内部を知って業務に活かす

PostgreSQL研究所

石井 達夫 ISHII Tatsuo
ishii@postgresql.org

最終回 エグゼキュート処理
～テーブルへのアクセスと結果の返却

はじめに

PostgreSQLの内部構造を解説するこの連載も，いよいよ大詰めのエグゼキュータの解説となりました．

前回までで，PostgreSQLが問い合わせを実行するための実行プランがどのようにして作られるかがおわかりいただけたかと思います．エグゼキュータの仕事は実行プランを忠実に実行していくだけです．このため，エグゼキュータは本質的にはシンプルな処理なのですが，ストレージマネージャ，キャッシュマネージャをはじめ，問い合わせを実行するためのさまざまな仕掛けが用意されており，それらを理解する必要があります．誌面の関係ですべては解説できませんので，特に重要と思われるものに的を絞って解説していきたいと思います．

今回は，解説のターゲットを現時点で最新安定版のPostgreSQL 8.1.2とします．

エグゼキュータのメインルーチン

エグゼキュータのメインルーチンはExecutorRunで，主な引数はQueryDescです．QueryDescの構造をリスト1に示します（コメントは筆者による）．この中で，今まで解説していなかったスナップショットについて説明します．

スナップショットとは

スナップショットは，**リスト2**のような構造体で示されるメモリ上のオブジェクトです．スナップショットは，データベース上のオブジェクトの可視性を判断するための重要なデータです（コメントは筆者による）．このデータ構造を理解するためには，トランザクションID（XID）とコマンドID（CID）の使い方に関する理解が必要です．

実行中のすべてのトランザクションにはXIDが割り当てられています．XIDは単調に増加し，したがって古いトランザクションは新しいトランザクションよりも大きなXIDを持っています[注1]．

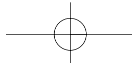
SnapShotDataのxminは実行中（すなわちまだコミットもロールバックもしていない）XIDのうち最も小さい（古い）もの，xmaxはSnapShotを取得した時点で次に割り当てられる予定の次のXIDを示します．詳細はstorage/ipc/procarray.cに定義されているGetSnapShotDataを参照してください．

コマンドIDは同じトランザクション内で管理される番号で，SQLコマンドが実行されると必要に応じてカウントアップされていきます．コマンドIDの使い方については後述します．

DestReceiver

検索処理のプランを実行すると，結果の行が出力されます．通常は接続されたフロントエンドに結果が送られますが，スタンドアローンモードで動くpostgresでは端末に結果が出力されます．また，フロントエンドと言っても，古いバージョンのPostgreSQLでは今とは少し違ったネットワークプロトコルで動くので，同じ方法では古いフロントエンドをサポートできません．

注1）実際にはXIDは32ビットの整数で，いつかはすべての値を使い尽くしてしまいますが，PostgreSQLではXIDは循環する数字として扱っており，XIDを比較することによってどちらが新しいトランザクションかを識別できるようになっています．



▼リスト1　QueryDesc構造体

```
typedef struct QueryDesc
{
    /* 以下のフィールドはCreateQueryDescが生成 */
    CmdType       operation;             /* CMD_SELECT, CMD_UPDATEなど */
    Query         *parsetree;            /* リライト処理後のパースツリー */
    Plan          *plantree;             /* プランツリー */
    Snapshot      snapshot;              /* この問い合わせのスナップショット */
    Snapshot      crosscheck_snapshot;   /* UPDATE/DELETEの参照整合性チェックの用のスナップショット */
    DestReceiver  *dest;                 /* 行の出力先 */
    ParamListInfo params;                /* パラメータ値 */
    bool          doInstrument;          /* TRUEならば実行時間測定を行う */

    /* これらのフィールドはExecutorStartがセットする */
    TupleDesc     tupDesc;               /* 結果タプルディスクリプタ */
    EState        *estate;               /* エグゼキュータの状態情報 */
    PlanState     *planstate;            /* プランノードごとの情報ツリー */
} QueryDesc;
```

▼リスト2　SnapshotData構造体

```
typedef struct SnapshotData
{
    TransactionId xmin;  /* XID < xminならば自分にとって可視 */
    TransactionId xmax;  /* XID >= xmaxならば不可視 */
    uint32        xcnt;  /* xip[]内のトランザクションの数 */
    TransactionId *xip;  /* 実行中トランザクションIDの配列 */
    /* 注: xip中のトランザクションIDはxmin <= xip[i] < xmaxを満たす */
    CommandId   curcid;  /* トランザクション中ではCID < curcidならば可視 */
} SnapshotData;
```

▼リスト3　DestReceiver構造体

```
struct _DestReceiver
{
    /* Called for each tuple to be output: */
    void (*receiveSlot) (TupleTableSlot *slot,DestReceiver *self);
    /* Per-executor-run initialization and shutdown: */
    void (*rStartup) (DestReceiver *self,int operation, TupleDesc typeinfo);
    void (*rShutdown) (DestReceiver *self);
    /* Destroy the receiver object itself (if dynamically allocated) */
    void (*rDestroy) (DestReceiver *self);
    /* CommandDest code for this receiver */
    CommandDest mydest;
    /* Private fields might appear beyond this point... */
};
```

このような違いを吸収するために，PostgreSQLでは結果の入出力処理などを必要に応じてエグゼキュータの実行前にセットします．そのための構造体をDestReceiverと言います（リスト3）．通常，rStartupには行のメタ情報（列名，データ型など）を出力する関数であるprinttup_startup，receiveSlotには実際に行を出力するprinttupを割り当てます．

単純な集約関数の実行例

今回は，単純な順スキャン＋集約関数実行のケースを取り上げます．実行するのは以下のようなSQLです．

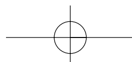
```
SELECT count(*) FROM t1;
```

ここで，t1は以下のようなテーブルです．

```
CREATE TABLE t1(i INTEGER);
```

このSQLはEXPLAINで見ると，図1のような実行プランを生成します．

さらに詳細な実行プランを見るには，debug_print_planを有効にして問い合わせを実行します．その際，debug_pretty_printも有効にしておくと結果が見やすくなります．また，client_min_messagesを調整し，



▼図1 実行プラン例

```
test=# EXPLAIN SELECT count(*) FROM t1;
                         QUERY PLAN
-----------------------------------------------------------
 Aggregate  (cost=36.75..36.76 rows=1 width=0)
   ->  Seq Scan on t1  (cost=0.00..31.40 rows=2140 width=0)
(2 rows)
```

結果が端末に出力されるようにします（図2）．

このままではちょっとわかりにくいので，このプランツリーを図にしたものが図3です．

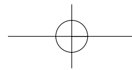
「メインプラン」はtypeに「AGG」と書いてあるもので，問い合わせの中の「count(*)」に対応します．count(*)の対象になっているのはテーブルt1で，メインプランのlefttreeからリンクしている下位のプランで処理されます．このプランのtypeに「SEQSCAN」とあることからわかるように，このプランはテーブルt1を順スキャンします．上位のプランであるメインプランは下位のプランの結果，すなわちt1テーブルを検索した結果を受け取ってcount(*)，すなわち行数を数える処理を実行し，最終的な結果をフロントエンドに返します．

ここでは，データベースを検索する下位のプランで，実際にどのような方法でデータベースにアクセスするのかを見ていきます．

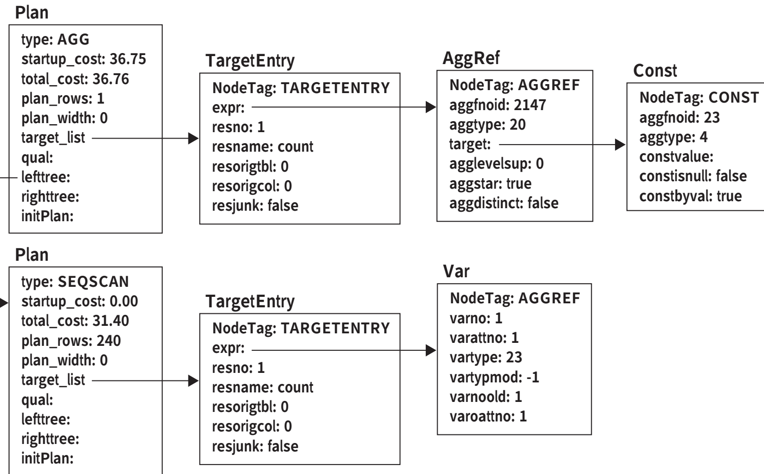
▼図2 詳細なプランの表示

```
test=# set debug_print_plan to on;
SET
test=# set debug_pretty_print to on;
SET
test=# set client_min_messages to 'debug5';
DEBUG:  CommitTransactionCommand
DEBUG:  CommitTransaction
DEBUG:      name:   unnamed;   blockState:
STARTED;  state:  INPROGR,  xid/subid/cid:
32086/1/0, nestlvl: 1, children: <>
SET
test=# SELECT count(*) FROM t1;
DEBUG:  StartTransactionCommand
DEBUG:  StartTransaction
DEBUG:      name:   unnamed;   blockState:
DEFAULT;  state:  INPROGR,  xid/subid/cid:
32087/1/0, nestlvl: 1, children: <>
DEBUG:  plan:
DETAIL:     {AGG
   :startup_cost 36.75
   :total_cost 36.76
   :plan_rows 1
   :plan_width 0
   :targetlist (
      {TARGETENTRY
      :expr
         {AGGREF
         :aggfnoid 2147
         :aggtype 20
         :target
            {CONST
            :consttype 23
            :constlen 4
            :constbyval true
            :constisnull false
            :constvalue 4 [ 1 0 0 0 ]
            }
         :agglevelsup 0
         :aggstar true
         :aggdistinct false
         }
      :resno 1
      :resname count
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
   )

        【中略】

      :qual <>
      :lefttree <>
      :righttree <>
      :initPlan <>
      :extParam (b)
      :allParam (b)
      :nParamExec 0
      :scanrelid 1
      }
   :righttree <>
   :initPlan <>
   :extParam (b)
   :allParam (b)
   :nParamExec 0
   :aggstrategy 0
   :numCols 0
   :numGroups 0
   }

DEBUG:  CommitTransactionCommand
DEBUG:  CommitTransaction
DEBUG:      name:   unnamed;   blockState:
STARTED;  state:  INPROGR,  xid/subid/cid:
32087/1/0, nestlvl: 1, children: <>
 count
-------
     0
(1 row)
```

PostgreSQL研最終回　06.2.27　10:45 AM　ページ217



▼図3　SELECT count(*) FROM t1;のプランツリー



ExecSeqScan

エグゼキュータのメインルーチンであるExecutorRunは，すべてのタプルを処理し終わるまでExecutePlanを呼び出します．ExecutePlanは，プランノードを調べ，対応するエグゼキュータのモジュールを呼び出します．先ほど説明した下位のプランは順スキャンを実行するもので，ExecSeqScanで処理されます．

ExecSeqScanの呼び出し形式は次のようになります．

```
TupleTableSlot *ExecSeqScan(SeqScanState *node)
```

TupleTableSlotは，ExecSeqScanが取得した1行を格納する構造体です．一方，この関数の引数SeqScanStateは，順スキャンを行うのに必要な情報を格納しています（**リスト4**）．

PlanStateはエグゼキュータが実行中の状態情報を格納します．実行プランの各ノードに対応してPlanStateが存在します．Relationは処理対象のテーブル，すなわちここではt1を「開いた」際の情報で，HeapScanDescはそのテーブルをスキャンするための情報を格納しています．

テーブルへのアクセス

PostgreSQLでは，テーブルやインデックスの実体は1つまたは複数のファイルです．テーブルの大きさが1Gバイトを超えると，テーブルは1Gバイトのファイルに分割されます．ファイル名はすべて数字です．したがってどこかにこの数字と実際のファイル名の対応表があることになり，PostgreSQLでは「pg_class」というシステムカタログがその役割を果たしています．

pg_classはすべてのテーブルを管理する特別なテーブルで，データベースクラスタの初期化時に作成されます[注2]．

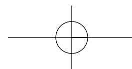
SysCache

pg_classへのアクセスは非常に頻繁に行われるため，SysCache（System Cache）という特別なサブシステムを使って特に効率よくアクセスできる工夫がされて

▼リスト4　ScanState構造体

```
typedef struct ScanState
{
    PlanState     ps;
    Relation      ss_currentRelation;
    HeapScanDesc ss_currentScanDesc;
    TupleTableSlot *ss_ScanTupleSlot;
} ScanState;
```

注2）テーブルを管理するテーブルなのに「class」という名前が付いているのは，PostgreSQLの前身であるpostgresがオブジェクト指向を旗印にしていたからだと思われます．postgresでは，テーブル＝クラス，行＝インスタンスというふうに概念化していました．



います．PostgreSQLのテーブルアクセス用のキャッシュは通常共有メモリ上に置かれますが，SysCacheはさらに高速にアクセスするために，バックエンドプロセスの中のメモリ（ヒープメモリ）に中に保持されます．つまり，各バックエンドプロセスはpg_classのプライベートコピーを持っていることになります．こうすると，pg_classを更新したときにキャッシュの内容が一致しないバックエンドが出る心配がありますが，そのあたりは共有メモリを使ってキャッシュ内容の変更通知を発行するようにしてうまく管理するようにしています[注3]．

pg_classにはテーブル名のほか，テーブルとファイルの実体を結びつけるための情報が含まれています．具体的にはrelfilenodeがファイル名を，reltablespaceが所属するテーブルスペースを管理するpg_tablespaceのOID（オブジェクトID）を表しています．reltablespaceが0はデフォルトテーブルスペース，すなわち$DATABASE/base/の直下を表しており，baseの下にはデータベースに対応するディレクトリが，さらにその下にテーブルの本体が格納されています．

RelCache

PostgreSQLはpg_classのrelfilenodeなどの情報を使ってテーブルをオープンするわけですが，こうしたルックアップのオーバヘッドを減らすために，RelCacheというサブシステムが利用されます．RelCacheはRelation Cacheの略ですが，キャッシュするのはあくまでテーブルをアクセスするための情報であり，テーブルの行などのキャッシュは後述するバッファキャッシュというサブシステムで行います．RelCacheの管理する情報については，src/include/utils/rel.hで定義されているRelationData構造体をご覧ください．

アクセスメソッド

テーブル本体へのアクセスは，heapと呼ばれるサブシステムによって行われます．heapは「アクセスメソッド」の一種として抽象化されています．他のアクセスメソッドには，インデックスをアクセスするindex，btree indexをアクセスするためのnbtreeなどがあり，src/backend/access/アクセスメソッド名としてソースコードが格納されています．

heapには，テーブルへの操作を行うための関数のセットが用意されています．主なものを表1に示します．

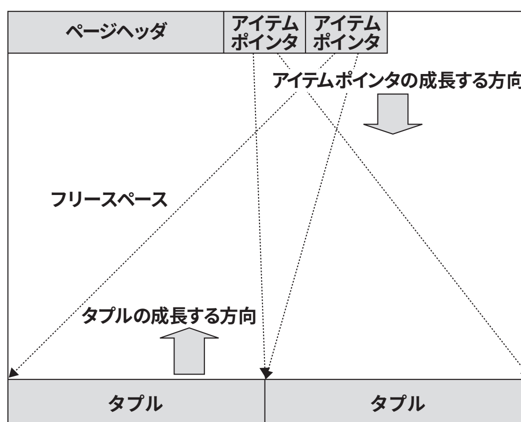
テーブルから行を読み取るまで

行の取得はheap_getnextもしくはheap_fetchで行います．ここでは，PostgreSQLがどのようにしてテーブルファイルから行を読み取るのか見てみましょう．

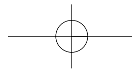
表1◎heapのアクセスメソッド

●関数名	●機能
heap_open	テーブルを開く
heap_close	テーブルを閉じる
heap_beginscan	テーブルのスキャンを開始
heap_endscan	テーブルのスキャンを終了
heap_getnext	行を取得
heap_fetch	タプルIDで行を取得
heap_insert	行を挿入
heap_delete	行を削除
heap_update	行を更新

▼図4　ページの構造



注3）詳細に興味のある方は，src/backend/utils/cache/inval.cをご覧ください．ただし冒頭「This is subtle stuff, so pay attention」とあるように結構ややこしいコードになっています．



ブロック／ページ

PostgreSQLではすべてのテーブルやインデックスは8192バイト単位の固定の「ブロック」としてアクセスされます．ブロックの中身のフォーマットは「ページ」として管理されています注4．

図4にページのフォーマットを示します．ページの中には行が0個以上入ります．PostgreSQLのソースコード上では，行ではなくて「タプル」という用語がよく使われているようなので，以後タプルという用語を使います．

ページの先頭に「ページヘッダ」という管理領域がありトランザクションログの管理，ページの中の空き領域を管理しています．

ページヘッダの後には「アイテムデータ」が続きます．アイテムデータは可変長の領域で，ページ内のタプルの位置を管理しています．タプルはページの後方から追加されていきます．タプルが1個追加されるたびにアイテムデータの1項目がページヘッダの後に追加されます．つまり，ページの中央の空き領域を，前からはアイテムデータ，後ろからはタプルが攻めていくようなイメージでページの空き領域が埋まっていくわけです．

タプルの構造

個々のタプルには管理用の「タプルヘッダ」があります．その構造を表2に示します．

タプルヘッダの後には「ヌルビットマップ」があり，NULLデータを含むタプルにおいて，ある列がNULLかどうかを記録します．その後には4バイトのOIDがありますが，テーブル作成時に「WITH OID」を指定しない限りは作成されません．

この後にタプルを構成する各列のデータが入ります．

ブロックへのアクセス

ブロックのアクセスは必ずバッファマネージャ（src/backend/storage/buffer/bufmgr.c）を通じて行います．バッファマネージャは共有メモリ上のバッファを管理しています．共有メモリ上のバッファは「shared buffer」と呼ばれ，postgresql.confの「shared_buffers」でその数を設定できるようになっています注5．

バッファマネージャの中で中心になる関数はReadBufferという関数で，指定されたテーブルの指定されたブロックをバッファに読み込んで返すという，一見単純な作業を行います．しかし，そのロジックは奥が深く，ここでは全部を語りきれませんが，考慮すべき問題をいくつかピックアップしておきましょう．

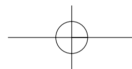
- 空いているバッファを効率的に探すにはどうしたらよいか．共有バッファはすべてのバックエンドプロセスがアクセスするため，競合をなるべく避けてア

表2◎タプルヘッダの構造

●フィールド名	●バイト長	●用途
t_xmin	4	行を挿入したトランザクションID
t_cmin	4	行を挿入したコマンドID
t_xmax	4	行を削除したトランザクションID
t_cmax/t_xvac	4	行を削除したコマンドIDまたはVACUUMによって移動された行のバージョン
t_ctid	6	この行あるいは新しい行のTID（タプルID）
t_natts	2	列の数
t_infomask	2	フラグビット
t_hoff	1	行データへのオフセット

注4）論理的には1ブロックの中に複数のページを格納することも可能ですが，現在の実装では1ブロック＝1ページとなっています．したがってブロックとページという用語の区別は曖昧ですが，ファイルからの物理的なアクセスを意識するときはブロック，ブロックの中身を意識する場合はページという用語を使い分けているようです．

注5）一時テーブルのアクセスは共有メモリバッファではなく「ローカルバッファ」，すなわちプロセス内のプライベートなメモリ上のバッファを使って行います．



クセスしなければなりません．これはかなり奥が深い問題で，PostgreSQL 8.1ではじめて根本的な改良が行われたくらいです．

- 全部のバッファが使用中だったらどうするか．なるべく「古い」バッファを探すのがよいのですが，凝ったことをやりすぎると逆にバッファの「新旧」を管理するオーバヘッドが増えてしまいます．PostgreSQLでは「clock sweep」という単純ですが効率のよい方法を使っています．

ReadBufferは空いているバッファを探し，もしなければ使用中でないバッファのうち今後もっとも使われそうにないバッファを再利用します．その際，以前そのバッファを経由して書き込みが行われている場合「dirty」フラグが立っています．つまり，バッファとディスクの内容が違っていることになるので，そのバッファをディスクに書き込んで同期を取ります．こうしてバッファの内容をディスクに反映することをできるだけ遅らせることができます．

ただし，いつまでも遅らせるわけにはいかないので，バックグラウンドライタ（bgwriter）と呼ばれるプロセスがときどき書き出しを行います．それでも残ったものはCHECKPOINTのタイミングで書き出されます．

使用中のバッファには「使用中」のフラグを立てます．このことを「ピンを立てる（pin）」と言い，逆に使用中フラグを落とすことを「ピンを抜く（unpin）」と言います．

ストレージマネージャ

バッファマネージャ（src/backend/storage/smgr/smgr.c）は直接ファイルI/Oは行いません．ファイルI/Oは「ストレージマネージャ」というサブシステムを通じて行います．ストレージマネージャはファイルI/Oを抽象化したもので，理論的にはハードディスク以外の外部記憶装置も扱うことができます．実際，PostgreSQLがpostgresであった時代には，光ディスクや，不揮発性メモリなどもストレージマネージャが管理していたこともありました．現在では「磁気ディスク」，すなわちハードディスクだけが外部記憶装置としてサポートされています．

磁気ディスクマネージャ

磁気ディスクマネージャ（the magnetic disk storage manager）はハードディスクへのI/Oを管理します．I/Oは「仮想ファイルディスクリプタ」を通じて行われます．そのほか，テーブル名やテーブルスペースから物理ファイル名へのマッピングもここで行います．

仮想ファイルディスクリプタ管理

仮想ファイルディスクリプタ管理サブシステム（src/backend/storage/file/fd.c）は直接ファイルI/Oは，物理ファイルディスクリプタのキャッシュ管理などを行います．このレイヤになってはじめて実際のファイルへのアクセスが行われます．

PostgreSQLでは，個々のテーブルがファイルにマップされているため，多数のファイルのオープン／クローズを頻繁に行います．本当にファイルのオープン／クローズを繰り返すとパフォーマンスが低下するため，ファイルディスクリプタをキャッシュしてできるだけファイルのオープン／クローズを減らすのがこのサブシステムの目的です．

なお，ここでキャッシュするファイルディスクリプタの数は，postgresql.confのmax_files_per_processで設定可能です．

タプルの正当性検証

以上，長い道のりでしたが，heap_getnext（heap_fetch）⇨バッファマネージャ⇨ストレージマネージャ⇨磁気ディスクマネージャ⇨仮想ファイルディスクリプタ管理 というルートでエグゼキュータはタプルを取得していることがわかりました．

しかし，これで話はまだ終わりません．PostgreSQLでは追記型のデータ管理を採用しており，削除されたタプルも物理的にページの中に残されています．したがって，タプルを取得したあと，それが本当に正当なものなのかどうかを検証する必要があります．「だったら削除フラグを設けてそれをチェックすればよいのではないか」という意見もあると思いますが，事態はそう簡単ではありません．

PostgreSQLではMVCC（MultiVersion Concurrency

Control）を採用しており，更新中のタプルを他のトランザクションから読み出すことができます．MVCCをサポートしていないDBMSでは，更新中の行をアクセスするとブロックされてしまいます．MVCCによって高度なトランザクションの並列実行が可能になっているわけです．

反面MVCCの世界では，タプルの正当性の確認が複雑になります．たとえばトランザクションAがWHERE句を発行し，テーブル内の全タプルを更新し，まだコミットしていないとします．あるタプルaが更新されて，更新後の新しいタプルa1が作成されます．この場合少なくとも以下のような判断が必要です．

① そのUPDATEコマンドを発行したトランザクションA自体からはタプルa1が見えてはいけない．もし見えてしまうとa1を更新してa2を作り，さらにそのa2を更新してa3を作り……という無限ループに陥ってしまう，いわゆる「ハロウィーン問題」が発生します[注6]．
② 一方，Aの中でUPDATEコマンドの後に実行されたSELECTコマンドからは，a1が見えなければならない．
③ 他のトランザクションBからは，Aがコミットされるまではaが見え，a1が見えない状態でなければならない．
④ Aがコミット前にROLLBACKやクラッシュしてしまったら，すべてのトランザクションでaが見え，a1は見えない状態になっていなければならない．

①②のためには「コマンドID」というものが使われます．コマンドIDはINSERTやUPDATEで行を追加した際にt_cminとして記録されています（表2参照）．PostgreSQLでは，トランザクションの中のあるコマンドから見て，t_cminが現在のコマンドIDよりも小さくなければ不可視の行として処理します．UPDATEコマンドが終了すればコマンドIDがカウントアップされるので，後続のSELECTからは該当行が見えるようになるわけです．

③④のためには，タプルヘッダ中のXIDに関係するフィールド（t_xmin，t_xmax）およびフラグ（t_infomask）が使用されます．フラグでは，その行を変更したトランザクションがコミットされているかどうかなどが管理されます．たとえば，フラグがコミット済み，xminが自分のトランザクションIDではなくかつxmaxが未設定なら，その行は他のトランザクションが挿入してコミットしたものですから，自分にとって見えてよいタプルであると判断できます[注7]．

上位プランの実行と結果の返却

こうして下位プランがt1テーブルを順スキャンしてタプルを取り出しました．上位プランは下位プランが結果を1行取り出すたびに処理を行います．この場合はcount(*)ですから，単に行数を数えるだけです．

下位プランがすべてのタプルを取り出し終えたら，上位プランは結果をDestreciverのreceiveSlotに割り当てた関数を通じてフロントエンドに転送します．

これでエグゼキュータの処理は終わりです．

まとめ

PostgreSQLの内部に迫る連載の最終回は，実際に問い合わせを実行するエグゼキュータを解説しました．テーブルから行を読み出すという単純な作業においても，背後では複雑な処理が行われていることをおわかりいただけたと思います．

PostgreSQLに限りませんが，ソフトウェアの内部処理とその実装を理解することで，ソフトウェアに対する理解の次元は確実に高まります．本連載がPostgreSQL理解のステップアップの道しるべとしてお役に立てば幸いです．

最後に，1年間この連載におつき合いいただいた読者の皆様に感謝の気持ちを表して連載の締めくくりとさせていただきます．

ではまたどこかでお会いしましょう．さようなら．

注6）聞くところによると，最初にこの種の問題が発見されたのが10月31日，すなわちハロウィーンの日だったところから，こう名付けられたのだそうです．
注7）詳細なロジックに関しては，HeapTupleSatisfiesNow（src/backend/utils/time/tqual.c）をご覧ください．