PostgreSQL 解析資料

～ MVCC ～

(株) NTT データ
オープンソース開発センタ
井久保 寛明

1. はじめに

本ドキュメントでは、PostgreSQL で使われている MVCC(Multi-Version Concurrency Control)の実装について説明する。MVCC とは、多版式同時実行制御と訳され、同時実行制御に使われる方式である。MVCC の概要については、２章で説明する。

1.1. 対象バージョン

本資料は、PostgreSQL7.4.3 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。

1.2. 用語について

この資料は、データベースで使われるトランザクションやロックなどの概念を知っているものとして書いている。それぞれの用語などは、適宜、書籍などを参考にしてほしい。ジム・グレイ、アンドレアス・ロイター著の「Transaction Processing : Concepts and Techniques」の訳本「トランザクション処理 概念と技法」[1]などが参考になる。

2. MVCC とは

トランザクションにおける同時実行性制御の最も一般的な方法として、ロックが挙げられる。トランザクションは、タプルを読む際に共有ロックを取得し、タプルの更新を行う際には排他ロックを取得する。ロックマネージャがロック要求を制御することで、同時実行をした際にも矛盾が起こらないようにする。同時実行の性能とトランザクションの分離レベルにはトレードオフの関係があり、ほとんどのDBMS製品は、トランザクションの分離レベルをユーザが選択できるようになっている。トランザクションの強い分離レベルである「直列化可能性」を実現するためには、取得した共有ロックと排他ロックの両方に対して２フェーズロックを強制する。学術的には直列化可能性を重視する傾向にあるが、直列化可能性を重視すると同時実行性能が著しく落ちるため、現在のほとんどのDBMSではもっと緩いトランザクションの分離レベルをデフォルト値としている。PostgreSQLを含め、多くの製品はREAD COMMITED（または、カーソル安定）と呼ばれるレベルになっている。これは、他のトランザクションがコミットしたタプルはすぐに見ることができるというものであり、厳密に直列化可能

[1] 「トランザクション処理 概念と技法」（上） ISBN: 4-8222-8102-7、（下）ISBN: 4-8222-8103-5

Copyright © 2004 NTTDATA

性を実現することはできないが、アプリケーションレベルでは矛盾が起きないようにプログラムの作成が可能なレベルである[2]。

READ COMMITED の場合、排他ロックは２フェーズロックになっているが、共有ロックは参照が終わるとすぐにロックを開放する。このような動きをするために、更新がかかったタプルを読み出そうとする場合、更新がかかったタプルがコミットされるまで読み出しは待たされることになる。この部分を改善するために、多版式という考え方が出てくる。更新前のバージョンのタプルと更新後のバージョンのタプルを別に用意する方法である。これには２つの方法がある。１つは、更新前のタプルを別の場所に保管しておき、更新前のタプルを更新後のタプルで書き換える方法である。更新前のタプルは、それを参照しているトランザクションがある間保管しておく。もちろん、更新前のタプルをコピーしておくのではなく、更新前のタプルが必要な際にトランザクションログを利用して更新前のタプルを別の場所に再構築してもよい。

もう１つは、追記型のストレージアーキテクチャを用いる方法である。この方法では、更新前のタプルは元も場所に置いたまま、更新後のタプルを新しいタプルとして追記する。こちらが PostgreSQL の採用している追記型と呼ばれる方式である。利点としては、ロールバックやリカバリの実装が非常に単純になる。欠点は、更新が発生するごとにデータ領域を消費していくため、定期的に不要領域を回収する処理が必要である。PostgreSQL では、この処理を VACUUM と呼んでいる。

PostgreSQLのMVCCの動きを簡単に説明する。具体例を示しながら説明するために、図 2-1 を使用する。左側にある、①～④の数字のついている横長の四角がタプルである。右側の○×の表は、各トランザクションIDで、それらのタプルが見えるかどうかを示している。各タプルには、そのタプルを生成したトランザクションIDとしてt_xminとタプルを削除したトランザクションIDとしてt_xmaxを持っている。実は、これは説明を簡単にするために多少単純化している。実際は、同一トランザクション内でも前後関係が分かるようにt_xmin、t_xmaxに対応するコマンドID[3]も保存している。

	t_xmin	t_xmax		トランザクションID 90	106	120
①	100			×	○	○
②	101	105		×	×	×
③	105	110		×	○	×
④	110			×	×	○

図 2-1 タプルの可視性

タプルが生成されたときに、そのタプルを生成したトランザクション ID が t_xmin に入る。タプルが生成された直後は、①または④のように t_xmin だけが入っていて、t_xmax は空である。仮に同じト

[2] 例えば、１つのトランザクション中で同じテーブルを何度も読み出すと、その間にコミットされたレコードが見えるようになったりする。
[3] 同一トランザクション内で実行された、個々のコマンド（SQLなど）に付けられたID。

Copyright © 2004 NTTDATA

ランザクション内で生成削除が行われると、t_xmin と t_xmax には同じ値が入る。そして、他のトランザクションからは全く見えない。タプルを削除すると、②や③のように t_xmax にそのタプルを削除したトランザクション ID が書き込まれる。更新の場合は、前のタプルを削除して、同時に新しい行を追加する。③と④のように、③の t_xmax と④の t_xmin が同じになるタプルの組を作る。この図では、②のタプルが③に更新されて、さらに④に更新されたと考えられる。

ここで、各トランザクションからどのように行が見えるかを考える。実行しているトランザクションのトランザクション ID より t_xmin が大きければ、そのレコードは見えないことになる。t_xmin の方が小さければ、そのトランザクションより先に作成された行なので、そのタプルは見える可能性がある。このタプルに t_xmax が存在しないか、t_xmax の値がタプルのトランザクション ID より大きい場合、このタプルはそのトランザクションから見えることになる。トランザクション ID が t_xmax より大きい場合、そのタプルは削除されたか、または、更新により新しい行が作成されているので、そのタプルは見えない。

図 2-1 では、トランザクション ID が 90 のトランザクションからは全てのタプルが見えない。トランザクション ID が 106 のトランザクションでは、t_xmin を見ると、①と②と③が見える候補に入る。①は t_xmax がないので見える。②に関しては、t_xmax がトランザクション ID より小さいので、106 のトランザクションが開始する前に削除されたことになっているので見えない。③については、トランザクション ID が t_xmax より小さいので、見ることができる。

同様に調べていくと、120 のトランザクションで見えるのは①と④になる。このように、あるタプルが ② --> ③ --> ④ と更新されたとしても、他のトランザクションからは、どれか１つしか見えない。

ここまでの説明では、話を簡単にするために各トランザクション ID はコミットされたものとして説明した。続いて、同時実行を踏まえた例を説明する。

105 と 106 のトランザクションが開始されたとする。図と違う状態だが、この時点で、②のタプルはまだ更新されていないとする。そうすると②のタプルは、106 のトランザクションから見ることができる。そこで、105 のトランザクションが②のタプルを更新して、図のような②と③のタプル状態になったとする。再度、106 のトランザクションが②のレコードを読みに行ったらどうなるだろうか？

t_xmax に 105 が書き込まれたので、③のタプルを読むように勘違いしてしまうかもしれないが、実は、書き込まれているトランザクションがコミットされているかどうかをチェックするため、②の t_xmax を無効とみなして、②のタプルが見えるのである。

さらに、この続きとして 105 がコミットし、106 がコミットしていないという状況で、トランザクションの分離レベルを考えてみる。READ COMMITEDの場合、②と③を読み直すと、105 がコミットしているため③だけ読める。SERIALIZABLEの場合、スナップショット[4]の中に 106 の開始時点で 105 のトランザクションが実行中であったという記録があるため、105 がコミットしても、②のt_xmaxを無効とみなして、②のタプルが見える。

もし MVCC でない DBMS で SERIALIZABLE を実装しようと思うと、テーブルロックで排他制御することになり、同時実行性は著しく低下する。

[4] 後述。

Copyright © 2004 NTTDATA

3. MVCC の実装の概要

この章では、２章で説明した MVCC が、どのような技術を使って PostgreSQL に実装されているかを見ていく。

3.1. タプルのデータ構造

PostgreSQL のタプルは、ディスク上とメモリ上で同じ構造をしている。DBMS の実装としては、ディスクとメモリでデータ構造を変えることも可能だが、PostgreSQL では同じ構造を使っている。

3.1.1. ヒープタプルの構造

ヒープタプルの構造の概要は次の図のようになっている。

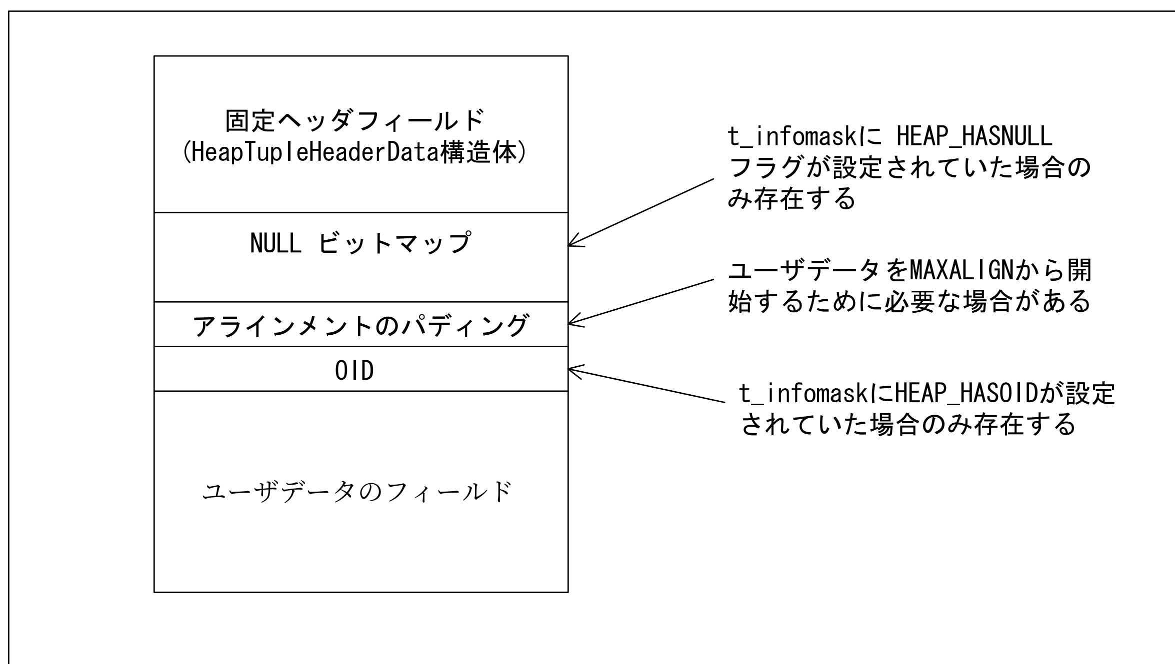

図 3-1 タプルのデータ構造

先頭にある固定長のヘッダフィールドとユーザデータのフィールド以外は、タプル中にない場合もある。これらは、ヘッダ中のビットフラグ t_infomask で制御される。無駄な情報を書かないことにより、ディスクおよびメモリの利用効率を高めている。NULL ビットマップは、タプル中に NULL のカラムが存在しない場合は、必要ないので作成されない。また、OID なしテーブルでは、図の OID のフィールドは存在しない。

3.1.2. HeapTupleHeaderData 構造体

タプルのヘッダである HeapTupleHeaderData 構造体は、ソースコード上では、次のように定義されている。

Copyright © 2004 NTTDATA

```
typedef struct HeapTupleHeaderData
{
    TransactionId t_xmin;          /* このタプルを insert した トランザクション ID */
    union
      {
         CommandId   t_cmin;     /* このタプルを insert した コマンド ID */
         TransactionId t_xmax;   /* このタプルを削除した トランザクション ID */
      } t_field2;
    union
      {
         CommandId   t_cmax;     /* このタプルを削除した コマンド ID */
         TransactionId t_xvac;   /* VACUUM FULL のトランザクション ID */
      } t_field3;
    ItemPointerData t_ctid;       /* current TID of this or newer tuple .
                                     BlockIdData (unit16 が２つ) と OffsetNumber (uint16)の
                                     合計 48bit で構成 */
    int16       t_natts;         /* number of attributes */
    uint16      t_infomask;      /* 様々なビットフラグ */
    uint8       t_hoff;          /* sizeof header incl. bitmap, padding */
    /* ^ - 23 bytes - ^ */
    bits8       t_bits[1];       /* bitmap of NULLs -- VARIABLE LENGTH */
    /* MORE DATA FOLLOWS AT END OF STRUCT */
} HeapTupleHeaderData;
```

まず、先頭にこのタプルを生成したトランザクションの ID t_xmin が入っている。次の union t_field2 で定義されているのは、タプルを生成したコマンド ID t_cmin か このタプルを削除したトランザクションの ID t_xmax である。このどちらが入っているかは、ビットフラグ t_infomask によって決定される。その次の union t_field3 で定義されているのが、このタプルを削除した コマンド ID t_cmax か このタプルを移動させた VACUUM FULL のトランザクション ID t_xvac である。これも、どちらが入っているかは、ビットフラグ t_infomask によって決定される。タプルには限られた状態しかないことから、両方の変数は同時に使わないことがわかっているため、この２つの union を使って物理領域を節約している。タプルの取り得る状態とこれらの変数の関係については後述する。
t_ctid は、このタプルの ID にあたる。何ブロック目の何タプル目かを示している。t_natts はタプルに含まれる属性（カラム）の数である。t_infomask は、タプルの付加情報のフラグである。詳しくは後述する。t_hoff は、ユーザデータの開始場所へのオフセットである。タプルの先頭から図 3-1 のユーザデータのフィールドまでのバイト数になる。

3.1.3. t_xmin、t_cmin、t_xmax、t_cmax、t_xvac

ディスク領域の節約のため、t_xmin, t_cmin, t_xmax, t_cmax, t_xvac の５つの情報は、３つの物理フィールドとして格納されている。t_xmin は、常に値が格納されている。しかし、t_cmin と t_xmax はフィールドを共有している。そして、t_cmax と t_xvac も共有している。これがうまくいくのは、タプルには限られた数の状態しかないということがわかっているからである。例えば、t_cmin と t_cmax は、それぞれ insert と delete を行っているトランザクションの生存期間しか使われることはない。
タプルには、次の表の状態しかない。

Copyright © 2004 NTTDATA

		t_xmin	t_cmin	t_xmax	t_cmax	t_xvac
1	NEW (削除されていなくて、VACUUM で動かされてもいない)	有効	有効	未設定	未設定	未設定
2	生成したトランザクションで削除された	有効	有効	= t_xmin	有効	未設定
3	他のトランザクションから削除された	有効	不要	有効	有効	未設定
4	VACUUM FULL によって移動された	有効	不要	有効	不要	有効

表 3-1 t_xmin, t_cmin, t_xmax, t_cmax, t_xvac の取り得る状態

ビットフラグ t_infomask によって t_xmax = t_xmin を表すことができるので、２の状態で、t_cmin と t_xmax を同時に保持しなくてよい。
VACUUM FULL は、t_cmin か t_cmax がまだ有効な状態 (例えば、insert 中または削除中)では、そのタプルを移動させようとしないというのが前提である。したがって、この表のようになるため、t_cmax と t_xvac を同時に保持しなくてもいい。

3.1.4. ビットフラグ t_infomask

タプルのデータ構造は、メモリおよびディスク領域を節約するために、可変構造になっており、不要なデータを持たないようにしてある。可変構造を制御するのが、HeapTupleHeaderData 構造体の中の t_infomask である。

t_infomask のビットフラグの情報は、include/access/htup.h に次のように定義されている。

```
#define HEAP_HASNULL           0x0001 : NULL の属性を持っている
#define HEAP_HASVARWIDTH       0x0002 : 可変長属性を持っている
#define HEAP_HASEXTERNAL       0x0004 : 外部に格納されている属性(TOAST)を持っている
#define HEAP_HASCOMPRESSED     0x0008 : 圧縮して格納してある属性を持っている
#define HEAP_HASOID            0x0010 : OID のフィールドを持っている
                               0x0020 : 現在未使用
#define HEAP_XMAX_IS_XMIN      0x0040 : 同一トランザクション中で、生成と削除が行われた
#define HEAP_XMAX_UNLOGGED     0x0080 : 更新のためにロックしている（ロギングはしていない)
#define HEAP_XMIN_COMMITTED    0x0100 : t_xmin がコミットされていた
#define HEAP_XMIN_INVALID      0x0200 : t_xmin が無効、またはアボートされた
#define HEAP_XMAX_COMMITTED    0x0400 : t_xmax がコミットされた
#define HEAP_XMAX_INVALID      0x0800 : t_xmax が無効、またはアボートされた
#define HEAP_MARKED_FOR_UPDATE 0x1000 : 更新のためにしるしをつける
#define HEAP_UPDATED           0x2000 : 更新済みの行
#define HEAP_MOVED_OFF         0x4000 : VACUUM FULL で別の場所に移動された
#define HEAP_MOVED_IN          0x8000 : VACUUM FULL で別の場所から移動されてきた
```

まず、先頭から５つの定義（HEAP_HASNULL から HEAP_HASOID）がタプルの構造に関係する情報ビットである。
例えば、HEAP_HASNULL にビットが設定されている場合は、NULL の属性を持っているので、ヒ

Copyright © 2004 NTTDATA

ープタプルの構造として、NULL ビットマップを持つようになる。そしてあるカラムが NULL 値の場合、この NULL ビットマップの該当位置にフラグが設定される。

次の HEAP_XMAX_IS_XMIN から最後の HEAP_MOVED_IN までは、前述のタプルの状態を保持して union で定義された変数の領域を制御するものや、処理の高速化や機能支援のためビットである。例えば、HEAP_XMAX_IS_XMIN は、表 3-1 のタプルの状態２を支援するもので、同一トランザクション中で生成と削除が行われた場合に使用される。これによって、t_xmin = t_xmax であることを表している。また、HEAP_XMIN_INVALID や HEAP_XMAX_INVALID なども、これらの変数が有効であるかどうかを示す。HEAP_XMIN_COMMITTED や HEAP_XMAX_COMMITTED は、そこに書かれたトランザクション ID がコミット済みかどうかを毎回チェックしなくていいようにする、処理高速化のためのビットである。

HEAP_XMIN_COMMITTED や HEAP_XMAX_COMMITTED は、検索処理の実行中でも変更される場合がある。特に、データ挿入や更新をおこなった後の次の検索処理では、ほぼ確実にビットが更新される。これは、タプルを挿入や更新したときは、t_xmin や t_xmax はまだ実行中のため、コミット済みではない。次の検索がかかったときに、ようやく t_xmin や t_xmax がコミットされていることが分かるので、t_infomask にコミット済みフラグを設定できるのである。
アクセス頻度の高いテーブルの場合、タプルの挿入や更新によるページの変更がディスクに反映される前に t_infomask が変更されるため、それほど余分なディスク I/O を引き起こすことはない。ただし、最悪の場合は、t_infomask の変更のためだけのディスク I/O を行うことになる。

3.2. コミットログとプロセステーブル

コミットログは、clog と呼ばれている。clog とは、トランザクション毎のステータスを管理するものであり、t_xmin や t_xmax などに書き込まれているトランザクション ID のトランザクションがコミットされているかどうかの判定に使われる。トランザクションのステータスには、実行中（IN_PROGRESS）、コミットされた（COMMITED）、アボートされた（ABORTED）の３つがある。これら３つのステータスを２ビット１エントリとして保持している。ソースコード中では、次のように定義されている。

```
#define TRANSACTION_STATUS_IN_PROGRESS      0x00
#define TRANSACTION_STATUS_COMMITTED        0x01
#define TRANSACTION_STATUS_ABORTED          0x02
```

あるトランザクションがどういう状態にあるかは、clog ディレクトリ以下にあるコミットログのそのトランザクションのトランザクションＩＤに該当する２ビットをチェックすることで確認できる。clog の構造については、別の資料でまとめることにするが、先頭からトランザクション ID の順に２ビットずつのデータが入っていると考えてもらえばよい。

トランザクションの状態を保持するものとして、プロセステーブルも同時に考えておく必要がある。プロセステーブルに、各バックエンドプロセスが実行中のトランザクション ID が保持されているので、実行中のトランザクション ID を調べたい場合は、プロセステーブルをスキャンする。ある瞬間に実行

Copyright © 2004 NTTDATA

中であったトランザクションの一覧を保持する機能が、次に説明するスナップショットである。

3.3. スナップショット

前述のように、タプルを生成したトランザクション ID（t_xmin）とタプルを削除したトランザクション ID（t_xmax）の組と、それらのトランザクション ID がコミットされているかどうかという情報に基づいて、タプルが見えるかどうかが決まる。PostgreSQL では、ある特定時点で、どのタプルが見えていたかを保存するためにスナップショットを利用する。スナップショットは、ある特定時点で実行されていたトランザクションのトランザクション ID を集めたものである。

3.4. タプルの有効性チェック

あるタプルが有効であるかどうかは、その有効性を論ずる時間を指定する必要がある。時間の指定は、主体となるトランザクションのトランザクションIDと、スナップショットを指定する必要がある。スナップショットのタイミングとしては、基本的に、そのトランザクションIDの開始時点のスナップショットか、そのクエリの開始時点のスナップショットが使われることになる。前者だと、トランザクションの分離レベルがSERIALIZABLEであり、後者だとREAD COMMITEDになる。また、内部的な事情[5]で、その瞬間のスナップショットが必要な場合に、スナップショットを取り直す場合もある。

前述したように、タプルには、タプルを生成したトランザクション ID とタプルを削除したトランザクション ID が格納されている。では、自分のトランザクション ID が決まれば、すぐにそのタプルの有効性が判るかというと、そうではない。タプルに書き込まれているトランザクション ID がコミットされたトランザクションの ID かどうかを判断する必要がある。このために、スナップショットとコミットログが使用される。

4. ソースコードで見る MVCC

タプルの有効性をチェックしているソースコードは、utils/time/tqual.c である。また、タプルの構造を知る上で include/access/htup.h などが参考になる。

4.1. スナップショットの種類

タプルの有効性を調べる場合の、スナップショットの指定の方法は、次の６つがある。ここでは、どこかの時点で取得したスナップショットを明示的に指定する場合と、先頭５つにあるような内部処理用のスナップショットを指定する場合があることを押さえてもらえばよい。違いは、「4.3 タプルの有効性チェックの実装」で説明する。

[5] 例えば、ユーティリティ実行時のカタログへのアクセスなどである。

Copyright © 2004 NTTDATA

SnapshotNow	今現在のスナップショットとしてタプルを評価する
SnapshotSelf	今のトランザクションとして、タプルを評価する
SnapshotAny	特にスナップショットを考えない
SnapshotToast	TOAST アクセス用のスナップショット
SnapshotDirty	現在コミットされていないものもスナップショットに含める
明示的なスナップショットの指定	どこかの時点で取得したスナップショットを引数で渡す

DML によるクエリ処理では、どこかの時点で取得したスナップショットを明示的に指定する。つまり、トランザクションの開始時点のスナップショットや、ある SQL の開始時点のスナップショットを引数として渡す。
ここに挙げた、先頭から５つの種類のスナップショット指定は、ユーティリティ系コマンド実行時のカタログ情報参照など、内部処理に使用される。例えば、SnapshotNow 時に実行される HeapTupleSatisfiesNow () 関数に printf 文や elog() を埋め込むなどすると、create table 実行時などに、SnapshotNow が大量に実行されることが確認できる。

先頭の４つのスナップショットは、次のように定義されている。

```
#define SnapshotNow                  ((Snapshot) 0x0)
#define SnapshotSelf                 ((Snapshot) 0x1)
#define SnapshotAny                  ((Snapshot) 0x2)
#define SnapshotToast                ((Snapshot) 0x3)
```

SnapshotDirty は、次のように、スナップショット SnapshotDirtyData を格納するアドレスを指定する。

```
static SnapshotData SnapshotDirtyData;
Snapshot            SnapshotDirty = &SnapshotDirtyData;
```

明示的なスナップショットも、ある時点のスナップショットを格納しているスナップショットデータのアドレスを指定する。

4.2. タプルの有効性チェック

タプルの有効性チェック関数の主な呼び出し元は、HeapTupleSatisfiesVisibility() マクロである。このマクロに、タプルと前述のスナップショットを渡すことで、そのタプルが有効かどうかの判定を行う。この中で呼ばれない VACUUM 時のタプルの有効性チェックを行うような関数は、直接それらの関数が呼び出される。

```
#define HeapTupleSatisfiesVisibility(tuple, snapshot) ¥
((snapshot) == SnapshotNow ? ¥
    HeapTupleSatisfiesNow((tuple)->t_data) ¥
: ¥
    ((snapshot) == SnapshotSelf ? ¥
```

Copyright © 2004 NTTDATA

```
            HeapTupleSatisfiesItself((tuple)->t_data) ¥
        : ¥
            ((snapshot) == SnapshotAny ? ¥
                true ¥
            : ¥
                ((snapshot) == SnapshotToast ? ¥
                    HeapTupleSatisfiesToast((tuple)->t_data) ¥
                : ¥
                    ((snapshot) == SnapshotDirty ? ¥
                        HeapTupleSatisfiesDirty((tuple)->t_data) ¥
                    : ¥
                        HeapTupleSatisfiesSnapshot((tuple)->t_data, snapshot) ¥
                    ) ¥
                ) ¥
            ) ¥
        ) ¥
    )
```

HeapTupleSatisfiesVisibility()マクロの実行内容は、指定されたスナップショット毎に、処理を分岐する。明示的にスナップショットを指定する場合が、一番内側の HeapTupleSatisfiesSnapshot()関数を呼ぶ場合である。また、SnapshotAny の場合は、必ず true が返り、そのタプルは常に有効ということになる。

HeapTupleSatisfiesVisibility()マクロは、HeapTupleSatisfies() マクロから呼ばれる。
HeapTupleSatisfies() マクロでは、HeapTupleSatisfiesVisibility() マクロを呼び出す前の t_infomask を保持しておき、HeapTupleSatisfiesVisibility() マクロを実行した結果、t_infomask が変更されていたら、そのタプルの入っているページのバッファにダーティフラグを設定する仕組みが入っている。したがって、タプルの有効性チェックの関数だけを見ていると、ビットフラグ t_infomask を変更しても、ダーティフラグを設定していないかのように見えるので注意が必要である。

そして、HeapTupleSatisfies() マクロの呼び出し元は、次の３つである。

- access/heap/heapam.c: heapgettup()
- access/heap/heapam.c: heap_fetch()
- access/heap/heapam.c: heap_get_latest_tid()

これらは、ヒープタプルからデータを取り出すところであり、ファイルから１レコード読み出して、それが有効なタプルかどうかチェックし、有効なタプルが見つかったら１つ見つかったとしてリターンする。

次のような場所では、HeapTupleSatisfiesVisibility()マクロを使わずに、タプルの有効性チェックの関数（次節で説明）を直接呼び出している。

- utils/misc/database.c:
- utils/adt/ri_triggers.c:
- access/heap/heapam.c:
- catalog/index.c:
- commands/vacuum.c:

Copyright © 2004 NTTDATA

- commands/vacuumlazy.c:
- commands/vacuumlazy.c:
- access/index/indexam.c:
- access/nbtree/nbtinsert.c:

VACUUM 時のタプルの有効性チェック関数 HeapTupleSatisfiesVacuum()は、VACUUM や index のソースコードから直接呼び出される。また、update 時のタプルの有効性チェック関数 HeapTupleSatisfiesUpdate()も access/heap/heapam.c から直接呼び出されている。

4.3. タプルの有効性チェックの実装

タプルの有効性をチェックするための関数は、次のようなものがある。指定されたスナップショットによって、タプルを有効とみなすかどうかの条件が異なってくる。

HeapTupleSatisfiesItself()	ヒープタプル「自体」が有効である場合、true を返す（後述)。
HeapTupleSatisfiesNow()	ヒープタプルが「今」有効なら true を返す（後述)。
HeapTupleSatisfiesToast()	TOAST の行として、ヒープタプルが有効の場合は、true を返す。HeapTupleSatisfiesItself()の簡略版。
HeapTupleSatisfiesUpdate()	HeapTupleSatisfiesNow() のロジックと同じである。しかし、より細かい結果コードを返す。update は、そのタプルが見えるかどうかだけでなく、どういう状態かという情報も必要としているからである。また、タプルの有効性チェックは、現行のコマンド ID (CurrentCommandId) ではなく、渡された CommandId に対してテストされる。
HeapTupleSatisfiesDirty()	オープンされているトランザクションの影響を含めて、タプルが有効なら true を返す。 付加情報として、グローバル変数 SnapshotDirty の中に、そのタプルに影響する同時実行されているトランザクションのトランザクション ID を入れる。また、もし更新が開始されていたら、そのタプルの t_ctid (前方へのリンク) も返される。
HeapTupleSatisfiesSnapshot()	引数で与えられたスナップショットに対して、ヒープタプルが有効なら true を返す。 スナップショットが取られたときの状態で、タプルの有効性をチェックするという点を除いて、HeapTupleSatisfiesNow() と同じである。
HeapTupleSatisfiesVacuum()	VACUUM のために、タプルのステータスを決める。この関数でチェックされるのは、調べるタプルが、どれか実行中のトランザクションから、一時的に可視状態になっているかどうかである。もしそうなら、まだ VACUUM で消すことはできない。 戻り値として、タプルの状態を返す。

表を見てもらうと分かるが、HeapTupleSatisfiesItself() と HeapTupleSatisfiesNow() が基本となっ

Copyright © 2004 NTTDATA

ている。そこで、この２つは比較的詳細に説明する。HeapTupleSatisfiesItself() が最も基本的な考え方になるのだが、実際には、HeapTupleSatisfiesNow() と HeapTupleSatisfiesSnapshot() が最も多く使われる。

HeapTupleSatisfiesItself() は、現行トランザクションでそのタプルが有効かどうかをチェックする。タプルが現行のトランザクションによって生成されている場合は、次のような条件を示す。

```
(Xmin == my-transaction &&              「タプルは現行トランザクションによって生成されている」かつ
    (Xmax is null                       ①「削除はされていない」
     [|| Xmax != my-transaction]))        [ ②または、別のトランザクションによって削除された ]
```

このように、Xmax が入っていない状態なら有効である。
この条件は、ソースコード中のコメントから引用しているのだが、[]で書かれている部分は、Xmin はまだコミットしていないので他のトランザクションから見えないため、このように省略してある。

タプルが他のトランザクションによって生成された場合は次のようになる。

```
(Xmin is committed &&                   「タプルはコミットされたトランザクションによって生成さ
                                        れている」かつ、
    (Xmax is null                       ①「行は削除されていなければ有効」
       ||                                   または、
      (Xmax != my-transaction &&        ②「別のトランザクションに削除されたが、
           Xmax is not committed )))      まだコミットされていない」
```

まだ Xmax が書かれていなかったら有効である。Xmax が書かれている場合、Xmax を書き込んだのが現行トランザクションならタプルは無効である。現行トランザクション以外が書き込んでいて、そのトランザクションがまだコミットされていない場合、タプルは有効である。Xmax のトランザクションがコミットされていたらタプルは無効となる。

HeapTupleSatisfiesNow() は、タプルが「今」有効かどうかをチェックする。HeapTupleSatisfiesItself()との違いは、「現行のコマンドによって行われた変更」が含まれないことである。こうすることによって、update の実行時に自分が出力したタプルを再更新するかもしれないという状況を解決する。

タプルが現行のトランザクションによって生成されている場合は、次のような条件を示す。

```
(Xmin == my-transaction &&              「現行トランザクションによって作成された」、かつ
    Cmin != my-command &&               「このコマンドではない」そして、
      (Xmax is null                     ①「この行は削除されていない」ならば有効
        ||                                  または、
       (Xmax == my-transaction &&       ②「このトランザクションによって削除されている」、かつ、
           Cmax != my-command)))          「削除したのは、このコマンドではない」場合に有効
```

次に、タプルが他のトランザクションによって生成された場合は、次の場合に有効となる。

Copyright © 2004 NTTDATA

```
(Xmin is committed &&                    この行は、コミットされたトランザク
                                         ションによって変更された。
    (Xmax is null                        ①「まだ削除されていない」ならば有効
       ||                                    または、
     (Xmax == my-transaction &&          ②「このコマンドが削除しようとしている」ならば有効
          Cmax == my-command)
         ||                                  または、
     (Xmax is not committed &&           ③「別のトランザクションによって削除された」、かつ、
          Xmax != my-transaction)))        「まだ、そのトランザクションはコミットされていない」
                                           ならば有効
```

HeapTupleSatisfiesToast()は、HeapTupleSatisfiesItself ()の簡略版である。TOAST を使う場合、必ず元になるタプルが別のテーブルに格納されている。したがって、元のタプルが有効であれば、TOAST は自動的に有効なので、TOAST 側の行でタプルの有効性をチェックすることはない。ただし、VACUUM は、元のテーブルと独立に行われる。したがって、VACUUM 実行によって移動されたかどうかの条件だけをチェックする。

HeapTupleSatisfiesUpdate()は、更新の時に使われるタプルの有効性チェックの関数である。ロジックは、HeapTupleSatisfiesNow() と全く同じであるが、２つの違いがある。１つは、更新の場合、有効かどうかだけでなく、より細かい状態を知る必要があるため、タプルの状態のコードを返す。もう１つは、HeapTupleSatisfiesNow() が現行のコマンド ID (CurrentCommandId) に対して動作するのに対して、HeapTupleSatisfiesUpdate() は引数で渡されたコマンド ID に対してチェックが行われる。

HeapTupleSatisfiesDirty() は、基本的に HeapTupleSatisfiesItself ()と同じである。ただし、そのタプルを作ったり削除したりしたトランザクションがまだ実行中であっても、その瞬間有効であれば、true を返す。また、ダーティな状態で有効と判定した場合、タプルを操作しているトランザクションＩＤをスナップショットの領域（SnapshotDirty）に保存する。

HeapTupleSatisfiesSnapshot() は、引数に渡されたスナップショットが取られた時点でタプルの有効性を判定するという点を除いて、HeapTupleSatisfiesNow() と同じである。つまり、スナップショットを取った時点で、開始されていないトランザクションは全く見えないと判断し、また、タプルの可視性をチェックした時点でコミットされていたとしても、スナップショットを取った時点でコミットされていなかったトランザクションは、コミットされていないものと判断する。

HeapTupleSatisfiesVacuum() は、VACUUM 実行時にタプルを移動させていいかどうかを判定するために使用される。どれか実行中のトランザクションから、タプルが一時的に可視状態になっているかどうかがチェックの基準であり、HeapTupleSatisfiesItself() や HeapTupleSatisfiesNow() とは少し異なる。

4.4. スナップショットのインタフェース

スナップショット制御のためのインタフェースとして次のようなものがある。

Copyright © 2004 NTTDATA

SetQuerySnapshot()	新しいクエリのために、クエリスナップショットを取得する。取得するスナップショットは、トランザクションの分離レベルによって異なる。
CopyQuerySnapshot()	現在のクエリスナップショットのコピーを作成する。コピーは、現在のメモリコンテキストで palloc される。
CopyCurrentSnapshot()	現在を最新としてスナップショットを再作成し、そのコピーを返す。コピーは、現在のメモリコンテキストで palloc される。
FreeXactSnapshot()	トランザクションの終了時に、クエリスナップショットを開放する。

これらのインタフェースは、クエリ開始時にスナップショットを設定するもの、そのクエリ開始時のスナップショットのコピーを取り出すもの、その瞬間のスナップショットを作成してそのコピーを取り出すもの、最初に設定したクエリスナップショットを開放するものからなる。クエリスナップショットは固定領域に作成しておき、そのコピーを作成して使いまわすような使い方になる。

4.5. スナップショットの実装

スナップショットのデータ構造は、次のようになっている。

```
typedef struct SnapshotData
{
    TransactionId xmin;         /* XID < xmin are visible to me */
    TransactionId xmax;         /* XID >= xmax are invisible to me */
    uint32        xcnt;         /* # of xact ids in xip[] */
    TransactionId *xip;         /* array of xact IDs in progress */
    /* note: all ids in xip[] satisfy xmin <= xip[i] < xmax */
    CommandId     curcid;       /* in my xact, CID < curcid are visible */
    ItemPointerData tid;        /* required for Dirty snapshot -:( */
} SnapshotData;
typedef SnapshotData *Snapshot;
```

xmin は、このスナップショットに含まれる最小のトランザクション ID である。また、xmax は、このスナップショットに含まれる最大のトランザクション ID である。これらは、トランザクション ID とスナップショットの比較処理を高速化するため使用される。あるトランザクション ID が xmin より小さいか、または xmax より大きければ、スナップショット内の配列を調べるまでもなく、そのトランザクション ID が含まれていないことが分かる。

上記ソースコードのコメントで、「visible（見える）」「invisible（見えない）」と言っているのは、XID として、あるタプルを生成したトランザクションの ID を例にとっている（つまりタプル内の t_xmin であって、t_xmax ではない）。まず、XID < xmin、つまり、タプルの t_xmin がスナップショットの xmin より小さい場合、t_xmin は確実に終了しているので、そのタプルは見えるのである。t_xmin が実行中であれば、xmin は t_xmin を含むので、これより小さいことはない。このコメントでは、XID がアボートしていることは考慮していないようである。次に、XID >= xmax ということは、スナップショット作成時には、まだ t_xmin が開始されていないため、そのタプルは見えないことになる。

Copyright © 2004 NTTDATA

xcnt は、このスナップショット内に保存しているトランザクション ID の数である。そして、xip が、このスナップショットが作成されたときに実行されていたトランザクションのトランザクション ID の配列である。
curcid は、このスナップショットを作成したトランザクション ID である。curcid は、比較するトランザクションがこのスナップショットを作成したトランザクションと同じトランザクション ID の場合に使用される。タプルの作成したコマンド ID（t_cmin）と比較した場合、t_cmin が curcid より小さいと、そのタプルは見えることになる。タプルを削除したコマンド ID（t_cmax）との比較の場合、t_cmax が curcid より大きければ、そのタプルはまだ見えることになる。

tid は、Dirty Snapshot の場合に更新がかかっていた場合（つまり dirty なタプルであった場合）、その更新前のタプルの t_ctid を、リンクとして保持するために使用される。

4.5.1. スナップショットの領域管理

スナップショットの領域は、次のように固定領域として確保されている。

```
static SnapshotData QuerySnapshotData;
static SnapshotData SerializableSnapshotData;
static SnapshotData CurrentSnapshotData;
static SnapshotData SnapshotDirtyData;
```

これに対して、ポインタを使って参照する。

```
Snapshot    QuerySnapshot = NULL;
Snapshot    SerializableSnapshot = NULL;
Snapshot    SnapshotDirty = &SnapshotDirtyData;
```

SerializableSnapshot は、そのトランザクションではじめて作成するスナップショットである。トランザクションアイソレーションレベルが SERIALIZABLE の場合、QuerySnapshot のポインタは常に SerializableSnapshotData を指している。READ_COMMITED の場合は、QuerySnapshotData の領域に、クエリごとにスナップショットを作成する。

CurrentSnapshotData は、その瞬間のスナップショットを作成する際に、一時的に利用される。スナップショットのコピーを作成する関数内で使われるので、取得したスナップショットのコピーを渡すため、すぐに再利用可能である。

4.5.2. GetSnapshotData()

スナップショットの作成は、GetSnapshotData() 関数によって行われる。これは、storage/ipc/sinval.c に定義されている。
スナップショットには、xip[]という可変長の配列を持っていて、スナップショット作成時に実行されているトランザクションのトランザクション ID を保持する。領域を最大限に節約するのであれば、この配列の領域は、毎回必要な数を確保するのがよい。しかし、PostgreSQL では、２つの理由から最大バックエンドプロセス数を使っている。１つは、必要数を確認するために、PROC 構造体からプロセス数を見るためにはロックが必要であり、メモリアロケーションは、このロックを取得する前に実行

Copyright © 2004 NTTDATA

した方が良いという判断をしているためである。もう１つは、１度、最大バックエンドプロセスの数をmalloc()で確保してしまえば、あとは開放しないで再利用することで、毎回実行するメモリアロケーションのオーバヘッドを節約することができるためである。従って、FreeXactSnapshot() によるスナップショットの開放の際は、すぐにこの領域が利用されることを見越して、xipに割り当てたメモリは開放しない。

GetSnapshotData()では、バックエンドプロセスの情報を保持しているPROC構造体を順番に調べ、トランザクションを実行中であるバックエンドのトランザクションIDをスナップショットのxip配列に保存する。そして、最大のトランザクションID、最小のトランザクションIDなども計算する。

4.5.3. アイソレーションレベルによる違い

スナップショットの作成は、原則として、SetQuerySnapShot()で作成される。トランザクションアイソレーションレベルが、READ_COMMITEDの場合、SetQuerySnapShot() が実行されるごとに、QuerySnapshot として、QuerySnapshotDataの領域に新しいスナップショットを作成する。SERIALIZABLEの場合は、QuerySnapshot はつねに、SerializableSnapshot と同じになる。

5. 動作確認例

ここまでで、MVCCの実装に関する説明を行った。この章では、実際にPostgreSQLを動かしながら、動作の確認例を示そう。

5.1. タプルの可視性の確認例

まずは、タプルの可視性を確認する例を示す。

まず準備として、図 5-1のように、テーブルの生成、２つの値の挿入を行って、SELECT文で内容を確認する。SELECT文では、oid、xmin、xmax、cmin、cmaxを確認している。PostgreSQLでは、oid、xmin、xmax、cmin、cmaxをselect文に埋め込むことで、タプルのOIDやxmin、xmaxなどの値を確認することができる。

```
(682) create table b ( id int );
(683) insert into b values ( 1 );
(684) insert into b values ( 2 );
(685) select oid,xmin,xmax,cmin,cmax,* from b;

  oid  | xmin | xmax | cmin | cmax | id
-------+------+------+------+------+----
 17170 |  683 |    0 |    0 |    0 |  1
 17171 |  684 |    0 |    0 |    0 |  2
```

図 5-1 テーブルの作成、データ挿入とテーブルの状態

xminを見ると、それぞれのinsert文を実行したトランザクションIDが判り、そこから前後のcreate文とselect文のoidも判別できる。各SQL文の前に、丸で囲んで書いてある数字が、それぞれのトラ

Copyright © 2004 NTTDATA

ンザクション ID である。今回は、begin 〜 commit で明示的にトランザクションを指定していないため、各 SQL 文が１つのトランザクションとして、実行されている。

それでは、準備ができたところで、同時実行を行いながら、動きを見ていく。
図 5-2 では、トランザクション T1 に続いて、すぐにトランザクション T2 を起動する。psql を２つ起動すれば、このようなテストができる。

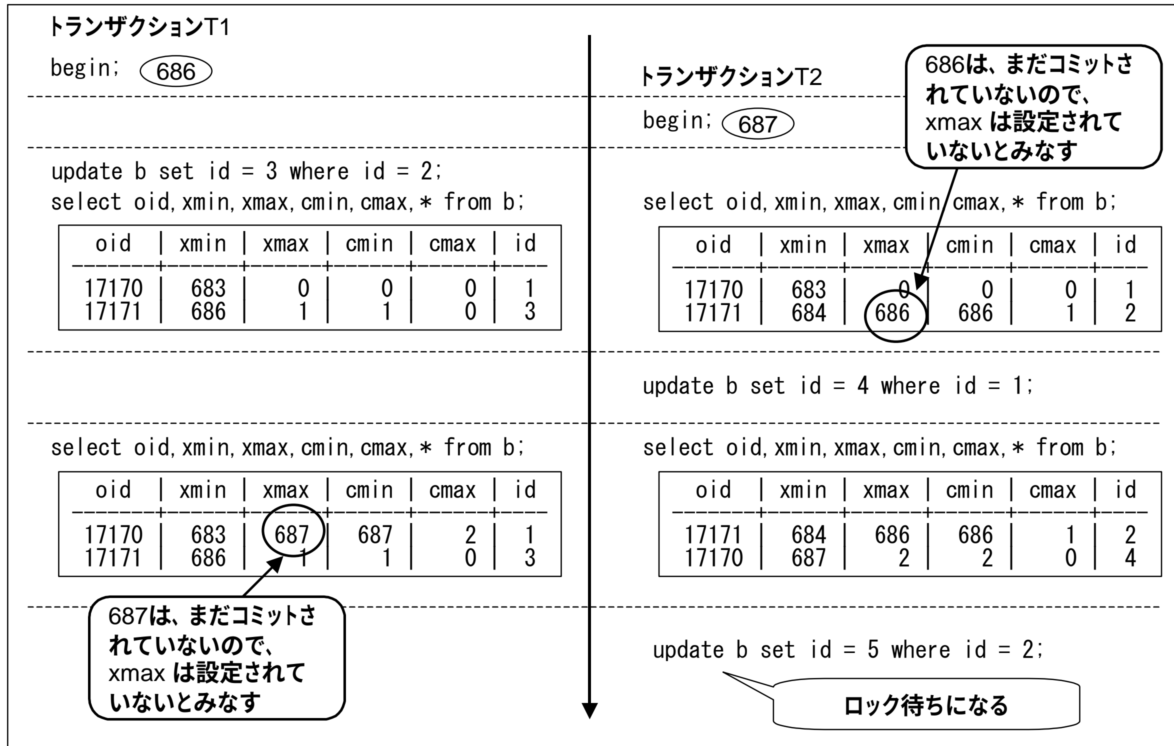

図 5-2 同時実行によるタプルの見え方

今度は、begin によってトランザクションを明示的に開始しているので、複数のＳＱＬ文が１つのトランザクション ID として実行される。
T2 を開始したら、T1 で update 文を実行している。その後、select 文を実行して、この状態でのそれぞれのトランザクションからのテーブルの見え方を表示している。左上の select 文の実行結果を見ると、id が 2 だったタプルが 3 に更新されて見える。id が 3 のタプルでは、xmax と cmin が 1 になっている。xmax と cmin に同じ値が入っているのは、union により同じ領域になっているためである。この場合は、cmin が有効であり、686 というトランザクション ID の 1 というコマンド ID によってこのタプルが生成されたことを示している。
T2 では、id が 2 だったタプルに、xmax=686、cmin=686、cmax=1 が入っている。このタプルは、他のトランザクションから削除された状態にあり､cmin は無効で xmax の値ということになる。ここで、xmax が入っているにもかかわらず、このタプルが表示されている理由は、686 というトランザクション ID がまだコミットされていないため、この xmax の値が有効とみなされていないためである。
この時点では、実際は図 5-3 のように３つのタプルが入っている。

Copyright © 2004 NTTDATA

```
  oid   | xmin | xmax | cmin | cmax | id
--------+------+------+------+------+----
 17170  |  683 |    0 |    0 |    0 |  1
 17171  |  684 |  686 |  686 |    1 |  2
 17171  |  686 |    1 |    1 |    0 |  3
```

図 5-3 ディスク上の実タプル

T1 の select 文はトランザクション ID が 686 で、コマンド ID が 2 である。id が 2 のタプルは見えず、id が 3 のタプルは見える。T2 の select 文では、xmin と xmax に入っている 686 というトランザクション ID がコミットされたものでないという理由で、id が 1 と 2 のタプルが見えているわけである。

話を図 5-2 に戻す。T1 の update が終わると、今度は T2 が update 文を発行している。次の見え方は、最初の update 文の逆なので説明は省略する。
最後に、T2 がもう１度 update 文を発行している。これは T1 が update で更新したタプルと衝突するため、T1 がコミットするかロールバックするまで実行が遅延される。この部分は、ロックマネージャで排他制御される。

まだロックマネージャやエグゼキュータを調査していないため、ここから先は、ロックテーブルを見ながら実行した結果からの推測になる[6]。update文で更新を行うと、更新前のタプルに対して排他ロック（ExclusiveLock）を取得する。更新後のタプルは、MVCCの仕組みで他のトランザクションから見えないので、特にロックは必要としない。参照や更新を行う際に、各タプルをxmin、xmaxなどを見ながらMVCCの仕組みでそのタプルの可視性を調べる。もし、そのトランザクションから見える場合、ロックマネージャに対して、タプルの共有ロック（ShareLock）を取りに行く。ここではじめて、ロックマネージャレベルでの排他制御を行う。
図 5-2 の最後の update 文を実行した状態で、ロックテーブルを確認すると、T1 が排他ロックを持っていて、T2 の共有ロックの取得が失敗していることが確認できる。

5.2. t_infomask の動作確認例

続いて、t_infomask の値がどのようになっているのか確認した例を示す。

postmaster を起動して、psql で接続を行い、次のような SQL 文を実行する。

```
create table foo ( id int );
insert into foo values ( 1 );          ---- ①
begin;
insert into foo values ( 2 );          ---- ②
rollback;
insert into foo values ( 3 );          ---- ③
update foo set id = 4 where id = 3;    ---- ④
```

その後、postmaster を停止するか、checkpoint を実行するなどして、キャッシュの情報をディスクに

[6] “select * from pg_locks” で確認できる。

Copyright © 2004 NTTDATA

フラッシュさせる。そして、次のように、データベースのディレクトリへ移動して、作成したテーブルのダンプを表示してみる。

```
$ cd $PGDATA/base/17142
$ od -x 17164
```

表示した結果は次のようになっているはずである。タプル内のデータ見方は、図 5-4 に示して後述するので、そちらを見てほしい。下線を引いた部分が、t_infomask である。

```
0000000  0000 0000 bb64 00a6 0017 0000 0024 1f80  ┐
0000020  2000 2001 9fe0 0040 9fc0 0040 9fa0 0040  ├ ページヘッダ
0000040  9f80 0040 0000 0000 0000 0000 0000 0000  ┘
0000060  0000 0000 0000 0000 0000 0000 0000 0000
*
0017600  0283 0000 0000 0000 0000 0000 0000 0000  ┐ 4タプル目
0017620  0004 0001 2810 001c 4310 0000 0004 0000  ┘
0017640  0282 0000 0283 0000 0000 0000 0000 0000  ┐ 3タプル目
0017660  0004 0001 0110 001c 4310 0000 0003 0000  ┘
0017700  0281 0000 0001 0000 0000 0000 0000 0000  ┐ 2タプル目
0017720  0002 0001 0a10 001c 430f 0000 0002 0000  ┘
0017740  0280 0000 0000 0000 0000 0000 0000 0000  ┐ 1タプル目
0017760  0001 0001 0910 001c 430e 0000 0001 0000  ┘
0020000
```

１タプル目が 1 を insert したもので、２タプル目は 2 を insert して rollback したものである。３タプル目は 3 を insert して、その後の update 文で、４タプル目に更新されている。
すべてのタプルは、0x0010 のフラグが立っているので、OID を保持していることがわかる。
１タプル目の t_infomask は、0x0910 になっており、9 のところの意味は、t_xmin がコミットされていることを示す 0x0100 と、t_xmax が無効であることを示す 0x0800 を合成したものであることが分かる。つまり、clog を確認しなくても、このタプルを作成したトランザクション ID 0x00000280 はコミットされていることが分かる。
２タプル目の t_infomask 0x0a10 の a の意味は、t_xmin がアボートされたことを示す 0x0200 と先ほどの 0x0800 の合成であることが分かる。先に示した②の inset 文が rollback されていることが正しく反映されている。
３つ目のタプルの 0x0110 の左側の 1 であるが、t_xmin がコミットされたことしか示していない。④の update 文が発行されているので、本当は、t_xmax がコミットされているのだが、ビットマスクには反映されていない。したがって、このタプルが次に読まれたときに、clog を使って t_xmax がコミットされているかどうかチェックされる。そのタイミングでコミットされていることが分かるので、キャッシュ中では、t_xmax がコミットされていることを示す 0x0400 と t_xmin がコミットされていることを示す 0x0100 が合成されて、0x0510 に更新されるはずである。これがディスクに反映されるタイミングは、このページに何らかの更新が発生したときになる。

Copyright © 2004 NTTDATA

４つ目のタプルは、0x2810 になっている。0x2000 は、更新を行ったタプルであることを示している。２つ目の 8 についてであるが、t_xmax が無効であることしか示していない。更新時にこのタプルを生成したときは、t_xmin はまだ有効でなかったので、t_xmin のコミットを示す 0x0100 は合成されていない。また、t_xmin 用のフラグは、初期化として 0x0200 にすることをしていないので、このような数値が入っている。次にこのタプルが参照されたタイミングで、このタプルは 0x2910 に変更される。もちろん、ディスクに反映されるのは、他の更新処理により、ページの書き換えが発生した場合である。

ここで、タプル内のデータの見方を示しておく。この例は、３タプル目を使っている。３章で示した、HeapTupleHeaderData 構造体と合わせて見てもらうと分かりやすいだろう。

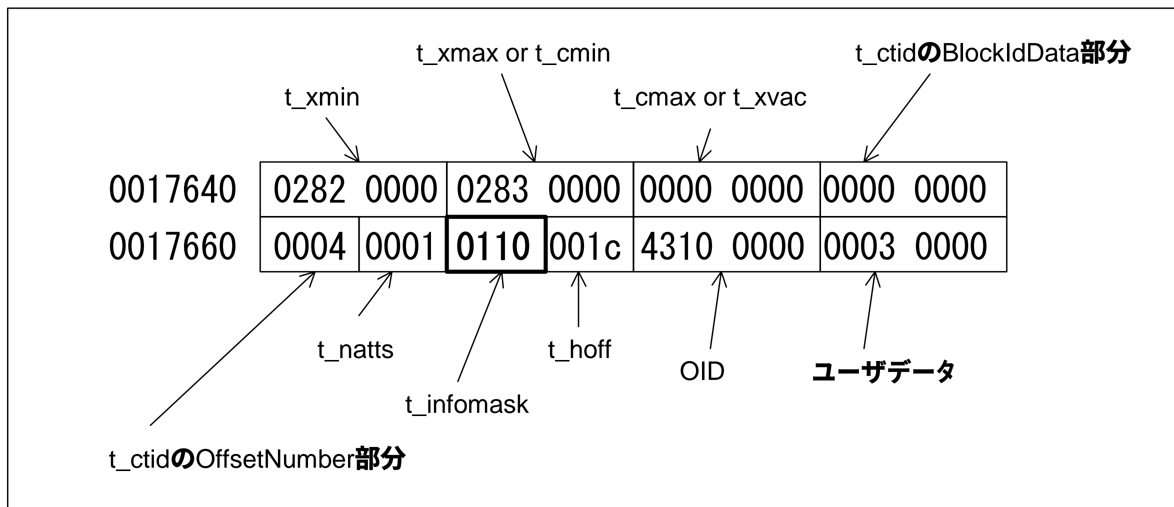

図 5-4 タプルのデータ構造

今回は、od –x でデータをダンプしたため、インディアンの関係で、４バイトのデータは上位２バイトと下位２バイトが逆に表示されている。この例では、t_xmin が 0x00000282、t_xmax が 0x00000283 ということである。

6. その他

この辺りのソースコードから分かる PostgreSQL の制限に関する話題に触れておく。

6.1. カラム数の上限について

図 3-1 に示したように、タプルに NULL 値を含んでいる場合にだけ、NULL 値判定のためのビットマップを持つ。通常のタプルのヘッダに、この NULL ビットマップとアラインメント調整のためのスペースを含めて、ヘッダサイズとして t_hoff という uint8 型の変数に値を保持している。この NULL ビットマップの最大数が最大カラム数となっている。
NULL ビットマップとアラインメントを除くと、多くの 32 ビットマシンでは、23 バイトがヘッダとして使用される。一部の 64 ビットマシンを考慮して、倍の 46 バイトにアラインメントを考えて 48

Copyright © 2004 NTTDATA

バイトとし、これを uint8 型の最大値 256 から引いて 208 を算出する。そこで、208 バイト×8 で、1664 ビットの NULL ビットマップが保持可能と計算している。
これを元に、タプル中の属性数の制限として、次のような定義がしてある。

```
#define MaxTupleAttributeNumber 1664    /* 8 * 208 */
```

作業カラムなどに余裕を持たせるため、実際にユーザが使用できるカラム数を制限は、MaxHeapAttributeNumber として、次のように定義されている。

```
#define MaxHeapAttributeNumber  1600    /* 8 * 200 */
```

実際に、1000 以上のカラムがある場合、ブロックサイズの制限にかかりやすくなるために、これで十分であるだろうということが、ソースコード中のコメントに書かれている。

＜ＥＯＦ＞

Copyright © 2004 NTTDATA