SRA OSS, INC.

LPI-JAPAN
Linux Professional Institute Japan

『OSS-DB Exam Gold 技術解説無料セミナー』

@東京 9/5（土）

SRA OSS, Inc 日本支社

マーケティング部

松坂 大地

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

OSS-DB /Silver

データベースシステムの設計・開発・導入・運用ができる技術者

OSS-DB /Gold

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

OSS-DB技術者認定資格の必要性

商用/OSSを問わず様々なRDBMSの知識を持ち、データベースの構築、運用ができる、または顧客に最適なデータベースを提案できる技術者が求められている

OSS-DB /Gold

出題範囲

運用管理 30%

- データベースサーバ構築　重要度：2
- **運用管理用コマンド全般　重要度：4**
- データベースの構造　重要度：2
- ホット・スタンバイ運用　重要度：1

性能監視 30%

- **アクセス統計情報　重要度：3**
- テーブル / カラム統計情報　重要度：2
- クエリ実行計画　重要度：3
- その他の性能監視　重要度：1

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　重要度：4
- チューニングの実施　重要度：2

障害対応 20%

- **起こりうる障害のパターン　重要度：3**
- **破損クラスタ復旧　重要度：2**
- ホット・スタンバイ復旧　重要度：1

目次

- 運用管理コマンド　- バックアップ -
  - 物理バックアップ と PITR（Point In Time Recovery）
  - 論理バックアップ
- 運用管理コマンド　- メンテナンス -
  - VACUUM, ANALYZE, autovacuum
  - インデックスの再構築
- アクセス統計情報
  - 統計情報ビュー、ロック情報の取得
- 起こりうる障害のパターン

運用管理コマンド

- PostgreSQL のバックアップ -

PostgreSQL 物理バックアップ（1）

- 物理バックアップ
  - PostgreSQLを停止して、$PGDATA を物理的にバックアップ
    - tar、cp、rsync などディレクトリを扱う任意のツールを使用

  Point!!

    - テーブルスペースを使っている場合、そのディレクトリもコピーが必要
- pg_start_backup(), pg_stop_backup()
  - PostgreSQL を停止せずに物理バックアップするための関数
    - pg_start_backup() を実行してから、pg_stop_backup() が実行される間は
      $PGDATA を物理的にバックアップが取得できる
    - PITR のベースバックアップ取得のために登場

  Point!!

    - pg_start_backup() は CHECKPOINT を発行
    - pg_stop_backup() はトランザクションログ格納領域に .backup ファイルを作成し、
      バックアップのトランザクションログ開始位置、終了位置、バックアップ開始時刻、終了時刻を記録

```
$ ls /mnt/archive_log
000000010000000000000004
000000010000000000000004.00E81AAC.backup
000000010000000000000005
```

PostgreSQL 物理バックアップ（2）

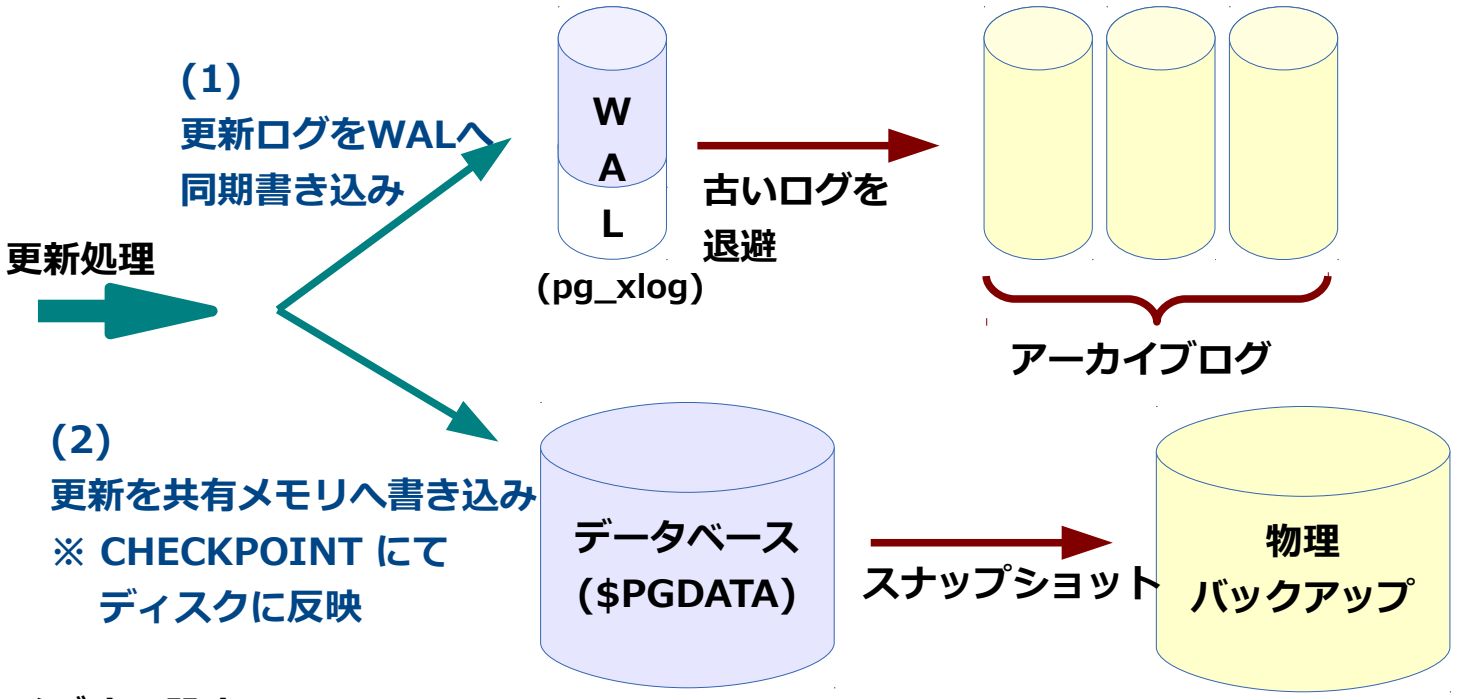
- pg_basebackup
  - 9.1 に追加された $PGDATA を物理バックアップするためのコマンド
  - PostgreSQL 接続を経由して、レプリケーションプロトコルを利用しバックアップを取得
    利用にはストリーミングレプリケーション稼働系の設定が必要
  - PITR のベースバックアップ取得やストリーミングレプリケーションの待機系構築で利用される

```
$ pg_basebackup -h primary_server -p 5432 -U postgres -D ./backup -R -P
59481/59481 kB (100%), 1/1 tablespace
$ ls backup
PG_VERSION     pg_dynshmem    pg_multixact  pg_stat       pg_xlog
backup_label   pg_hba.conf    pg_notify     pg_stat_tmp   postgresql.auto.conf
base           pg_ident.conf  pg_replslot   pg_subtrans   postgresql.conf
global         pg_log         pg_serial     pg_tblspc     recovery.conf
pg_clog        pg_logical     pg_snapshots  pg_twophase
```

  - -h, -p バックアップしたいDBサーバが稼働しているホスト名とポート番号を指定
  - -D 取得したバックアップを配置するディレクトリを絶対パスまたは相対パスで指定
  - -R recovery.conf の雛形を配置
  - -P バックアップの進捗情報を表示

Point In Time Recovery（1）

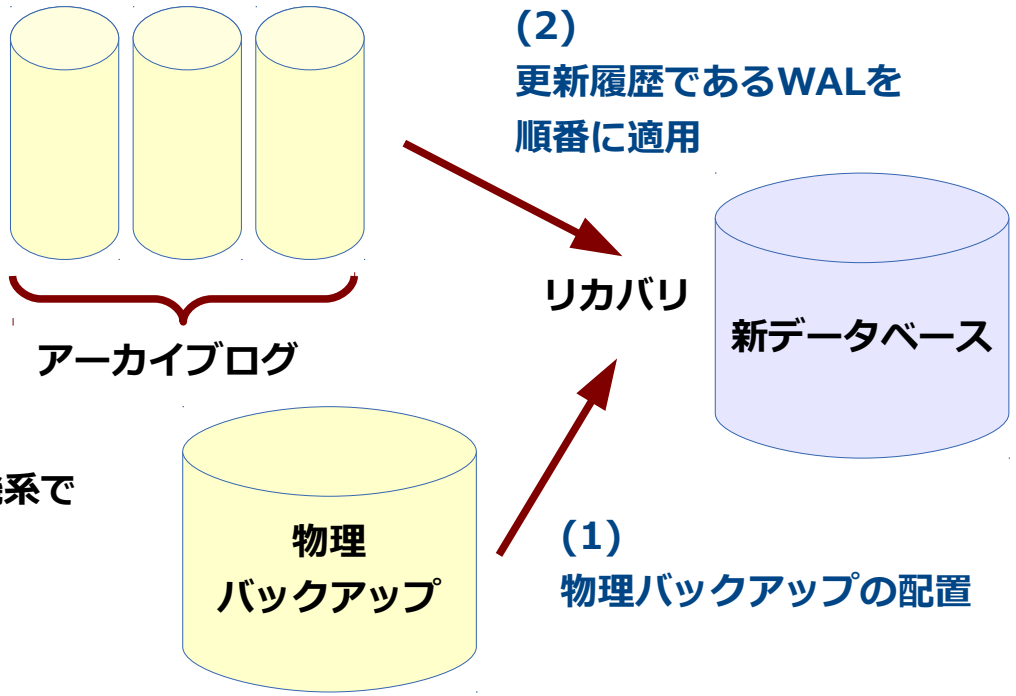
- アーカイブログを用いたバックアップ



- 準備
  - WAL をアーカイブする設定
    - archive_mode, archive_command
  - ベースバックアップの取得
    - pg_basebackup または pg_start_backup(), pg_stop_backup() を利用

Point In Time Recovery（2）

- 障害発生時のリカバリ手順

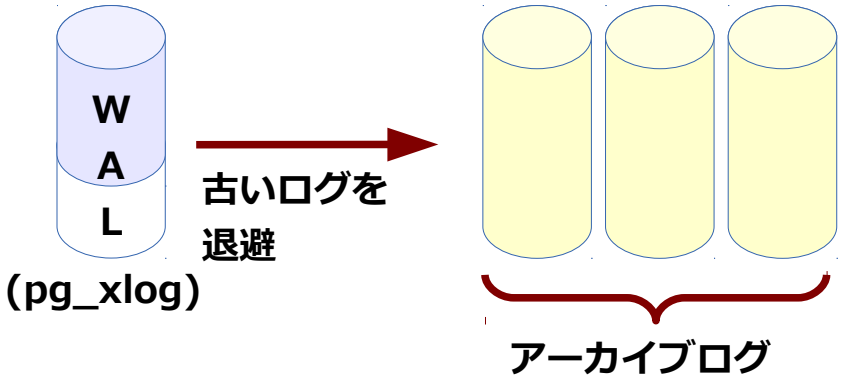


- recovery.conf
  - PITRリカバリ時や
    ストリーミングレプリケーション待機系で
    利用する設定ファイル
  - $PGDATA に配置して起動すると、
    リカバリモードで起動
- PITR リカバリ時の設定項目
  - restore_command
    アーカイブされたWALを適用するためのコピーコマンドを指定
  - recovery_target_time, recovery_target_xid
    指定した時刻までリカバリを実施、指定したトランザクションIDまでリカバリ実施

Point In Time Recovery（3）

Point!!

- 障害直前までの完全な復旧には
  pg_xlog 内のログが必要



- アーカイブログは溜まり続ける
  - 定期的にベースバックアップを取り直し、
    .backup より古いファイルは削除してよい

...00004 で取得した
ベースバックアップにとっては
...00003 以前のログは不要

```
$ ls /mnt/archive_log
000000010000000000000002
000000010000000000000003
000000010000000000000004
000000010000000000000004.00E81AAC.backup
000000010000000000000005
```

- Hash インデックスは WAL ログ対象外
  - PITR によるリカバリ後は REINDEX が必要
- テーブルスペースを使用している場合、リカバリ時も同じディレクトリ構成にする必要がある

PostgreSQL 論理バックアップ（1）

- pg_dump
  - データベース稼働中も一貫性のあるバックアップを取得できるコマンド
  - PostgreSQL メジャーバージョンアップ時は論理バックアップとリストアにて行う
  - オプション指定により、バックアップ対象を柔軟に指定できる
- -Fp テキスト形式
  - SQL文によるテキストファイルでのバックアップ
  - リストアは psql コマンドの -f オプション

```
$ pg_dump database -f backup.sql
$ createdb newdb
$ psql newdb -f backup.sql
```

- -Fc / -Ft / -Fd バイナリ形式
  - カスタム形式、ディレクトリ形式は圧縮を行う
  - ディレクトリ形式は -j オプションで並列実行が可能
  - リストアは pg_restore コマンド

```
$ pg_dump database -Fc -f backup.dump
$ createdb newdb
$ pg_restore backup.dump -d newdb
```

PostgreSQL 論理バックアップ（2）

- pg_restore
  - pg_dump のテキスト以外の形式で取得したバックアップファイルを使ってリストアを行うコマンド
  - オプションが豊富で柔軟にリストア対象の指定が可能
    - `-s  --schema-only`
      スキーマ（データ定義）だけをリストア
    - `-t  --table=`*`tablename`*
      リストアしたいテーブルを指定（複数指定可）
    - `-l  --list`
      バックアップ内容を一覧として出力
    - `-j` *`num`* `--jobs=`*`num`*
      指定された数で並列実行
  - リストア先データベースを指定しない場合、テキスト形式に変換

```
$ pg_restore backup.dump > sql.txt
```

PostgreSQL 論理バックアップ（3）

- pg_dumpall

```
$ pg_dumpall -f dumpall.sql
```

  - テキスト形式のみのバックアップコマンド
  - バイナリ形式で行いたい場合
    - グローバルオブジェクトを pg_dumpall データベース単位で pg_dump

```
$ pg_dumpall -g > globals.sql
$ pg_dump -Fc db1 > db1.dump
$ pg_dump -Fc db2 > db2.dump
```

  - リストア
    - データベースクラスタ初期化
    - 個々のデータベースをリストア

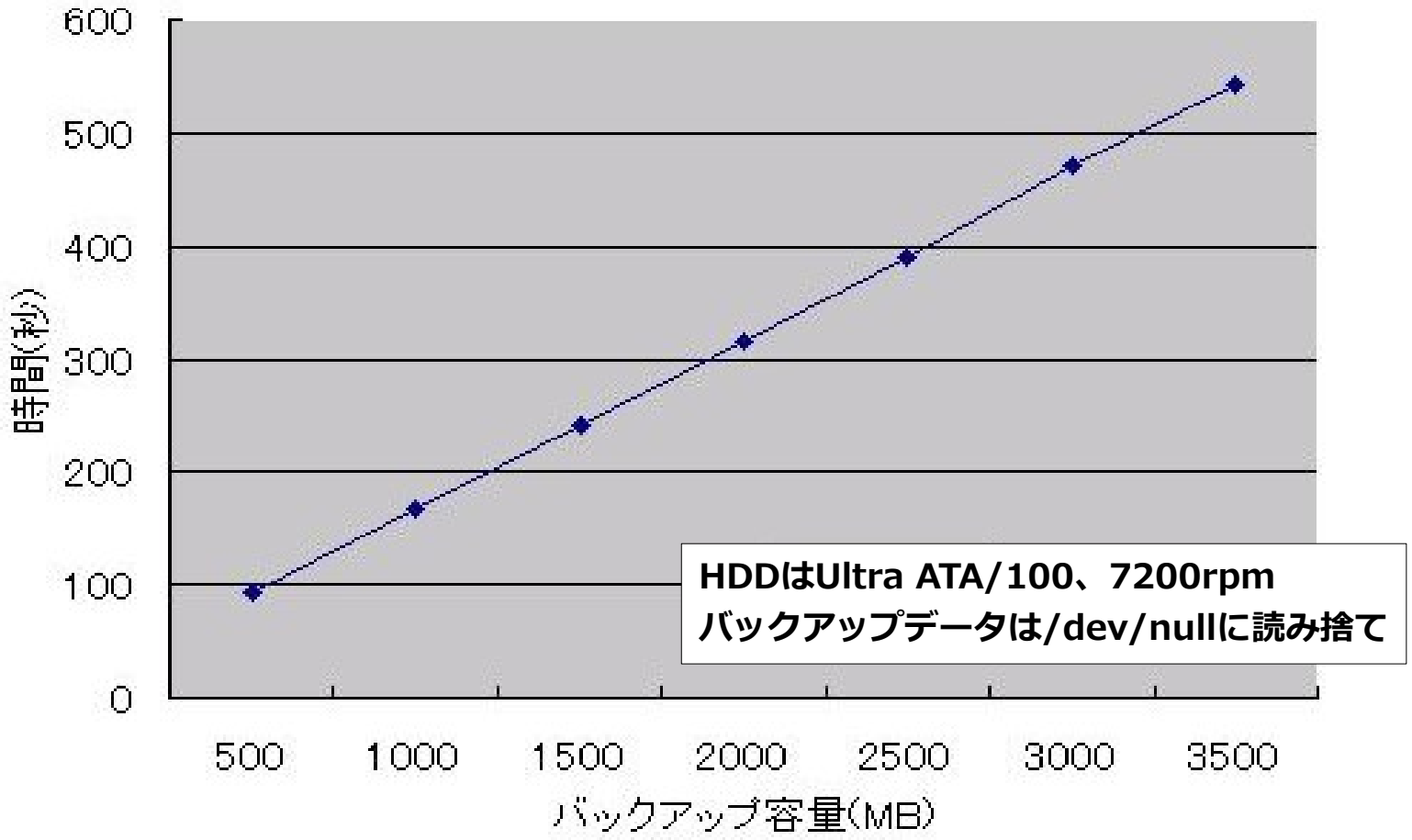
```
$ initdb --no-locale -E UTF8
$ pg_ctl start
$ psql -f globals.sql template1
$ createdb db1
$ pg_restore -d db1 db1.dump
$ createdb db2
$ pg_restore -d db2 db2.dump
```

PostgreSQL 論理バックアップ（4）

- pg_dump の注意点
  - バックアップ中に以下のことを行わない
    - DDL操作
    - ラージオブジェクトのデータを書き換える
  - バックアップされないもの
    - postgresql.conf, pg_hba.conf などの設定ファイル
- pg_dump の所要時間
  - テーブルのデータ量に比例
  - 一般的にリストアのほうが2倍以上時間がかかる

PostgreSQL 論理バックアップ（5）

- 参考バックアップ時間（pg_dump）



運用管理コマンド

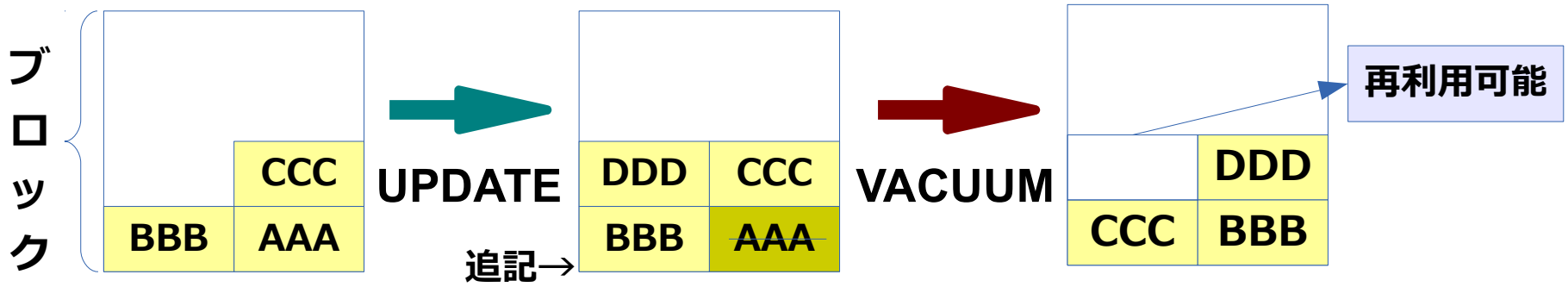
- PostgreSQL のメンテナンス -

VACUUM（1）

PostgreSQL は追記型アーキテクチャ

```
UPDATE tb1 SET val = 'DDD' WHERE val = 'AAA';
```

テーブルファイルの不要領域



ブロック単位で有効データを再配置

VACUUM（2）

- 通常の VACUUM（コンカレントVACUUM）
  - 運用を妨げずに並行して実行可能
  - データベース領域のファイルサイズは基本的に不変

Point!!

- ファイルが切り詰められて小さくなるわけではない
- 例外としてファイル末尾の不要タプルのみのブロックは縮小される
- ビジビリティマップにて不要領域があるブロックを管理することで VACUUM は高速に動作（8.3以前のVACUUMはテーブル全スキャン）

VACUUM（3）

- VACUUM FULL
  - 実行中はテーブルを完全にロック
    - ACCESS EXCLUSIVE ロック（一番強いロック）を取得
    - 実質サービスが停止する
  - テーブルファイルを新しく作り直す
    - 空き領域を詰めてサイズを小さくする
    - テーブルファイルと同程度の空き領域が必要になる
      （ディスクフル間近のケースで救えない）

VACUUM（4）

- VACUUMをいつ行うか？
  - 自動 VACUUM で運用 (デフォルト)
  - 自動 VACUUM を利用しない場合には
    - 最低一日一回、全データベースに対して実行
    - 更新頻度によっては、個別テーブルに対して特別に頻度を上げる

Point!!

  - VACUUM FULL は通常使用しない
    - 長期間 VACUUM が実行されていなかった場合など
      問題が発生したときの応急処置として利用
  - 自動 VACUUM は負荷状況を考慮しない
    - 場合によっては cron などでスケジュール実行

VACUUM（5）

pgstattuple

- タプルレベルの統計情報を確認できるツール
- 不要領域の確認ができる
  - table_len
    リレーションの物理長
  - tuple_len, tuple_percent
    有効タプルのサイズ、割合
  - dead_tuple_len, dead_tuple_percent
    無効タプルのサイズ、割合
  - free_space, free_percent
    空き領域のサイズ、割合
- サイズは pg_size_pretty() を使うと単位がつく

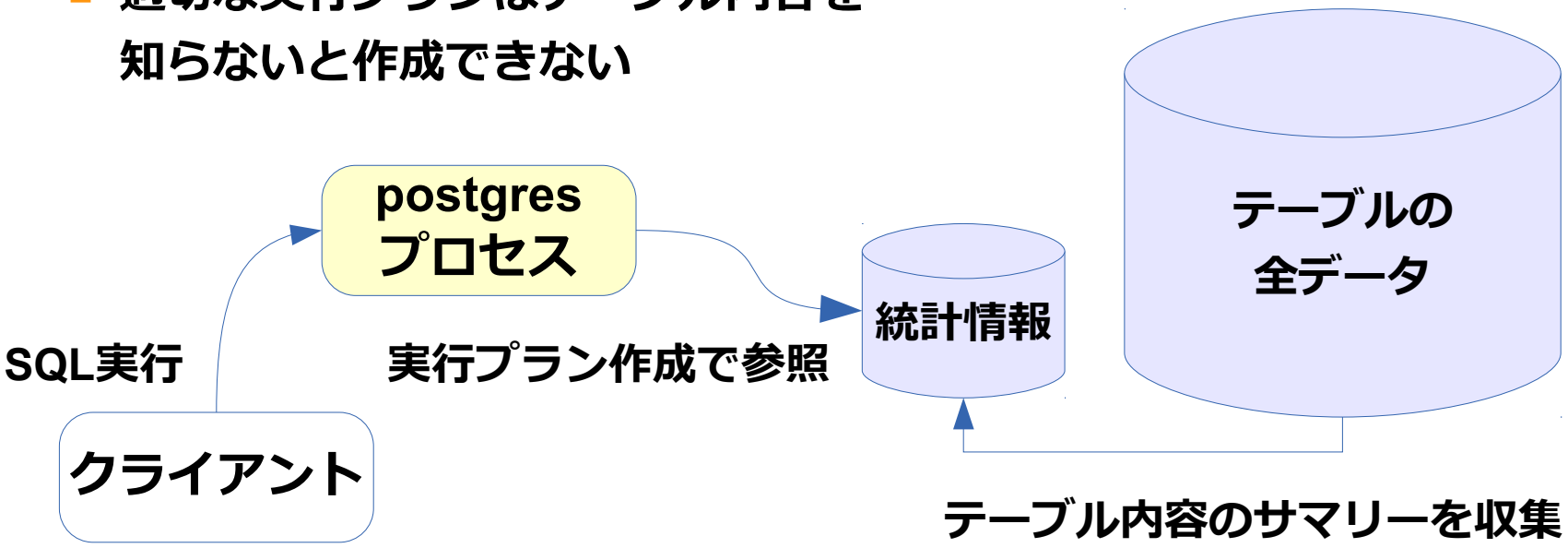
```
=# SELECT * FROM pgstattuple('テーブル名');
-[ RECORD 1 ]------+---------
table_len          | 27385856
tuple_count        | 200000
tuple_len          | 24200000
tuple_percent      | 88.37
dead_tuple_count   | 3305
dead_tuple_len     | 399905
dead_tuple_percent | 1.46
free_space         | 395484
free_percent       | 1.44

=# SELECT pg_size_pretty(table_len)
             FROM pgstattuple('テーブル名');
-[ RECORD 1 ]--+------
pg_size_pretty | 26 MB
```

VACUUM（6）

- その他の VACUUM 効果
  - pg_clog ディレクトリの不要ファイル削除
  - XID 周回問題対策
  - テーブルに関連する統計情報の収集
    - いずれも一日一回実行しておけば問題ない

統計情報（1）

- SQLはデータの操作を指示する言語
  - 実行プラン作成は PostgreSQL に一任されている
    - 適切な実行プランはテーブル内容を
      知らないと作成できない



統計情報（2）

ANALYZE コマンド

- 統計情報を収集するコマンド
- 全スキャンではなくサンプリングによって収集
  - 巨大なテーブルでも時間はかからず、負荷も軽い
  - 毎回収集結果が異なるため、巨大なテーブルではサンプリング数を調整

```
#default_statistics_target = 100
```
統計情報のエントリー数

```
=# ALTER TABLE "テーブル名" ALTER aid SET STATISTICS 300;
```

ANALYZEをいつ行うか？

- 自動 VACUUM で運用 (デフォルト)
  - データ更新頻度を考慮して定期的に実行
- 大量のデータを更新、挿入、削除した後
- CLUSTER コマンドによるクラスタ化の後

autovacuum による VACUUM/ANALYZE

自動 VACUUM

- 更新量や不要領域の割合を監視し、VACUUMを自動実行するデーモンプロセス

`autovacuum launcher`	不要領域を監視するプロセス
`autovacuum worker`	自動VACUUMを実行するプロセス（同時に複数起動する）

- デフォルトで有効
- しきい値
  - scale_factor で指定

    VACUUM 20%の不要領域
    ANALYZE 10%の更新
  - 変更はテーブル単位で

```
#autovacuum = on
#log_autovacuum_min_duration = -1
#autovacuum_max_workers = 3
#autovacuum_naptime = 1min
#autovacuum_vacuum_threshold = 50
#autovacuum_analyze_threshold = 50
#autovacuum_vacuum_scale_factor = 0.2
#autovacuum_analyze_scale_factor = 0.1
```

```
=# ALTER TABLE "テーブル名" SET ( autovacuum_vacuum_scale_factor=0.05,
                                  autovacuum_analyze_scale_factor=0.02);
```

\d+ で設定確認

VACUUM/ANALYZE の手動実行

vacuumdb コマンド

```
$ vacuumdb -az
```

```
$ vacuumdb database_name
```

- -a 全データベースにVACUUM
- -f VACUUM FULLを実施
- -t テーブル指定
- -v 詳細なメッセージを出力
- -z ANALYZEも一緒に実施
- -Z ANALYZEのみを実行

- 通常は -az を cron に登録して実行
- 更新頻度の多いテーブルは更に短い間隔で事項

SQL による実行

`=# VACUUM;`	接続データベース全体を VACUUM
`=# ANALYZE;`	接続データベース全体を ANALYZE
`=# VACUUM table_name;`	テーブルを指定
`=# VACUUM FULL;`	接続データベース全体を VACUUM FULL

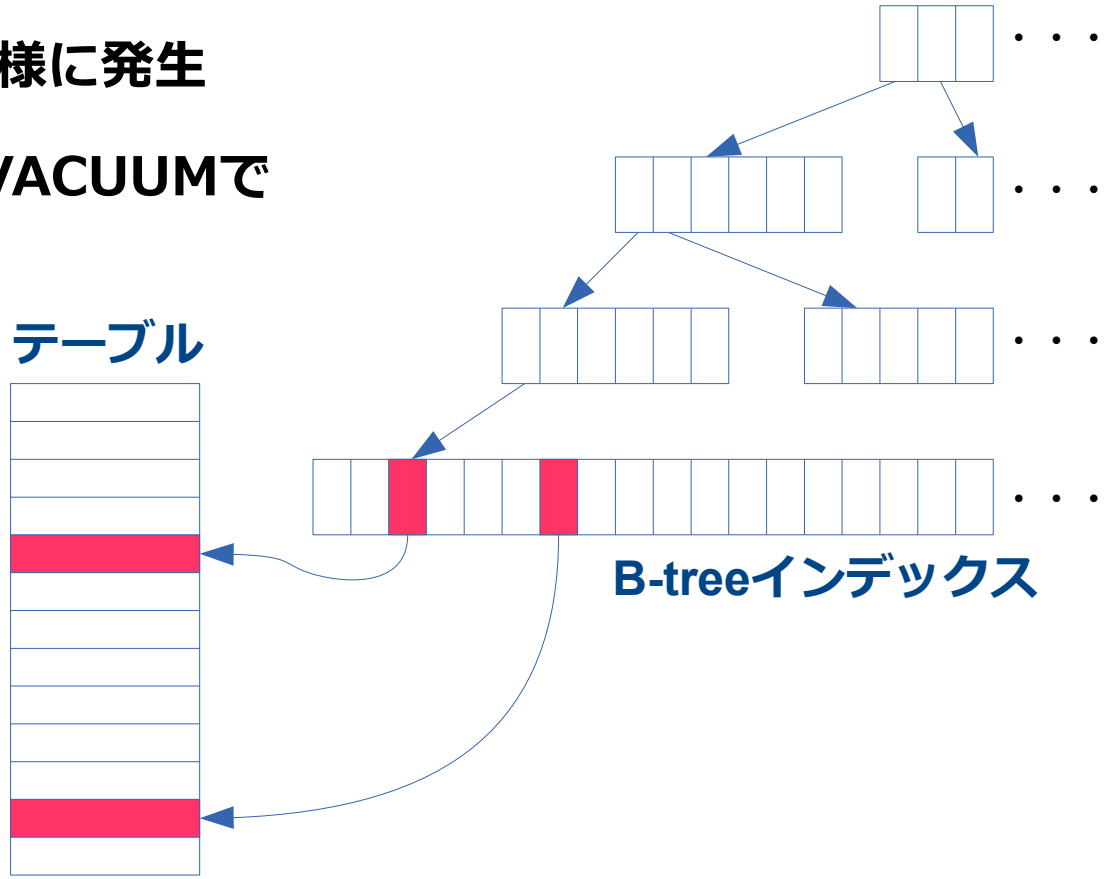
インデックスの再構築（1）

インデックスの不要領域

- テーブルの不要領域と同様に発生
- B-tree インデックスはVACUUMで再利用される

Point!!

- ページ単位での回収
  - テーブルほど効率的ではない
  - 時にはインデックスを作り直す必要がある



インデックスの再構築（2）

- インデックスの再構築をいつ行うか？
  - インデックスに不要領域がたくさんある場合
    - 構築後にたくさんの行を更新、削除した
  - インデックスが破損したとき
    - Hashインデックスは、クラッシュリカバリ時に破損することがある
  - pgstatindex() を使ってインデックスの断片化を確認
    - contrib モジュール pgstattuple に含まれる関数
    - B-tree インデックスの断片化状況を確認できる

インデックスの再構築（3）

インデックスの断片化

pgstatindex

- index_size
  インデックスの総サイズ
- avg_leaf_density
  リーフページの平均密度
- leaf_fragmentation
  リーフページの断片化

- autovacuum によりインデックスの不要領域も回収されるが断片化は進む
- 断片化が進むと、インデックスサイズが大きくなり共有メモリに乗りづらくなる
  ⇒ Index Scan が選択されにくくなる

```
=# SELECT * FROM pgstatindex('インデックス名');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 8192
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 0.54
leaf_fragmentation | 0
```

作成直後

大量の更新

```
index_size         | 184 kB
avg_leaf_density   | 65.41
leaf_fragmentation | 45.45
```

REINDEX

```
index_size         | 8192 bytes
avg_leaf_density   | 0.54
leaf_fragmentation | 0
```

インデックスの再構築 方法（1）

- “DROP INDEX” → “CREATE INDEX”
  - いったんインデックスを削除してから再構築

```
=# DROP INDEX インデックス名;
=# CREATE INDEX インデックス名;
```

  - DROP 中は読み書きをブロック
  - CREATE 中は書き込みをブロック、INDEX を使わない読み込み可
    - プランナが index scan を選択するとブロックされる
  - バッチ処理で大量のデータ更新を行う場合向き

- REINDEX
  - REINDEX 中は書き込みをブロック
  - 対象のインデックスを使う検索もブロックされる

```
=# REINDEX INDEX インデックス名;
```

```
=# REINDEX INDEX インデックス名;
```

インデックスの再構築 方法（２）

CREATE INDEX CONCURRENTLY

- CREATE INDEX CONCURRENTLYで新インデックスを作成
- 新インデックス作成後、旧インデックスを削除する

```
=# CREATE INDEX CONCURRENTLY [新インデックス名] ON table [オプション];
=# DROP INDEX 旧インデックス名;
```

- DROP INDEX中の一瞬の読み書きロック
- 作成中は旧インデックスが利用されるため読み込みも可
- テーブルを2回全スキャンするため作成速度が遅い

Point!!

- 制約違反により失敗する場合がある
  - 失敗したインデックスは削除して再構築

```
=# \d tab
        Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col) INVALID
```

ディスク容量の監視

- 運用中にディスク容量不足にならないように監視する
  - OSの一般的なツールで定期的にチェック
    - グラフ化して傾向がわかるようにしておくと良い
  - 増加速度を把握しておく
    - 行の増加速度
    - アーカイブログの追加速度
- VACUUM 不足によるテーブル肥大化が起きていないか？
  - VACUUM 不足による肥大化は顕著に現れる

アクセス統計情報

統計情報コレクタ

- stats collector プロセスよって収集される統計情報
  - データベースの動作状況に関する情報
  - 収集された統計情報はビューや関数によって参照できる
- 統計情報の収集はデフォルトで有効
  - `track_counts = on`
    - データベースの活動についての統計情報の収集
    - 自動バキュームを使用するためにも有効に設定する必要がある
  - `track_activities = on`
    - 実行中のコマンドに関する情報を収集
- データベースごとに統計情報をリセットするには
  `pg_stat_reset`関数を実行

```
=# SELECT pg_stat_reset();
```

統計情報ビュー　- pg_stat_activity -

実行中のSQLに関する統計情報ビュー

track_activities = on

9.4 の出力例

```
postgres=# SELECT * FROM pg_stat_activity ;
-[ RECORD 1 ]----+----------------------------------------------------------------------
datid            | 13056
datname          | postgres
pid              | 13446
usesysid         | 10
usename          | postgres
application_name | pgbench
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2015-08-24 13:00:53.162677+09
xact_start       | 2015-08-24 13:01:27.394888+09
query_start      | 2015-08-24 13:01:27.395179+09
state_change     | 2015-08-24 13:01:27.395366+09
waiting          | f
state            | active
backend_xid      | 5126
backend_xmin     |
query            | UPDATE pgbench_tellers SET tbalance = tbalance + -2741 WHERE tid = 19;
  :
-[ RECORD 2 ]----+----------------------------------------------------------------------
(snip)
```

統計情報ビュー　- pg_stat_activity -

- 実行中のSQLに関する統計情報ビュー
  - pid(proc_pid), query(current_query)
    - 実行中の SQL とプロセス ID の紐付け
  - waiting
    - ロック待ちの状態かどうか
  - backend_start, xact_start, query_start
    - プロセス開始日時、トランザクション開始日時、SQL実行の開始日時
  - client_addr
    - クライアントの IP アドレス
    - UNIX ドメイン接続の場合は NULL

9.2 で名称変更

統計情報ビュー　- pg_stat_database -

データベースに関する統計情報ビュー

- キャッシュヒット率

$$\frac{\text{blks\_hit}}{(\text{blks\_hit} + \text{blks\_read})}$$

- tup_returned
  - シーケンシャルスキャンにより取得された行数
- tup_fetched
  - インデックススキャン、ビットマップスキャンにより取得された行数

```
postgres=# SELECT * FROM pg_stat_database;
 :
-[ RECORD 3 ]--+------------------------------
datid          | 13056
datname        | postgres
numbackends    | 1
xact_commit    | 310468
xact_rollback  | 12
blks_read      | 8553
blks_hit       | 8798591
tup_returned   | 140531047
tup_fetched    | 1692340
tup_inserted   | 206106
tup_updated    | 17815
tup_deleted    | 0
conflicts      | 0
temp_files     | 0
temp_bytes     | 0
deadlocks      | 0
blk_read_time  | 0
blk_write_time | 0
stats_reset    | 2015-05-08 10:17:25.918014+09
```

統計情報ビュー　- pg_stat_user_tables -

- テーブルに関する統計情報ビュー（行単位）
  - n_live_tup / n_dead_tup
    - 有効行数、無効行数
  - n_tup_hot_upd
    - HOT により更新された行数
  - last_vacuum など
    - 最後に VACUUM や ANALYZE が実行された日時
  - vacuum_count など
    - 最後にリセットしてからの実行回数
    - 各テーブルに対して定期的に VACUUM や ANALYZE が実行されていることを確認

```
postgres=# SELECT * FROM pg_stat_user_tables;
-[ RECORD 1 ]-------+------------------------------
relid               | 16404
schemaname          | public
relname             | pgbench_branches
seq_scan            | 5932
seq_tup_read        | 11864
idx_scan            | 0
idx_tup_fetch       | 0
n_tup_ins           | 2
n_tup_upd           | 5930
n_tup_del           | 0
n_tup_hot_upd       | 5930
n_live_tup          | 2
n_dead_tup          | 0
n_mod_since_analyze | 0
last_vacuum         | 2015-08-24 13:00:53.132605+09
last_autovacuum     | 2015-08-24 13:01:57.907927+09
last_analyze        | 2015-08-24 13:00:39.394919+09
last_autoanalyze    | 2015-08-24 13:01:57.914395+09
vacuum_count        | 2
autovacuum_count    | 2
analyze_count       | 1
autoanalyze_count   | 2
```

統計情報ビュー　- pg_stat_user_indexes -

- インデックスに関する統計情報ビュー（行単位）
  - idx_scan
    - インデックススキャンが実行された回数

```
postgres=# SELECT * FROM pg_stat_user_indexes ;
-[ RECORD 1 ]-+----------------------
relid         | 16404
indexrelid    | 16408
schemaname    | public
relname       | pgbench_branches
indexrelname  | pgbench_branches_pkey
idx_scan      | 321
idx_tup_read  | 321
idx_tup_fetch | 321
```

- 行単位の情報
  - pg_stat　_all_tables, _user_tables, _sys_tables
  - pg_stat　_all_indexes, _user_indexes, _sys_indexes
- ブロック単位の情報
  - pg_statio　_all_tables, _user_tables, _sys_tables
  - pg_statio　_all_indexes, _user_indexes, _sys_indexes
    - ブロック＝8kB がデフォルト

統計情報ビュー　- 比較的新しいもの -

- pg_stat_replication
  (スタンバイサーバへのレプリケーションに関する統計情報)
  - sent_localtion, write_location, flush_location, replay_location
    - スタンバイサーバに送信された、書き込んだ、同期書き込みした、再生された
      トランザクションログ位置
  - sync_state
    - スタンバイサーバの同期状態（同期/非同期 が確認可能）
  - aplication_name
    - スタンバイサーバの recovery.conf にて指定したスタンバイサーバの名称が確認できる

9.1 で追加

- pg_stat_archive
  (WALファイルのアーカイブ状況に関する統計情報)
  - アーカイブ数やアーカイブ失敗数、時刻などが確認できる

9.4 で追加

システムカタログ　- pg_locks -

- 実行中トランザクションにより獲得されたロックに関する情報
  - locktype
    - relation（リレーション全体）
    - page（リレーション内のページ）
    - tuple（ページ内のタプル）
    - transactionid（トランザクション）
    - virtualxid（仮想トランザクション）
    - など
  - mode
    - ロックモード　※次ページに表
  - granted
    - ロックを保持しているか（true/false）

```
=# SELECT locktype, database, relation, ↵
transactionid, pid, mode, granted FROM ↵
pg_locks;
-[ RECORD 1 ]-+-----------------
locktype      | transactionid
database      |
relation      |
transactionid | 3745
pid           | 27958
mode          | ExclusiveLock
granted       | t
-[ RECORD 2 ]-+-----------------
locktype      | relation
database      | 16384
relation      | 16402
transactionid |
pid           | 27947
mode          | AccessShareLock
granted       | t
```

ロックモード

ロックモードの競合関係

”ACCESS EXCLUSIVE” は全てのモードと競合

ロックモード（強→弱）	ACCESS EXCLUSIVE	EXCLUSIVE	SHARE ROW EXCLUSIVE	SHARE	SHARE UPDATE EXCLUSIVE	ROW EXCLUSIVE	ROW SHARE	ACCESS SHARE
ACCESS EXCLUSIVE	●	●	●	●	●	●	●	●
EXCLUSIVE		●	●	●	●	●	●	●
SHARE ROW EXCLUSIVE			●	●	●	●	●	●
SHARE			●	●		●	●	●
SHARE UPDATE EXCLUSIVE				●	●	●	●	●
ROW EXCLUSIVE					●	●	●	●
ROW SHARE							●	●
ACCESS SHARE								●

ACCESS SHARE / ROW SHARE / ROW EXCLUSIVE / SHARE UPDATE EXCLUSIVE / SHARE / SHARE ROW EXCLUSIVE / EXCLUSIVE / ACCESS EXCLUSIVE

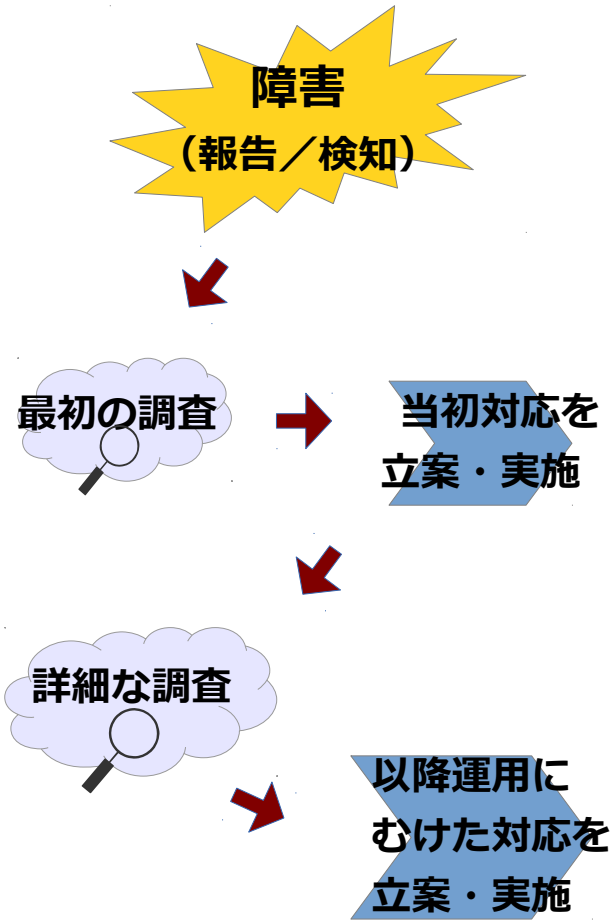
- ALTER TABLE, DROP TABLE TRUNCATE, REINDEX CLUSTER, VACUUM FULL
- CREATE INDEX
- VACUUM, ANALYZE CREATE INDEX CONCURRENTLY
- UPDATE, DELETE, INSERT
- SELECT FOR UPDATE / SHARE
- SELECT

● ロックの競合が発生

起こりうる障害のパターン

障害対応の基本的な考え方

- 最初の状態把握
  - クライアント側？ サーバ側？
  - PostgreSQLが主たる問題元？
    従属的に影響を受けた？
  - 問題が起きた後のプロセス状態
    - 対応の緊急度、「今すべきこと」
- 見るべきところ
  - ログメッセージ
    - クライアント、PostgreSQL、OS
  - 各種ツール・機能で情報収集
    - OSコマンド、PostgreSQL機能



起動できない

- postgresサーバの起動に失敗する
  - サーバリソース（マシン・OS）に問題がある
  - リカバリが継続している
  - データベースクラスタの内容が壊れている

大まかに
３パターン

- 確認すべきところ
  - プロセスは立ち上がったのか？
  - エラーメッセージとログ

↓正常に起動しているとは限らない

```
$ pg_ctl start
server starting
```

↓このメッセージがでても正常に起動することの方が多い

```
$ pg_ctl start
pg_ctl: another server might be running; trying to
start server anyway
```

起動できない - サーバリソースの問題 -

- SYSV 共有メモリ不足
  - 前回起動時にリソース解放漏れ
- TCP/IPソケットが作成できない
  - ポート番号重複など
- データベースクラスタにアクセスできない
  - ファイル/ディレクトリのオーナー/読み書き権限
  - SELinux の制限
  - マウントできていない
  - ウィルス対策ソフトウェア等のアクセスと競合

共有メモリ、セマフォを解放する

```
# ipcs
# ipcrm -m 123456
```

```
FATAL: could not open file "〜": Permission denied
```

起動できない - リカバリが継続している -

- 既に起動中のプロセスが存在している
  - リカバリ処理が途中で止まっているケースを疑う
- recovery.conf の設定を見直す
  - データベースクラスタ($PGDATA)に recovery.conf ファイルが置いてあると、PostgreSQL はリカバリモードで起動
    - recovery_command に不備があると、リカバリ処理が終わらない
  - 原因が特定できたら、KILL コマンドにて 停止させる

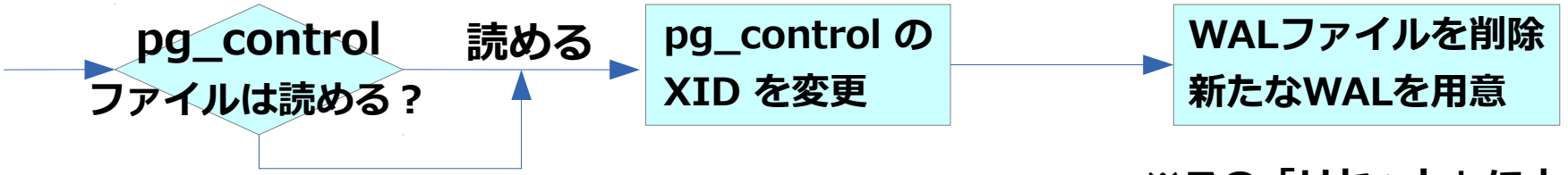
起動できない - データベースクラスタの破損（１） -

WAL が壊れている場合

pg_resetxlog コマンドで修復

引数にデータベースクラスタを指定

```
$ pg_resetxlog $PGDATA
```



※この「リセット」により、
障害直前の書き込み情報が
失われる

-f で適当に修復
※引数や他ファイルを参考に

予めディレクトリの物理バックアップを取る

- pg_resetxlog による変更は元に戻せないため

起動できない - データベースクラスタの破損（2） -

- 管理用ファイルが壊れている場合
  - pg_clog 等の復旧

```
PANIC:  could not access status of transaction 78901
DETAIL:  could not read from file "pg_clog/0123" ・・・
```

    - エラーメッセージで指摘されているファイルがかけ離れている場合
      WALファイルやテーブルファイルの破損を疑う

  - 同サイズのゼロ埋めファイルを設置して起動を試みる

```
$ dd if=/dev/zero of=pg_clog/0123 bs=1024 count=256
$ chmod 600 pg_clog/0123
```

起動できない - データベースクラスタの破損（3） -

- インデックスのファイルが壊れている場合
  - ほとんどの場合 REINDEX で解決

```
=# SELECT * FROM pgbench_accounts WHERE aid = 1;
ERROR:  invalid page header in block 1 of relation base/11874/24638
```

- システムテーブルのインデックスが破損した場合
  - ignore_system_indexes = on
    - システムテーブル読み込み時、システムインデックスを無視
      ただし、テーブル更新に伴うインデックス更新は行う
  - シングルユーザモード

```
$ postgres --single -O -P -D $PGDATA
backend>  REINDEX INDEX pg_opclass_oid_index;
```

Ctrl-D で終了

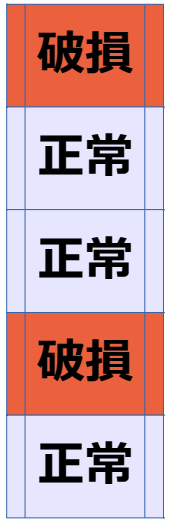
起動できない - データベースクラスタの破損（4） -

- テーブル本体が壊れた場合
  - zero_damaged_pages = on

破損部分はゼロ埋めして
正常部分だけでも読める
ようにするもの

```
=# SELECT * FROM pgbench_accounts LIMIT 100;
ERROR:  invalid page header in block 0 of relation
  base/11874/24654
=# SET zero_damaged_pages TO on;
SET
=# SELECT * FROM pgbench_accounts LIMIT 100;
WARNING:  invalid page header in block 0 of relation
 base/11874/24654; zeroing out page

 aid | bid | abalance |          filler
-----+-----+----------+----------------------------
  64 |   1 |        0 |
  65 |   1 |        0 |
```

テーブルファイル



- 書き込みや VACUUM を行うとゼロ埋めした内容がディスクに反映される
- REINDEX を行ってから設定を戻す（消えたデータをインデックスに反映）

ハングアップ

- 応答しない、停止できない
  - postgresサーバプロセスが接続要求に応答しない
    - pg_ctl stop にも応じない
  - バックエンドプロセスがクライアントに応答しない
- 対応
  - 止まっているのかを確認 （単に遅いのでなく）
    - ロック待ちでないか？

postgres [local] idle in transaction

  - どこで止まっているのかをOSツールで確認
  - 原因情報を採取したらシグナル TERM、QUIT、ABRT、KILL にて止める

ハングアップ - 調査 -

- 原因の調査
  - 一部のバックエンドプロセスのみ応答がない場合は pg_locks や pg_stat_activity を確認

SQLレベルの問題か？

- OS ツールで調査
  - ps
    - どこで止まっているか？
  - strace
    - 動いているか？
  - gdb
    - ソースレベルで今、何を実行している状態かを確認

```
$ strace -p 12345
Process 12345 attached - interrupt to
recv(7,  <unfinished ...>
Process 12345 detached
```

Ctrl-C で detach

```
$ gdb -p 12345
(gdb) bt
(gdb) detach
(gdb) quit
```

「--enable-debug」オプションでのビルドが無いと関数名は表示されない

終了してしまう - PANIC -

- サーバプロセスが PANIC メッセージで終了
  - PostgreSQL全体が終了してしまう
  - 比較的分かりやすいメッセージ
    - アクセスできない、
      リソース不足、など

```
PANIC:  too many semaphores created
PANIC:  could not write to ～
PANIC:  could not seek in ～
PANIC:  could not open ～
```

  - 分かりにくいメッセージの場合
    - インデックスなどデータ内容の破損
    - 本体障害（バグ）

終了してしまう - クラッシュ -

- signal 11: Segmentation fault など
  - PostgreSQLやユーザ定義C関数の障害
  - リリースノートで既知の障害を確認
  - 一つの postgresプロセスがクラッシュすると、
    他の接続も切断して自動的に再起動
  - マスタープロセスのクラッシュは自動再起動しない

```
LOG:  server process (PID 4609) was terminated by signal 11:
Segmentation fault ・・・
WARNING:  terminating connection because of crash of another
server process ・・・
FATAL:  the database system is in recovery mode
```

終了してしまう - メモリ不足 -

- Linux仕様による OOM-Killer
  - 他OSも類似のしくみ

ログ出力例 （syslog）

```
kernel: Out of Memory: Killed process 1234
```

- メモリ不足対応
  - スワップ領域のサイズを増やす
  - 物理メモリを増やす
  - 設定で PostgreSQL が多くメモリを使わないようにする
    - shared_buffers、work_mem、maintenance_work_mem
  - クライアント側（psql、libpq）のメモリ使用に注意
    - ulimit -v

ご清聴ありがとうございました