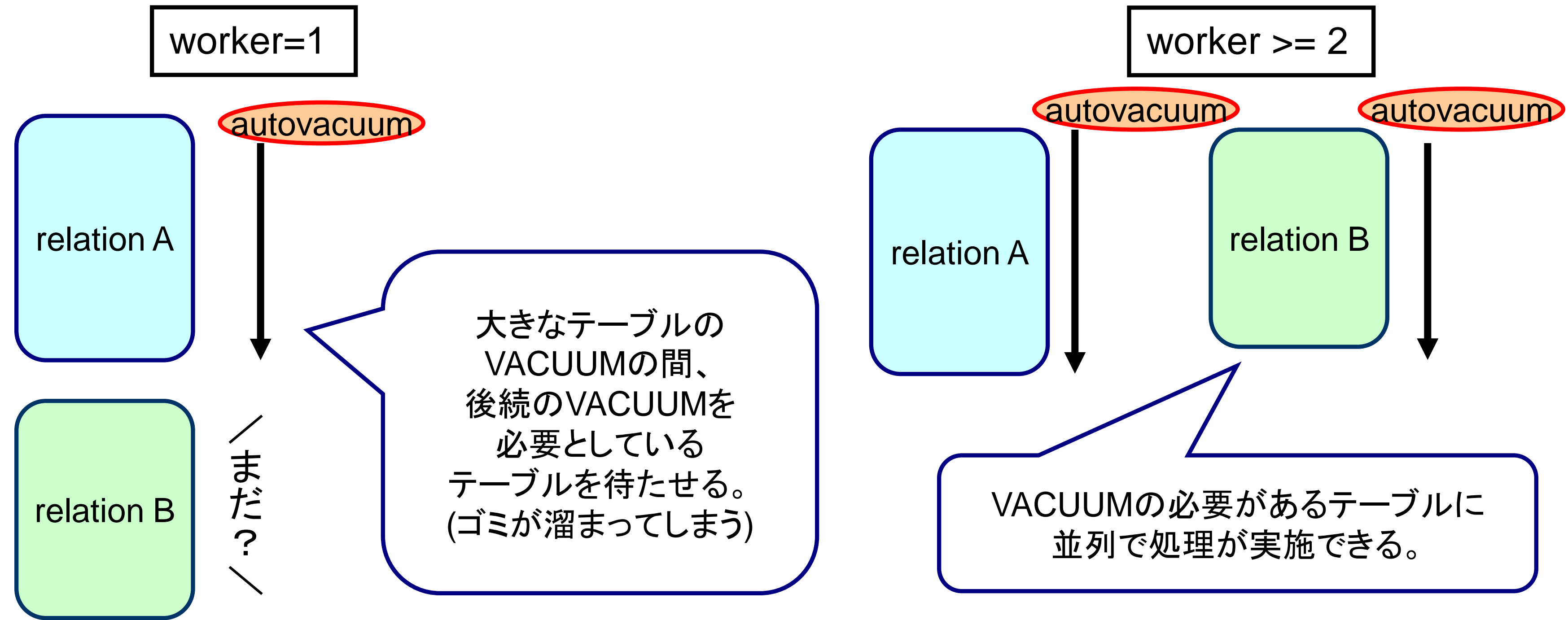
- メンテナンスとしてどんな作業があるかを知ろう
  - Let's PostgreSQLの記事が網羅的です。
    - http://lets.postgresql.jp/map/operation
  - メンテナンス
    - 日々の運用により内部状態が変化していきます。常に一定のパフォーマンスを発揮するには、良い状態を保つためのメンテナンスが必要です。
    - 主に VACUUM や ANALYZE が該当します。

頻度	メンテナンス	目的	方針
日単位	VACUUM	不要領域による肥大化と断片化の防止 トランザクションID周回回避	自動vacuumに任せる
	ANALYZE	統計情報の最新化	自動vacuumに任せる
月単位～	REINDEX	索引の再構築	適切に自動vacuumを実施していれば不要。性能監視等で必要になった場合にのみ実施。
	CLUSTER	表の再編成・索引の再構築	
	VACUUM FULL	表の物理サイズの圧縮	
不定期	フェイルオーバ	サービスの可用性向上	-
	再起動	OSのハングやOOMキラー時など	必要に応じて実施

自動VACUUM不足にならないようにする。

メンテナンス

- 何を設定する？
  - autovacuum_max_workers
  - 同時に実行されるautovacuumプロセスの数
- 設定方針は？
  - 大きなテーブル (over 20% of DB size)の数 + 1
- 影響は？
  - (小さいと) VACUUMが必要なテーブルに長時間VACUUMされない

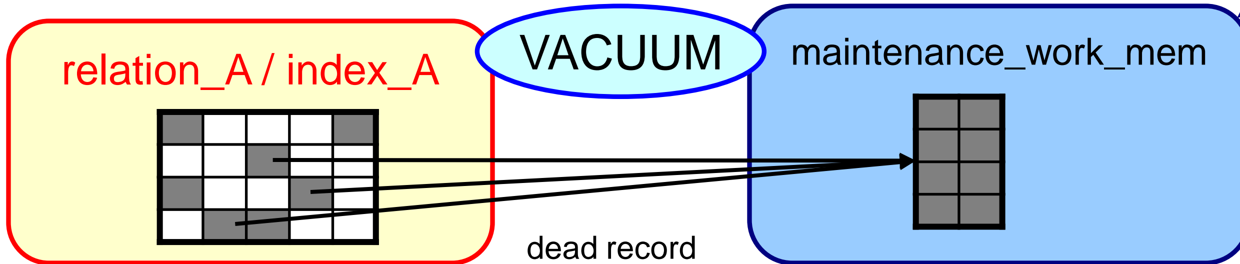


VACUUMを手早く終わらせる。

メンテナンス

- 何を設定する ？
  - maintenace_work_mem
  - VACUUMやREINDEX用の作業メモリサイズ
- 設定方針は？
  - 最も行数の多いテーブルの行数 * 0.2 * 6 (vacuum実施間隔でテーブルの2割が更新されると仮定)
- 影響は？
  - (小さいと) vacuum 速度低下
  - (大きいと) メモリリソース消費

dead record 1つあたり6byteを消費
16MB(def)で約280万行までOK



VACUUM処理の流れ
1. dead record を読み取り maintenance_work_memへ列挙
2. dead record に対応する index を掃除
3. dead record を掃除

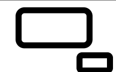
！maintenace_work_mem が小さいと、1-3を何度も繰り返すので遅くなる
VACUUM実施間隔で発生するガベージの分を保持できるのが理想

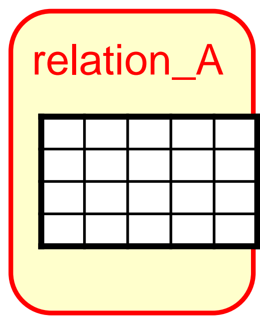
統計情報を正確に取得する。

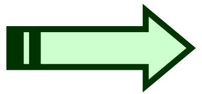
メンテナンス

- 何を設定する？
  - default_statistics_target
  - ANALYZEでのサンプリング数
- 設定方針は？
  - LIKE検索があるカラムは100
- 影響は？
  - (小さいと) 適切な実行計画が作成されない
  - (大きいと) ANALYZEが長時間に及ぶ(ロングトランザクション問題)

SELECT…. FROM relation_A WHERE col LIKE '…..';
→ LIKE検索性能が☹ or 巨大なテーブルへの検索性能が☹







ANALYZE

サンプリング数はカラム単位で調節可能！

```
ALTER TABLE relation_A ALTER COLUMN col
SET STATISTICS 100;
  or
SET default_statistics_target TO '100';
```