

PostgreSQL入門 ～クエリ処理編～

Masahiko Sawada

PostgreSQLテクニカルウェビナー

2022/2/16



Agenda

- クエリ処理
- 同時実行制御
- まとめ



本Webinarについて

- ・ PostgreSQL14を元に資料を作成しています
- ・ 本日のWebinarでは、PostgreSQLにおけるクエリ（SQL）処理、同時実行制御について解説します
- ・ 目指すゴール
 - ・ PostgreSQLの用語が理解できる
 - ・ PostgreSQLがどうクエリを処理しているのかが理解できる
 - ・ （簡単な）クエリの実行計画が読める
 - ・ 同時実行制御の仕組みが理解できる

本Webinarでのアプローチ

- ・ 1つのSQLを例に、PostgreSQLがどのように処理し、データを取得するのかを順番に解説
- ・ 必要に応じて少し脱線し、関連する機能や仕組みを紹介
- ・ 最終的には大まかな全体像が掴める
- ・ より詳細についてはAmitさんの以前の講演をご参照ください



```
SELECT * FROM users WHERE id = 100;
```

Table "public.users"

Column	Type	Collation	Nullable	Default
id	integer		not null	
name	text			

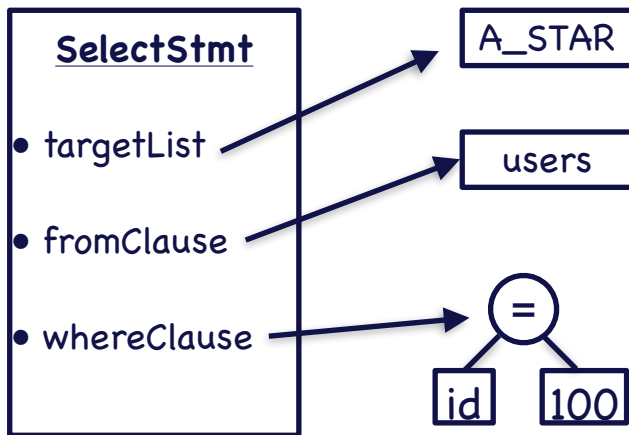
Indexes:

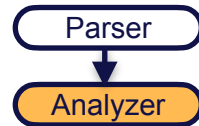
"users_pkey" PRIMARY KEY, btree (id)

SQLを理解する

SELECT * FROM users WHERE id = 100;

- ・ パーサが受け取ったSQLの構文を解析する
 - ・ SELECT文？UPDATE文？文法は正しい？
- ・ ERROR : syntax error at or near “xxxx”
- ・ 文法的に正しければOK
 - ・ テーブルや列が本当にあるかどうかは気にしない





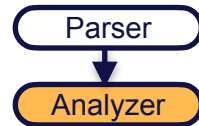
パースツリーを解析する

- アナライザがテーブルや列の存在チェックをする
- ERROR: relation "badtable" does not exist at character 15
- システムカタログを使う
- （リライタ（Rewriter）がクエリの書き換えを行う）

システムカタログ

PostgreSQLのメタデータが詰まった内部テーブル

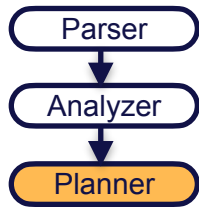
- ・ 実体は通常のテーブルと同じ
- ・ PostgreSQL 14には62個ある
- ・ テーブル、テーブルにある列、インデックス、データ型、演算子、トリガー、関数、ユーザ、型変換など、内部的な情報はすべてシステムカタログにある
- ・ テーブルとしてpg_catalogスキーマに存在するので、SQLでアクセスできる
 - ・ 例) テーブルを検索する時：
 - ・ `SELECT relname FROM pg_class WHERE relname = 'user' and relkind = 'r';`



例) テーブルを作った時

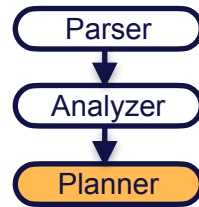
```
CREATE TABLE users (id INT PRIMARY KEY, name TEXT);
```

- テーブル、インデックス(主キー) → pg_class
- インデックス(主キー) → pg_index
- id列、name列 → pg_attribute
- 主キー制約 → pg_constraint
- 依存関係（テーブルを削除したら主キーも削除する） → pg_depend



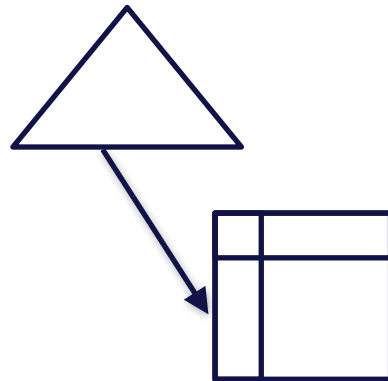
どのようにデータを取得するか？

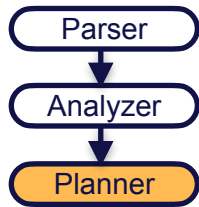
- ・ テーブルの状態、検索条件によって最適なアクセス方法は異なる
 - ・ テーブルにはたった10行しか入っていないかもしれない
 - ・ 1億行の内から一つの行を取得するかもしれない
- ・ プランナが最適（だと思われる）な実行プランを計画する
 - ・ 実行プラン：どのように目的のデータを取得するのかをまとめた計画



代表的なアクセス方法

- Seq Scan
 - ・ テーブルの（物理的な）先頭から順番にスキャンする
 - ・ テーブル内の多くの割合のデータを取得する時に高速
- Index Scan
 - ・ インデックスをスキャンしてから、テーブルをスキャンする
 - ・ テーブル内の一部のデータを取得する時に高速

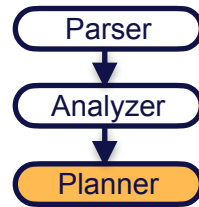




どうやってプランを決めるのか？

まずは選択度 (Selectivity)

- ・ テーブルの統計情報を利用して最適なアクセス方法を決める
 - ・ 列の固有値：(例) 一意の列の固有値は=1。都道府県を格納する列の固有値は=47
 - ・ 列の最頻値：最も多いデータとその割合のペアのリスト
 - ・ `most_common_vals` : {user0,user1,user2,user3,user4,user5,user6,user7,user8,user9}
 - ・ `most_common_freqs` : {0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1}
 - ・ 列のヒストグラム：度数を均一にしたときのBinの境界値
 - ・ `histogram_bounds` : {1,100,200,300,400,500,600,700,800,900,1000}
- ・ `pg_stats`ビュー
- ・ 統計情報はANALYZEで更新



行数の見積もり

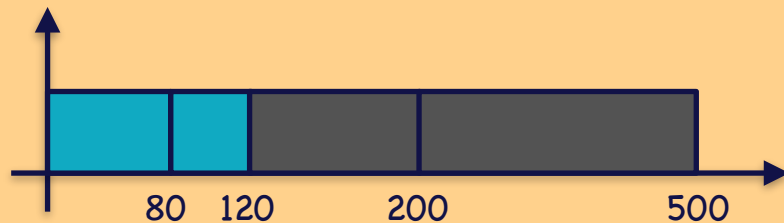
選択度 (Selectivity) の決定

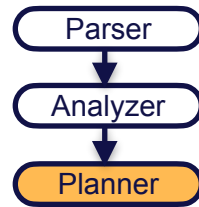
等価条件 (WHERE id = 100) の場合

- ・ 列が主キー (固有値が-1) なら簡単
 - ・ 必ず0行か1行
- ・ 最頻値から探す
 - ・ あればその割合が選択度
 - ・ なければ「それ以外の割合」から選択度を求める

範囲条件 (WHERE id < 100) の場合

- ・ ヒストグラムから100を含むBinの位置を求める
- ・ 「そのBin」 + 「それより前のBin」が占める割合を選択度とする

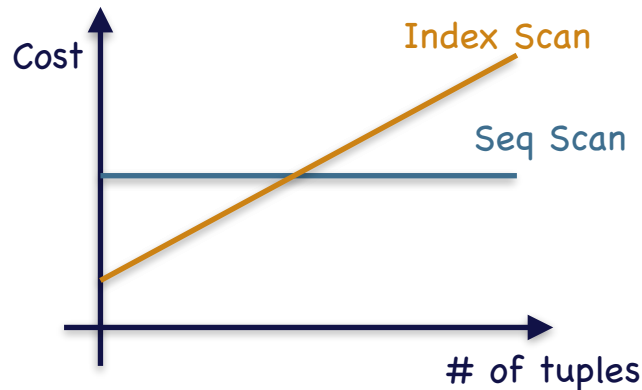


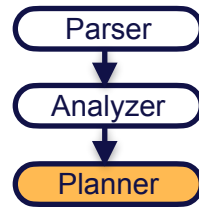


コストの見積もり

(推定) 取得行数から処理コストを計算する

- Seq Scanの場合
 - 取得行数によらず一定
 - 必要なデータ = テーブルサイズ
- Index Scanの場合
 - 取得行数によってコストが異なる
 - 必要なデータ = インデックス内を探す時に必要なデータ + テーブル内の検索対象が入っているデータ





EXPLAIN

実行プランの確認

```
=# EXPLAIN SELECT * FROM users WHERE id = 100;  
QUERY PLAN
```

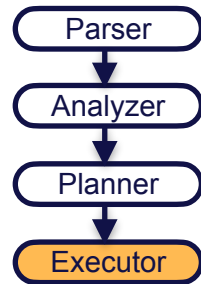
```
Index Scan using users_pkey on users (cost=0.28..8.29 rows=1 width=10)  
  Index Cond: (id = 100)  
(2 rows)
```

cost : プランナが見積もったコスト

rows : プランナが見積もった取得行数

width : プランナが見積もった行のサイズ

計画したプランを実行する



- ・ エグゼキュータが受け取ったプランを実行する
- ・ PostgreSQLのエグゼキュータはPull方式
 - ・ 再帰的にノードを実行していく
 - ・ 全ての実行ノードは1行返す
- ・ EXPLAIN ANALYZEでプランと一緒に実行時間を確認できる



プランはどのように実行されるのか？



QUERY PLAN

Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
 Index Cond: (id = 100)
(2 rows)



プランはどのように実行されるのか？



QUERY PLAN

Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
 Index Cond: (id = 100)
(2 rows)



プランはどのように実行されるのか？



QUERY PLAN

Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
Index Cond: (id = 100)
(2 rows)



プランはどのように実行されるのか？



どうぞ

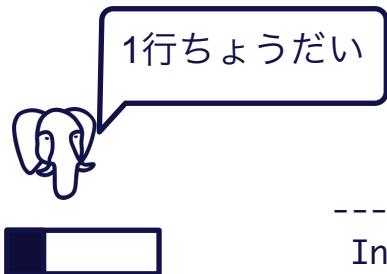
QUERY PLAN

最初の行を
取得するまでに
かかった時間(ms)

Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
Index Cond: (id = 100)
(2 rows)



プランはどのように実行されるのか？

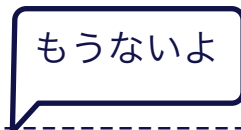
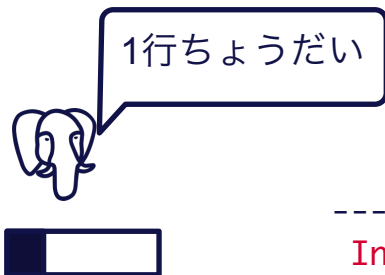


QUERY PLAN

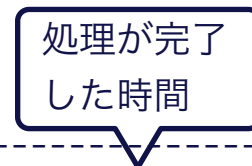
Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
Index Cond: (id = 100)
(2 rows)



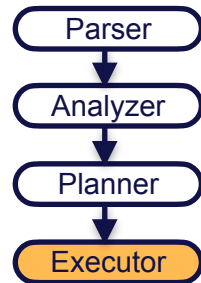
プランはどのように実行されるのか？



QUERY PLAN



Index Scan using users_pkey on users (actual time=0.020..0.021 rows=1 loops=1)
Index Cond: (id = 100)
(2 rows)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

ソートした結果を返したい

QUERY PLAN

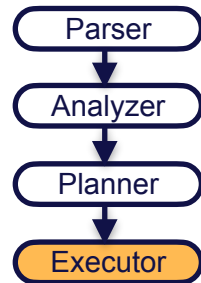
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

1行取得！

QUERY PLAN

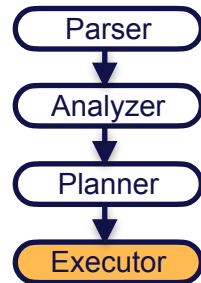
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

1行取得！！

QUERY PLAN

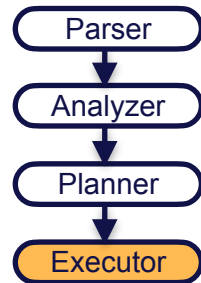
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

1行取得！！！！

QUERY PLAN

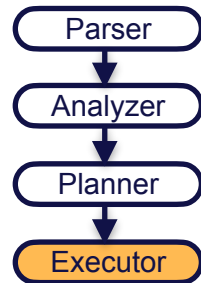
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

1行取得！！！！

QUERY PLAN

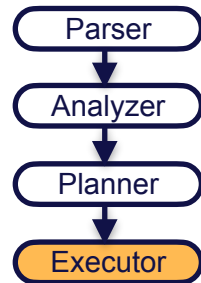
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

1行取得！！！！！！× 99

QUERY PLAN

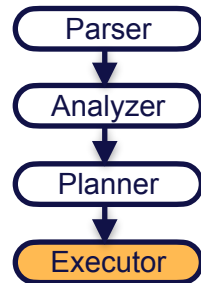
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

全部そろったからソートするよ

QUERY PLAN

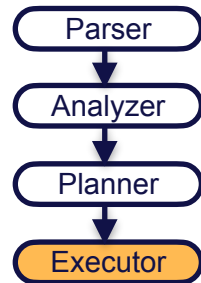
Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)



[少し脱線] ソートの場合はどうなる？

SELECT * FROM users WHERE id < 100 ORDER BY id;

やっと1行返せる

QUERY PLAN

Sort (actual time=1.646..1.656 rows=99 loops=1)

Sort Key: name

Sort Method: quicksort Memory: 30kB

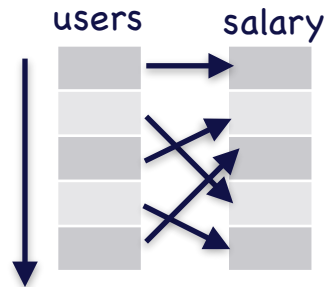
-> Index Scan using users_pkey on users (actual time=0.025..0.067 rows=99 loops=1)

Index Cond: (id < 100)

[少し脱線] 結合の場合はどうなる？

ユーザ(name = 'user_100')の給料を検索する

```
=# EXPLAIN SELECT * FROM users, salary WHERE  
users.id = salary.id and users.name = 'user_100';
```



QUERY PLAN

Nested Loop (cost=0.29..26.80 rows=1 width=18)

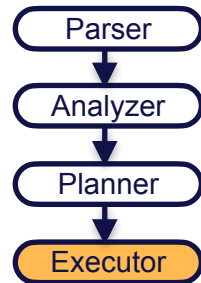
-> **Seq Scan** on users (cost=0.00..18.50 rows=1 width=10)

Filter: (name = 'user_100'::text)

-> **Index Scan** using salary_pkey on salary (cost=0.29..8.30 rows=1 width=8)

Index Cond: (id = users.id)

(5 rows)



[少し脱線] 結合の場合はどうなる？

ユーザ(name = 'user_100')の給料を検索する

まずは内部表（駆動表）
から1行とるよ

QUERY PLAN

Nested Loop (cost=0.29..26.80 rows=1 width=18)

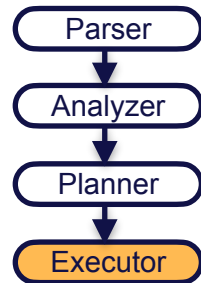
-> Seq Scan on users (cost=0.00..18.50 rows=1 width=10)

Filter: (name = 'user_100'::text)

-> Index Scan using salary_pkey on salary (cost=0.29..8.30 rows=1 width=8)

Index Cond: (id = users.id)

(5 rows)



[少し脱線] 結合の場合はどうなる？

ユーザ(name = 'user_100')の給料を検索する

まずは内部表（駆動表）
から1行とるよ

QUERY PLAN

Nested Loop (cost=0.29..26.80 rows=1 width=18)

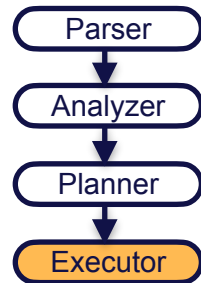
-> **Seq Scan on users** (cost=0.00..18.50 rows=1 width=10)

Filter: (name = 'user_100'::text)

-> **Index Scan using salary_pkey on salary** (cost=0.29..8.30 rows=1 width=8)

Index Cond: (id = users.id)

(5 rows)



[少し脱線] 結合の場合はどうなる？

ユーザ(name = 'user_100')の給料を検索する

先程取った行のID列の
値を使って次は内部表
から1行とるよ

QUERY PLAN

Nested Loop (cost=0.29..26.80 rows=1 width=18)

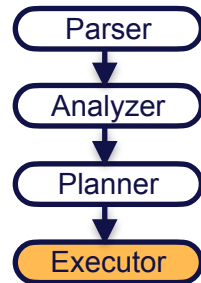
-> Seq Scan on users (cost=0.00..18.50 rows=1 width=10)

Filter: (name = 'user_100'::text)

-> **Index Scan using salary_pkey on salary** (cost=0.29..8.30 rows=1 width=8)

Index Cond: (id = users.id)

(5 rows)



[少し脱線] 結合の場合はどうなる？

ユーザ(name = 'user_100')の給料を検索する

1行結合できたので、
エグゼキュータに返すよ

QUERY PLAN

Nested Loop (cost=0.29..26.80 rows=1 width=18)

-> Seq Scan on users (cost=0.00..18.50 rows=1 width=10)

Filter: (name = 'user_100'::text)

-> Index Scan using salary_pkey on salary (cost=0.29..8.30 rows=1 width=8)

Index Cond: (id = users.id)

(5 rows)

(すこし脱線)

まっさらにテーブルに初めてINSERTした時にBackendプロセスが触るファイルは？

<code>open("base/5/2684\0", 0x2, 0x180)</code>	<code>= 13 0</code>	→	<code>pg_namespace_nspname_index</code>
<code>open("base/5/2615\0", 0x2, 0x180)</code>	<code>= 14 0</code>	→	<code>pg_namespace</code>
<code>open("base/5/2663\0", 0x2, 0x180)</code>	<code>= 15 0</code>	→	<code>pg_class_relname_nsp_index</code>
<code>open("base/5/2662\0", 0x2, 0x180)</code>	<code>= 16 0</code>	→	<code>pg_class_oid_index</code>
<code>open("base/5/2659\0", 0x2, 0x180)</code>	<code>= 17 0</code>	→	<code>pg_attribute_relid_attnum_index</code>
<code>open("base/5/2703\0", 0x2, 0x180)</code>	<code>= 18 0</code>	→	<code>pg_type_oid_index</code>
<code>open("base/5/1247\0", 0x2, 0x180)</code>	<code>= 19 0</code>	→	<code>pg_type</code>
<code>open("base/5/16384_fsm\0", 0x2, 0x180)</code>	<code>= -1 2</code>	→	<code>test_tbl</code> のFSM
<code>open("base/5/16384\0", 0x2, 0x180)</code>	<code>= 20 0</code>	→	<code>test_tbl</code>
<code>open("pg_wal/000000010000000000000001\0", 0x400002, 0x180)</code>	<code>= 21 0</code>	→	WAL

ここまでのまとめ



パーサがSQLを構文解析し、

アナライザがシステムカタログを使ってテーブルや列の有無を確認し、

プランナが統計情報を使って実行プランを計画し、

エグゼキュータが実行プランを実行する。

トランザクション

```
BEGIN;  
SELECT ...;  
INSERT ...;  
SELECT ...;  
COMMIT;
```

SELECTとUPDATEが同時に起きたら？

- ・ UPDATE途中のデータを見たくない
- ・ 1回目のSELECTと2回目でSELECTで同じ結果が見たい
- ・ ACIDのI (Isolation)
- ・ UPDATE中はSELECTできない、というのは性能的にNG

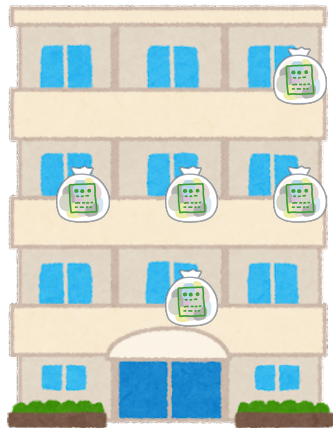
MVCC (Multi-Version Concurrency Control)

- ・ PostgreSQLは変更前、変更後の両方を持つ
- ・ 1つのデータに対し複数のバージョンを持てば、一方が変更している間にもう一方が読むことができる
- ・ PostgreSQLではUPDATEを「DELETE+INSERT」で実現
 - ・ UPDATEでもテーブルの物理的なサイズは大きくなっていく（テーブルの肥大化）
 - ・ 追記型アーキテクチャ
- ・ ROLLBACKしたデータ（誰からも参照されないデータ）もテーブルに残る
- ・ うまくフィルタリングしながらデータを読む

(すこし脱線) UNDOログとの違い

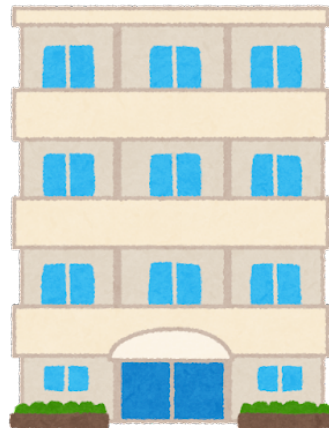
- ・ ゴミはページ内に放置
- ・ Vacuumは各部屋を回ってゴミを回収（ゴミがある部屋が事前にわかる＝Visibility Map）
- ・ ゴミじゃなかった（ROLLBACK）場合もそのままOK

PostgreSQL



- ・ ゴミはゴミ捨て場（UNDOログ）へ自分で持っていく
- ・ ゴミ捨て場にあるゴミを捨てるだけ
- ・ ゴミじゃなかった（ROLLBACK）場合は部屋に戻る必要がある

Database X



UNDO

トランザクションID (XID)

各トランザクションに割り当てられる非負整数値

- ・ トランザクションIDは単純増加する整数
- ・ トランザクション内で初めてデータを変更する際に、トランザクションIDが割り当てられる
- ・ 各タプルは2つのトランザクションIDを持つ
 - ・ xmin : そのタプルを追加したトランザクションのXID (INSERT、UPDATE)
 - ・ xmax : そのタプルを削除したトランザクションのXID (DELETE、UPDATE)

タプルの可視性

ゴミも混ざっているテーブル内のタプルをどのように正しく読むか？

- ・ 基本的な考え方
 - ・ 自分のXIDより小さいXIDで追加されたデータは読める（過去に追加された）
 - ・ 自分のXIDより小さいXIDで削除されたデータは読めない（過去に削除された）

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';

xmin (insert)	xmax (delete)	
100		Apple

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

```
XID = 100: INSERT 'Apple';  
          INSERT 'Ball';  
          COMMIT;
```

xmin (insert)	xmax (delete)	
100		Apple
100		Ball

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

```
XID = 100: INSERT 'Apple';  
          INSERT 'Ball';  
          COMMIT;  
XID = 200: UPDATE 'Apple' -> 'Cat';
```

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball
200		Cat

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

```
XID = 100: INSERT 'Apple';  
          INSERT 'Ball';  
          COMMIT;  
XID = 200: UPDATE 'Apple' -> 'Cat';
```

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball
200		Cat

XID = 150



データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple ✓
100		Ball
200		Cat

XID = 150



xmin = 100 → 過去にINSERTされた
 xmax = 200 → 未来でDELETEされる
 ⇒見える！！

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple ✓
100		Ball ✓
200		Cat

XID = 150



xmin = 100 → 過去にINSERTされた
 まだ誰からも削除されていない
 ⇒見える！！

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple ✓
100		Ball ✓
200		Cat

XID = 150



xmin = 200 → 未来にINSERTされる
 まだ誰からも削除されていない
 ⇒見えない！！

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

```
XID = 100: INSERT 'Apple';
          INSERT 'Ball';
          COMMIT;
XID = 200: UPDATE 'Apple' -> 'Cat';
```

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball
200		Cat

XID = 250



データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball
200		Cat

XID = 250



xmin = 100 → 過去にINSERTされた
 xmax = 200 → 過去にDELETEされた
 ⇒見えない

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball 
200		Cat

XID = 250



xmin = 100 → 過去にINSERTされた
 まだ誰からも削除されていない
 ⇒見える！！

データベースのスナップショットを見る

データベースある時点を切り取ったようなデータ

XID = 100: INSERT 'Apple';
 INSERT 'Ball';
 COMMIT;
 XID = 200: UPDATE 'Apple' -> 'Cat';

xmin (insert)	xmax (delete)	
100	200	Apple
100		Ball ✓
200		Cat ✓

XID = 250



xmin = 200 → 過去にINSERTされた
 まだ誰からも削除されていない
 ⇒見える！！

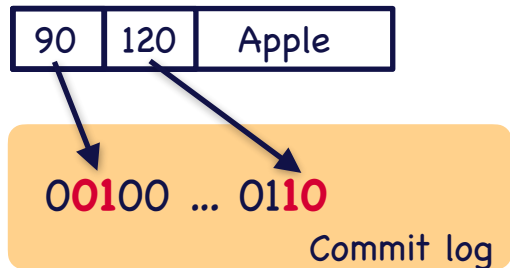
実はXIDだけの判断だけでは不十分

- ・ トランザクションがコミットされているとは限らない
 - ・ ロールバックされているかもしれないし、実行中かもしれない
- ・ 「過去のXIDによってINSERTされたタプル」 だったとしても、そのトランザクションがロールバックされている場合は、そのタプルは見えない

トランザクションのステータス

COMMIT、ABORTはどうやって判断する？

- ・ ACIDのA (Atomicity)
- ・ コミットログと呼ばれるファイルに、各トランザクションがCommit/Abortした情報を格納する (2bits/txn)
 - ・ たくさんテーブルを更新してもたった2bitの更新でCommit/Abortを記録する
 - ・ `${PGDATA}/pg_xact`ディレクトリ配下に保存
- ・ タプルから取得したXIDからステータスを確認して、「COMMITされたINSERT」なのか「ABORTされたINSERT」なのかを確認する



他にも考慮すべき点は多くある

- ・ SAVEPOINT
- ・ 自トランザクションが行った変更は見える必要がある
- ・ しかし、カーソルを使った場合はその時点のデータが見える必要がある
- ・ 各トランザクション分離レベルはどう実装しているのか？

まとめ

- ・ PostgreSQLがどのようにSQLを処理して、データを取得しているのかについて解説しました
- ・ ご要望等ありましたらぜひアンケートにてお知らせください
 - ・ ここを解説してほしい、ここを改善してほしい等

Thank you

Masahiko Sawada

masahiko.sawada@enterprisedb.com