PostgreSQL 解析資料

～ *ストレージマネージャ (2)* ～

(株) NTT データ
ビジネス開発事業本部 システム方式技術 BU
井久保 寛明

---

1. はじめに

本資料では、PostgreSQL の 狭い意味での ストレージマネージャについて紹介する。ストレージマネージャというと、インデックスのデータ構造からファイルの格納、I/O の実装まで含めて広く言う場合もある。ストレージマネージャ(1) で、storage/smgr を中心とした、OS に対して I/O を起こすような低レベルの部分を紹介している。今回は、storage ディレクトリ以下にある、ページ管理およびバッファ管理の部分について紹介する。
現時点では、インデックスやヒープの調査が終わっていないため、上位レイヤからの使い方の説明がうまくできてないところが多々あると思うが、あらかじめご了承願いたい。

1.1. 対象モジュール

対象としているのは、具体的には storage/page、storage/buffer、storage/freespace 以下のモジュールである。

1.2. 対象バージョン

本資料は、PostgreSQL7.3.4 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。

1.3. 用語の説明

本ドキュメントで、紛らわしい用語を簡単に説明しておく。

ページ	固定長サイズのメモリ領域。バッファ１枚分のメモリ領域の単位。
ブロック	ディスク上でのページのこと。

2. storage/page

storage/page 以下のコードは、バッファページマネージャの実装である。データバッファのページ内のデータを操作するためのモジュールが実装されている。

2.1. item

item とは、ページ内のオブジェクトである。別の言い方をすると、ページ内の１つのデータエントリである。オブジェクト自体は、データレコードであったり、インデックスのエントリであったりする。バッファページマネージャでは、item のオブジェクトが何かということは特に規定していない。ページ内でのデータエントリの操作を行うためインタフェースを提供している。

ItemPointer とは、ディスクページとオフセット番号を組み合わせたItemPointerData構造体へのポインタである。item の論理位置を指定することで、同一ヒープ[1]内でitem を一意に指定できる。ItemPointerData 構造体は、次のように定義されている。

```
typedef struct ItemPointerData
{
      BlockIdData ip_blkid;
      OffsetNumber ip_posid;
}

typedef ItemPointerData *ItemPointer;
```

ip_blkid は、リレーション内のブロック番号である。**ip_posid** は、ページ内[2]でのオフセット番号であり、ページ内のitemの位置をバイトオフセットではなく、1～nまでの番号で示している。

ページ内では、item は、実データ領域へのページ内オフセットと実データのサイズ、フラグを組み合わせた ItemIdData と呼ばれるデータ構造で管理される。この ItemIdData の構造は次のようになっている。ItemIdData は、ソースコード中で、line pointer という呼び方をされていることもある。

```
typedef struct ItemIdData
{                              /* line pointers */
    unsigned    lp_off:15,     /* offset to start of tuple */
                lp_flags:2,    /* flags for tuple */
                lp_len:15;     /* length of tuple */
} ItemIdData;

typedef ItemIdData *ItemId;
```

ItemIdData は、ページ内の item データのオフセット **lp_off**、item データの長さ **lp_len**、item のフラグ **lp_flags** から構成されている。つまり、ページ内の位置とデータ長とフラグを管理する固定長のデータである。

---

[1] ヒープとは可変長のデータ格納用のデータ構造。テーブルごと、インデックスごとに割り当て、あるテーブルの属性をまとめて格納する目的で使用される。

[2] ip_blkid で示されたブロックのこと。メモリに読み込まれるので、ページという呼び方にしている。

ページ内のオフセットが 15bit であることから、PostgreSQL の最大ブロックサイズは 32KB に制限されている。

lp_flags フラグには、ItemIdData が使用されているかどうかを示す LP_USED（0x01）と、ItemIdData が指し示す item が有効かどうかを示す LP_DELETE（0x02）の 2 つのビットがある。LP_DELETE ビットはインデックスページでのみ使用される。PostgreSQL ではヒープからレコードが削除されてもインデックスにはエントリが残ったままになるため、インデックスの効率が悪い場合がある。その対策として LP_DELETE フラグが用いられている。インデックススキャン中にヒープタプルが削除されていたインデックスエントリを検出すると、そのエントリの ItemIdData の LP_DELETE ビットをセットする。これにより次回以降のインデックススキャンでは、ヒープタプルが削除されているインデックスエントリを効率よく読み飛ばすことが可能になる。

2.2. ページの構成

ページヘッダには、LSN[3]、最終更新のID(トランザクションID), ページ内データの始点/終点、特殊データの始点、ページデータのサイズを持っている。ページヘッダの構造体は次のようになっている。

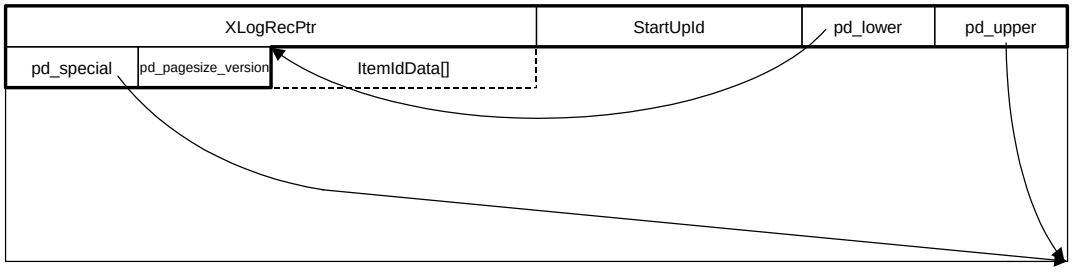
```
typedef struct PageHeaderData
{
    /* XXX LSN is member of *any* block, not */
    /* only page-organized - 'll change later */
    XLogRecPtr  pd_lsn;         /* LSN: next byte after last byte of xlog */
    /* record for last change of this page */
    StartUpID   pd_sui;         /* SUI of last changes (currently it's */
    /* used by heap AM only) */

    LocationIndex pd_lower;     /* offset to start of free space */
    LocationIndex pd_upper;     /* offset to end of free space */
    LocationIndex pd_special;   /* offset to start of special space */
    uint16      pd_pagesize_version;
    ItemIdData  pd_linp[1];     /* beginning of line pointer array */
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```

pd_pagesize_version には、ページのサイズとページ構造のバージョン番号が入っている。0xFF00 部分にサイズ、0x00FF 部分にページレイアウトのバージョンが入っている。7.3.4 では、ページレイアウトのバージョン PG_PAGE_LAYOUT_VERSION = 1 である。

初期化直後は、ページヘッダの指すポインタは次のようになっている。



[3] Log Serial Number の略で、ログエントリのIDである。

図 2.1

実際に item が入ってくると次の図ようになってくる。

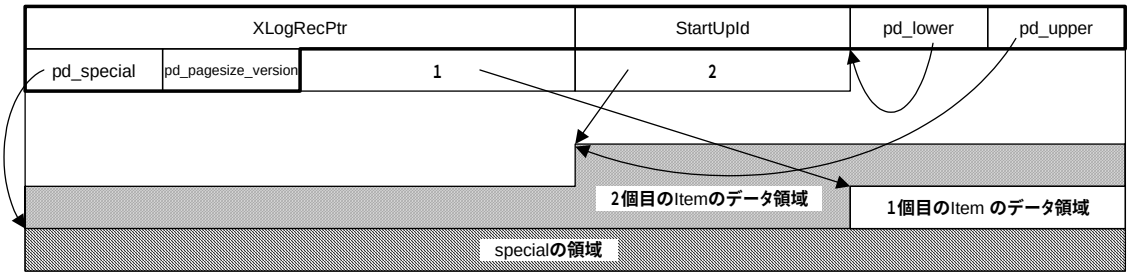


図 2.2

pd_lower が ItemIdData の最後を指し、**pd_upper** が item データのエントリ領域の先頭を指す。つまり、pd_lower と pd_upper は、それぞれページ内の空き領域の下限と上限を指している。新規のエントリを割り当てるときは、ItemIdData は pd_lower 側から、item データは pd_upper 側から割り当てられる。

pd_special は special データの先頭を指すことになる。ただし、special の領域は、ヒープの場合使われない。これは、インデックスのノードなどの場合に使用される。

ItemIdData は、１個目２個目それぞれの item データの位置を示している。

2.3. ItemPointer の実装（itemptr.c）

ItemPointer を比較する関数が１つ定義されているだけである。

2.4. バッファページの実装（bufpage.c）

PostgreSQL は、追記型のデータベースなので、ページ内の Item データに対して、追加の概念はあるが、データページの削除の概念はない。ただし、インデックスのデータエントリ削除のための関数は存在している。基本的にはデータが追記され続けるので、インデックスからもエントリが削除されることはない。インデックスが削除されるのは特殊な場合、つまりバキューム時にのみである。

どの関数も、データを書き換えるものに関しては、ページヘッダのチェックを厳しく行っている。理由は、壊れたページのためポインタがおかしくなっていると、他のデータ領域も破壊してしまうためである。

PageInit()	ページ内を初期化する関数
PageHeaderIsValid()	ページヘッダが有効な値かチェックする関数。HDD から読み込んだときに、データが壊れていないかチェックする。
PageAddItem()	ページ内に item を追加する
PageGetTempPage()	ページのコピーを作成して、special とヘッダ以外の領域を初期化する
PageRestoreTempPage()	PageGetTempPage() で作成したページを、バッファに書き戻

	す。
itemoffcompare()	PageRepairFragmentation() の補助関数。オフセットの比較に使う
PageRepairFragmentation()	item のデータ領域にあるフラグメントを取り除く。 ItemIdData 自体は、何も変更しない。
PageGetFreeSpace()	ページ内の空き領域のサイズを計算する。
PageIndexTupleDelete()	index ページだった場合に、item を１件削除する。

PageAddItem() では、ページに入らなかったら、すぐエラーで返す。基本的に、ページ内に十分な空き領域があることを確認してから PageAddItem()を呼ぶようになっている。
また、PageAddItem()では、領域を詰めるなどということはしていない。PageRepairFragmentation()を実行して領域をつめるのは、主にバキューム処理の時である。
PageIndexTupleDelete() では、データを削除する際に、データを詰める処理も行っている。

3. storage/buffer

storage/buffer 以下のコードは、バッファマネージャを実装したモジュールである。バッファには、共有バッファ(Shared Buffer) とローカルバッファがある。共有バッファは、共有メモリ上に置かれ、ディスク上のデータをバッファリングするために使用される。ローカルバッファは、１つのバックエンドプロセス内で使用され、主に、一時結果の保存するテンポラリテーブルに使用される。これらのバッファは、トランザクションログやその他のコントロールファイルなどのバッファとして使われることはない。

バッファは、複数のバックエンドから同時にアクセスされるため、共有メモリやロックのメカニズムと密接な関係がある。また、パフォーマンスを考慮して、多少複雑なロック機構を構成していることが、ソースコードを読むのを難しくしている。

バッファマネージャの実装は、次のようなファイルから構成されている。

buf_init.c	バッファマネージャの初期化モジュール。共有バッファ全体を初期化するために postmaster から１回だけ呼び出す InitBufferPool() と、バックエンド毎のバッファへアクセスする変数などの初期化を行う InitBufferPoolAccess() などを含んでいる。
buf_table.c	共有バッファに高速にアクセスするためのハッシュテーブルを管理するモジュール。ハッシュ関数としては、tag_hash() <utils/hash/hashfn.c> を使用している。
bufmgr.c	バッファマネージャのメインモジュール。共有バッファ専用の部分とローカルバッファと共有の実装を含んでいる。
freelist.c	バッファデスクリプタをリスト構造で管理するモジュール。 ※ バッファが使用中の状態(PIN の立てられた状態)以外では、バッファはフリーリストに入れられている。dirty なバッファは、フラグが設定されていて、再利用される前に書き出される。
localbuf.c	bufmgr.c の内、ローカルバッファ固有の実装部分。

3.1. バッファマネージャに関する予備知識

バッファは、トランザクションの ACID 特性を実現するために、ログやロックなどと非常に密接な関連を持つことになる。PostgreSQL で使われている方式について、はじめに簡単に説明しておく。

3.1.1. Write Ahead Logging (WAL)

最近主流のRDBMSでは、データのDurability(永続性)を保証するために、データボリューム(ディスク)にデータを書き込むだけでなく、ログに更新情報を記録していく。更新がある毎にデータをディスクに書き込むと、実行速度が非常に遅くなるので、DBMSではバッファを用いてデータやログをキャッシュしている。キャッシュからディスクに書き出す際に、データとログの書き込みのタイミングで、必ずログに更新情報を書き込んでからデータを書き込む方式のことを**Write Ahead Logging** (WAL)と呼

ぶ。
データとログの書き込みのタイミングはロギングの方式に依存するが、PostgreSQL で採用している undo/redo ロギング[4]では、コミットのタイミングに依存せずにバッファ内のデータをディスクに書き出すことができる。バッファをディスクに書き出す際は、必ず Write Ahead Logging のルールに従って、これから書き出そうとしているバッファ内の全ての更新に関するログをディスクに書き出してから、バッファをディスクに書き出さなければならない。undo/redo ロギングの特徴として、コミット時にはログの書き出しまで行ってあればよい。つまり、データバッファに関しては、別のタイミングで書き出しを行うことができる。また、コミット前でも、ログを先に書き出しておけば、データバッファ内の情報をディスクに書き込むこともできる。
PostgreSQL では、7.1 から Write Ahead Logging が採用され、このことによって信頼性と性能が大幅に向上した。

3.1.2. トランザクションとバッファロック

一般的に、DBMSでは、トランザクションのACID特性のうち、Isolation(独立性)を実装する方式としてロックを用いる場合、２層ロック(2Phase Lock)が有名である[5]。２層ロックとは、ロックを取得するフェーズとロックを開放するフェーズに分ける方法で、トランザクションでの個別のSQLなどの実行中はロックを取得するだけで、コミット時にまとめてロックは開放される。
PostgreSQL のように、行レベルロックを実装している場合、この２層ロックで管理するのは、あくまでデータベースの行データである。バッファロックは、バッファを操作する際に取得するロックであるが、トランザクションとは無関係に取得／開放ができる。そうは言っても、トランザクション終了時には、全てのバッファロックが開放される。
以上のように、行ロックとバッファロックは、それぞれまったく異なる目的で使用される。行ロックはトランザクションの Isolation を実現するために使用され、バッファロックは主に同時更新によるデータの破壊を防ぐために使用される。

3.2. 共有バッファの構造

PostgreSQL では、次のようなバッファの構造になっている。

[4] undo/redo ロギングとは、undo 情報と redo 情報をログに書き込む方式である。その他の代表的な方式では、undo ロギング、redo ロギングなどがある。ロックの単位やコミット時の処理方法が異なってくる。PostgreSQLの場合、追記型の特性上undo情報は書き込まないが、ログ・リカバリの方式としては、undo/redo ロギングに分類できる。
[5] PostgreSQL のロックの方式は現時点で未調査のため、一般論で書いておきます。

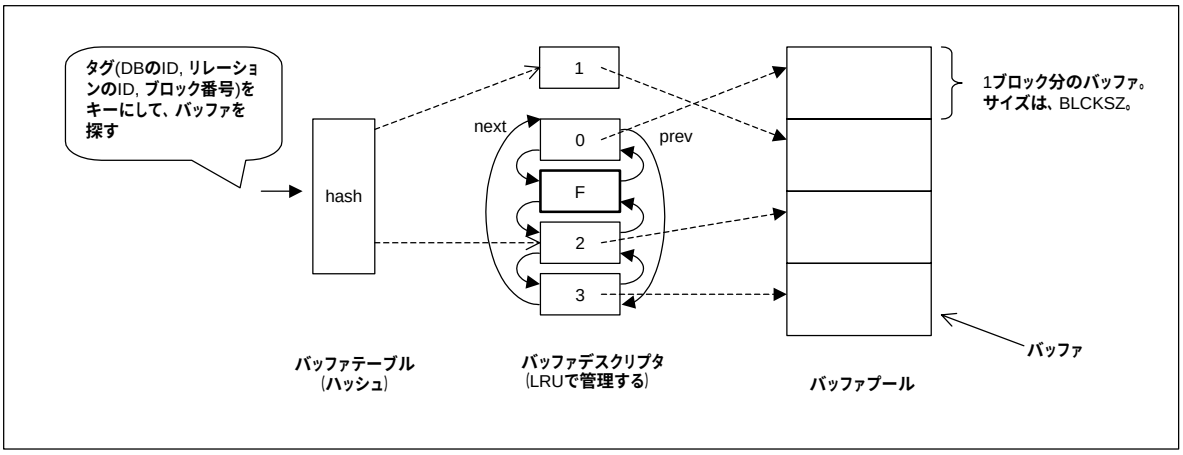


図 3.1

図中の一番右にあるバッファプールは、サイズが [ブロックサイズ(BLCKSZ)] × [バッファ数(NBuffer)] の領域として、物理的に１つにまとめて共有メモリ中に取られている。バッファプールに含まれるバッファは、ディスク上のリレーションのデータをブロック単位で読み込んでくるために使用される。したがって、個々のバッファの中は、２章で示したページバッファのような構造になっている。

各バッファを管理するデータ構造がバッファデスクリプタであり、１つのバッファにつき、対応するバッファデスクリプタが１つ存在している。この図では、バッファデスクリプタはリスト構造で書いているが、バッファデスクリプタもバッファと同様にまとめて共有メモリ中に取られている。これについては、フリーリストのところで詳しく説明する。

バッファの取り込んでくるディスク上のブロックは、データベースの ID、リレーションの ID、ブロック番号がわかれば一意に識別できる。ソースコード中では、このブロック識別の３つの情報を組にしたものを**バッファタグ**と呼んでいる。このタグに対応するバッファが既に存在するかどうかを高速に検索する手段として、図の中の左にあるバッファテーブル(ハッシュテーブル)を使用する。このハッシュも共有メモリ中に作成されている。

参照または書き換え中でないバッファ(PIN の立っていないバッファ)のバッファデスクリプタは、フリーリストにつながれている。図中では、1 となっているバッファデスクリプタが参照または書き換え中で、残りがフリーリストである。

バッファ内に読み込み済みのブロックのタグは、バッファテーブル(ハッシュ)に登録されている。バッファは、バッファデスクリプタによって LRU で管理されていて、LRU から追い出される際に、バッファタグがハッシュから削除される。

3.2.1. 共有バッファの初期化

共有バッファの初期化は２種類ある。１つは、共有バッファを共有メモリ中に作成し、その他の共有する情報(バッファテーブルやバッファデスクリプタ)を初期化する InitBufferPool() である。この関数は、postmaster から１回だけ呼び出される。２つ目は、各バックエンドプロセスが個別に管理している共有バッファにアクセスを補助するための変数を初期化する InitBufferPoolAccess() である。こちらの関数は、バックエンド初期化時に１回だけ、つまりバックエンドが起動されるたびに１回実行されている。

3.2.2. プロセスでローカルな情報と共有情報

本来ならば、共有バッファを管理する情報は、共有情報として共有メモリ中に１つだけあればよい。しかし、共有メモリ中の情報にアクセスするためには毎回ロックを取る必要があり、オーバヘッドが大きいだけでなくロックの競合による待ち時間が大きくなる。そのため、共有情報へのアクセス数を減らすために、一部の情報はローカルなところで管理している。

バッファの参照数がそれにあたる。共有メモリ中のバッファの参照数 refcount は、実際はバッファを参照しているバックエンドの数である。これに対して、バックエンドごとに、そのバックエンドプロセスでのバッファの参照数を PrivateRefCount として管理している。実装に合わせて、バックエンドごとの参照数という言い方をしたが、トランザクションごとのバッファの参照数と考えるとわかりやすい。

PrivateRefCount が 0 のバッファにアクセスする場合、PrivateRefCount をインクリメントするだけでなく、ロックを取得して共有メモリ中の refcount を１つインクリメントする。refcount が１以上になっていれば、他のバックエンドから使用中であることがわかるので、各バックエンドプロセスで PrivateRefCount が１以上のバッファにアクセスするなら、PrivateRefCount のみをインクリメントする。

バッファへのアクセスが終わって参照数を減らす場合、PrivateRefCount が２以上の場合は PrivateRefCount のみをデクリメントし、PrivateRefCount が１の時のみロックを取得して共有メモリ中の refcount をデクリメントする。

3.2.3. PIN

PostgreSQL でいう PIN とは、バッファをロックする仕組みの１つであり、バッファを参照していることを他のプロセスに知らせるためのものである。主に、アクセス中のバッファが他のバックエンドによって勝手にフラッシュされることを防ぐ目的で使われている。前述の refcount と PrivateRefCount がこの実装にあたる。更新などでバッファをロックするためには、PinBuffer() で PIN を設定するのではなく、LockBuffer()でバッファコンテキストロック(後述)を取得する必要がある。

教科書レベルの書籍では、PINとは、バッファを書き出すためにディスクI/O を起こす際に取得するロックになっていることが多く、PostgreSQL でいうところの io_in_progress[6] と同じなので注意が必要である。

PIN を設定することを英語で pin という動詞で書かれるために、「PIN 留めする」とか「PIN を立てる」という日本語を使われることが多い。

PostgreSQL では、同一プロセスから同じバッファに対して何度でも PIN を設定できるようになっている。また、PIN は比較的長い時間保持したままになっていても問題ない。例えば、JOIN 演算の実行中、外側のループのバッファであっても、内側の演算が終わるまで保持していても構わない。これは、PIN が排他ロックのような働きをするものではないからである。

3.3. フリーリスト管理 （freelist.c）

一般的に「バッファのフリーリスト」というと、未使用のバッファを管理する意味としてとられることが多い。PostgreSQL のバッファマネージャでは、バッファ内を参照中で内容を書き換えられては困るときやバッファを書換えるときに PIN を設定するが、PIN が設定されていないバッファは全て常に

[6] BM_IO_IN_PROGRESS フラグで実装されている。

フリーリストつながれる。従って、フリーリストにつながれた状態で、キャッシュとしての機能を果たす。また、書き換えられて dirty な状態のバッファも、PIN が設定されていなければフリーリストにつながれている。バッファが不足して、あるバッファを別の内容のために再利用しようとした際に、そのバッファに dirty フラグが設定されていたら、バッファの内容をディスクに書き出してから再利用する。

フリーリスト自体は、バッファデスクリプタの双方向リストである。使用するときに最初の PIN が立てられると同時にフリーリストから取り出され、PIN による参照がすべて開放されたときにフリーリストに戻される。フリーリストにつながれている状態でも、１度使われたバッファデスクリプタの先にあるバッファには、ブロックの内容がキャッシュされている。そのバッファが dirty であるかどうかは、バッファの書き換えを行ったかどうかに依存するので、その状態は不明である。

フリーリストの管理について、図を使って説明する。

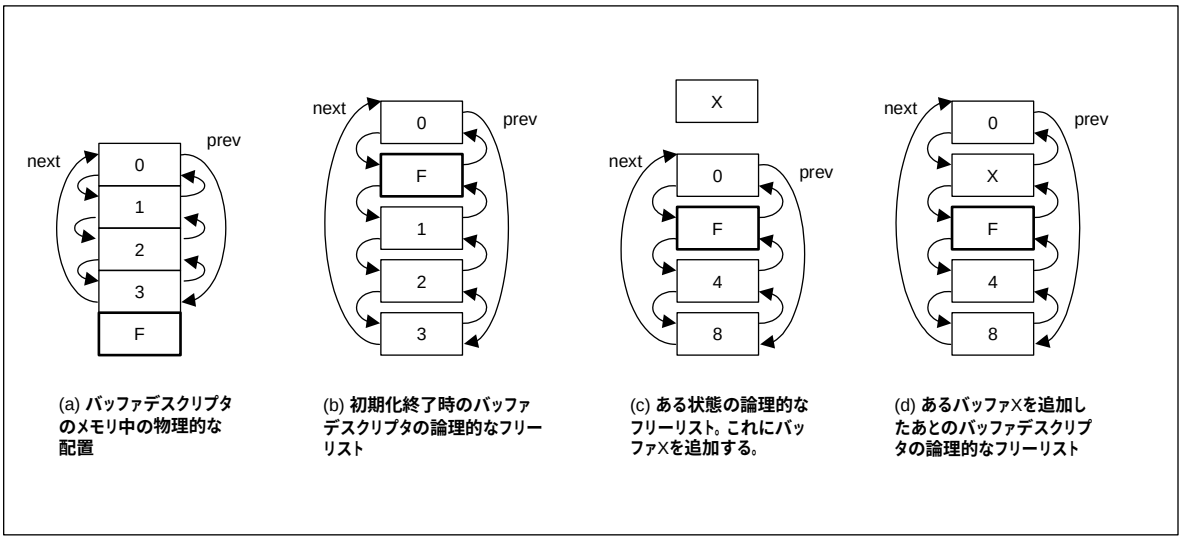


図 3.2

まず、InitBufferPool()で共有バッファの初期化しているとき、InitFreeList()[7] が呼び出される直前では (a) のような状態になっている。この図だと、バッファ数NBuffers が４であり、バッファデスクリプタ 0 ～ 3 とフリーリスト管理用のデスクリプタ F が (a) のような物理的な配置で取られる。そして 0 ～ 3 のバッファだけリング状の双方向リストになっている。
次に、InitFreeList() で バッファデスクリプタ F を リング状のバッファの適当な位置に入れる。現状の実装では、0 と 1 のバッファの間に入れるようになっているので、(b) のようになる。ちなみにこの図は(a)とは異なり、リスト構造の論理的は並びで、メモリの物理配置を表したものではない。
バッファをフリーリストから外す場合は、フリーリスト管理デスクリプタ F の次のノードから使用されるので、(b) でいうと 1 が使用されることになる。

(c) は、バッファを使っているうちに、このようなフリーリストになっていたとする[8]。そのとき、バ

[7] InitFreeList()は、InitBufferPool()の最後のほうで呼び出される
[8] この図は、(a),(b)の例のように、NBuffersが 4 ではない。もっとバッファが多いばあのある状態として書いてある。

ッファデスクリプタ X をこのフリーリストに戻そうとしているところである。バッファデスクリプタ X をフリーリストに加える場合は、F の前のところに入れられるので、(d) のようになる。これがLRUを実現している部分である。

3.4. バッファ入れ替えのロジック

バッファの入れ替えは、バッファが LRU のリストから追い出されるタイミングでのみ行われる。バッファから使用したいページがない場合、新しいブロックをディスクから読み込まなければならない。ディスクから読み込むためのバッファは、LRU の一番後ろのバッファを使用する。もし、このバッファに書換えが行われていて、dirty フラグが立っていた場合は、バッファをディスクに書き出してから使用しなければならない。ディスクに書き出す前には、このページのログが全て書き出されているかのチェックが行われる。まだ書き出されていなければ、このページまでの LSN のログをフラッシュする。
ログの書き出し、ディスクの書き出しが終わったら、このバッファを使用することができる。
現時点では、バッファを入れ替えるタイミングは、LRU から追い出されるタイミングだけである。
Postmaster 終了時には、dirty フラグの立っているデータの書き出しが行われる。

3.5. バッファマネージャのインタフェース

バッファマネージャのメインモジュールである、bufmgr.c に定義されているバッファマネージャのインタフェースは次のようなものである。

BufferAlloc()	バッファテーブルからバッファを探す。バッファ内から見つからなければ、そのバッファのための領域を追加する。しかし、メモリへのデータの読み込みは行わない。
ReadBuffer()	BufferAlloc() に似ているが、キャッシュにのっていない場合、データの読み込みを行う。
ReleaseBuffer()	バッファの PIN を外す
WriteNoReleaseBuffer()	dirty フラグを立てる。PIN はそのままで外さない。ディスク I/O は、バッファの入れ替えまで遅延される。
WriteBuffer()	WriteNoReleaseBuffer() + ReleaseBuffer()
BufferSync()	バッファプール中の全ての dirty バッファをディスクにフラッシュする。
InitBufferPool()	バッファモジュールの初期化を行う

バッファへのデータの読み込みは、通常は ReadBuffer() が利用される。BufferAlloc()は、バッファの再初期化のために使われ、現在のディスクの内容(バッファの内容)がどうなっているかは気にしない。
バッファの書き出しは、PIN を解放しない WriteNoReleaseBuffer() と PIN の開放まで行う WriteBuffer() が使い分けられる。どちらにしても、LRU から追い出されるか BufferSync()まで、実際の I/O は遅延される。
BufferSync() は、チェックポイントの実行や postmaster の終了時などに使用される。

3.6. バッファテーブル（ハッシュ）のインタフェース （buf_table.c）

buf_table.c に実装されているバッファテーブルのインタフェースは次のとおりである。

InitBufTable	ハッシュテーブルを共有メモリ中に作成する。
BufTableLookup	バッファタグに該当するバッファを探す。
BufTableDelete	ハッシュから引数で与えられたバッファタグのエントリを削除する
BufTableInsert	ハッシュにバッファタグをキーにして、バッファ ID を登録する。

この表から想像できるとおり、バッファタグが示すブロックがバッファプール中にあるかどうかを高速に調べるためだけの仕組みである。バッファにブロックのデータを読み込んだ際に、BufTableInsert() を実行し、バッファからこれまで保持していたブロックの内容を追い出す際に BufTableDelete() を実行する。
ハッシュ自体は、utils/hash/dynahash.c で実装されているハッシュを使用しており、ハッシュ関数には、utils/hash/hashfn.c の中の tag_hash() を使用している。

3.7. 共有バッファ管理のデータ構造 （バッファデスクリプタ）

バッファを管理するデータ構造について説明する。バッファの管理構造で最も重要なのは、バッファデスクリプタである。バッファ内部には、ディスク上でバッファ内のデータを管理するために最低限必要なヘッダ情報がある。バッファデスクリプタは、これらのバッファをメモリ中で管理するために必要なヘッダ情報といったところである。

```
typedef struct sbufdesc
{
    Buffer      freeNext;       /* フリーリストの双方向リンクのポインタ */
    Buffer      freePrev;
    SHMEM_OFFSET data;          /* バッファプールのデータへのポインタ */

    /* tag と buf_id の組で、ハッシュテーブルから探すのに使用する */
    BufferTag   tag;            /* リレーション ID/ブロック番号 で付けられた識別タグ */
    int         buf_id;         /* バッファのインデックス番号 (0 はじまり) */

    BufFlags    flags;          /* ビットの定義は、後述のバッファフラグ */
    unsigned    refcount;       /* PIN を立てているバックエンドの数 */

    LWLockId    io_in_progress_lock;    /* I/O が終わるのを待つためのロック */
    LWLockId    cntx_lock;      /* ページのコンテキストにアクセスするためのロック */

    bool        cntxDirty;      /* ブロックが dirty だというしるしをつける新しい方法 */

    BackendId   wait_backend_id;    /* pin の数が 1 になるのを待っているバックエンドの ID */
} BufferDesc;
```

freeNext, **freePrev** は、フリーリストの双方向リンクを実現するものである。フリーリストから外れた場合は、INVALID_DESCRIPTER が設定されている。**data** は、バッファプールのバッファへのポインタである。
tag は 前述のバッファタグである。バッファに保存されているデータブロックがどのリレーションのどのブロックかを示している。従って、バッファの内容が入れ替わるたびに値が変わる。**buf_id** は個々

のバッファに付けられた ID であり、初期化時に設定される。
flags は、バッファの状態を示すフラグである。フラグの種類は、次のようなものがある。

BM_DIRTY	dirty フラグ
BM_PRIVATE	未使用。現在のソースコード中では使用されていない。
BM_VALID	フリーリスト管理用のバッファデスクリプタのみ 0 になっている。その他は、初期化時に 1 が設定されている。
BM_DELETED	バッファが削除されたことを示す
BM_FREE	空きバッファであることを示す。フリーリストにつながれているバッファデスクリプタには、BM_FREE フラグが設定されている。PinBuffer() でバッファに PIN を立て、フリーリストから外す際に BM_FREE フラグを解除する。
BM_IO_IN_PROGRESS	I/O 中であることを示す
BM_IO_ERROR	I/O エラーを出したバッファであることを示す
BM_JUST_DIRTIED	バッファをディスクに書き出すために I/O の最中に、バッファが書き換えられたことを示すのに使われている。
BM_PIN_COUNT_WAITER	バッファの PIN がなくなるのを待っているバックエンドがいることを示す。refcount をデクリメントした際に、refcount が 1 になっていたら、シグナルで通知する。

refcount は、バッファの参照数である。正しくは、バッファを参照しているバックエンドの数になる。PIN を実現するために使用される。
io_in_progress_lock と **cntx_lock** は、ロックマネージャでバッファのロックを実現するために使用される領域である。
cntxDirty は、バッファコンテキストロックに X-Lock を取得した段階でフラグを true にする。cntxDirty は、BufMgrLock を取らずにバッファの状態をチェックするために使用できる。
wait_backend_id は、PIN が 1 になるのを待っているバックエンドプロセスの ID を保存するのに使用される。物理的に item を消す場合は、他のバックエンドが PIN を立てていてはならない。従って、refcount が２以上の場合、他のバックエンドの立てた PIN がなくなるのを待たなければならない。このときに、wait_backend_id に自分の ID を設定して、BM_PIN_COUNT_WAITER フラグを立てておけば、PIN が 1 になったときにシグナルで教えてもらえるようになっている。現在、この機能は１つのバッファにつき１つしか待つことができない。しかし、この機能はオンライン VACUUM でしか使用せず、オンライン VACUUM は１つのテーブルに対して同時に１つしか実行できないようになっているので、これで問題はない。

3.8. 共有バッファへのアクセスルール

ここでは、共有バッファにアクセスするためのルールを見ていく。

3.8.1. ロックの種類

バッファマネージャの管理するロックには次のようなものがある。

1. BufMgrLock

共有メモリで共有している、バッファの管理情報(バッファデスクリプタ、フリーリスト、ハッシュテーブル)を操作するときに必要とするロック。このロックは影響範囲が大きいため、I/O 中は、バッファ単位の I/O 用ロックである IO_IN_PROGRESS ロックを取得して、このロックは開放しなければならない。

2. IO_IN_PROGRESS
   実体は、バッファデスクリプタ中に設定されているフラグである。バッファを読み出したり書き出したりするときに I/O を開始する前に設定し、I/O が終了したら開放する。このロックは、I/O 中に他のバックエンドが読みかけのデータを使用しないようにするために使用される。
3. PIN (refcount)
   前述したとおり、バッファを参照していることを示すメカニズムである。
4. バッファロック（バッファコンテキストロック）
   バッファを S-Lock(Shared Lock)、X-Lock(Exclusive Lock)で管理する仕組みである。S-Lock は複数のバックエンドから同時に取得することができる共有ロックであり、X-Lock は同時に１つしか取れない排他ロックである。実装上、同一プロセスから２回ロックをかけようとしても、ロック待ちが発生するので注意が必要である。バッファロックを取得する前に、必ず PIN を設定しておかなければならない。
   cntx_lock を用いて実装されている。

3.8.2. バッファのアクセスルール

それぞれの関数は、次のようなルールで共有バッファにアクセスする。

1. ページ内のタプルをスキャンする場合とタプルコミットの状態(XID とステータスビット)を調べる場合、PIN とバッファロックが必要である。
2. あるトランザクションで 1 度タプルが見えたら、バッファロックはすぐに開放しなければならない。PIN を持ったままであれば、そのタプルのデータにはアクセスし続けることができる。つまり、削除されない。
3. 新しいタプルの追加や、xmin, xmax フィールドの変更を行うためには、そのタプルを含むバッファの PIN と X-Lock が必要である。
4. バッファに PIN と S-Lock しかなくても、コミットステータスビットを変更してよい。論理和で更新するのでほとんど矛盾を起こさないのと、単なるヒントとしてしか使用していないためである。
5. 物理的なタプルの削除や、ページ内の再配置を行うためには、PIN と X-Lock が必要である。さらに、PIN を持っている場合、バッファの参照数 refcount が 1 でなければならない。

3.8.3. cntxDirty と flags による dirty フラグの制御

cntxDirty は、バッファコンテキストロック(S-Lockと X-Lockのこと) でX-Lockを取得した段階でフラグをtrueにする。cntxDirty は、BufMgrLock を取らずにバッファの状態をチェックするために使用できる。また、すでにBufMgrLock を開放した後でも変更が可能である。cntxDirty は、BufferSync() 中で、バッファの書き出し smgrwrite() などを実行したら、ロック解放後に false に戻す。そのほか

でも、BM_DIRTYフラグを消せるタイミングで、false に戻している[9]。

一方、flags に BM_DIRTY フラグを立てるためには、BufMgrLock を必要とする。flags に BM_DIRTY フラグが立てられるのは、実際に WriteBuffer()でバッファを書き出すタイミングで ある。

3.9. ローカルバッファ

ローカルバッファは、他のバックエンドからはアクセスできない一時テーブル用のバッファとして使用される。共有バッファとは異なり、バッファ領域はバックエンドプロセスのローカルなメモリに取られる。ロックなどの制御が必要ない分、共有バッファより高速なバッファであると言える。ローカルバッファでも、共有バッファと同じインタフェースを使えるように、共有バッファがバッファ ID を０から振っているのに対して、ローカルバッファの ID は–2 から-3, -4 と負の数を振っている。図中のバッファデスクリプタの中に書いてある数字がバッファ ID である。
また、現在、ローカルバッファの数は、64 個で固定になっている。

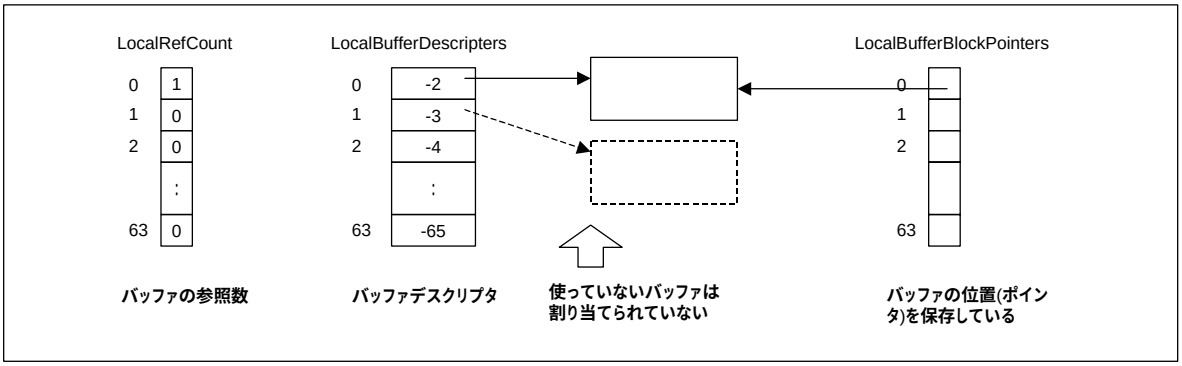


図 3.3

ローカルバッファは、全てのバックエンドで必要になるわけではないので、バッファデスクリプタの領域ははじめから確保してあるが、その先のデータ用のバッファ領域は必要になるまで割り当てない。

ローカルバッファは、ラウンドロビンを使用して管理される。ローカルバッファの書き出し WrireLocalBuffer() は、dirty フラグを立てるだけである。実際のバッファのディスクへの書き出しは、LocalBufferAlloc でのバッファ割り当て時に、ラウンドロビンで割り当てたバッファに dirty フラグが立っていたときのみである。

※ 今回の調査では、ローカルバッファが実際にどのような場面で使われているかは、判らなかった。キュエリの中間結果などの格納に使用されるはずである。

3.10. I/O のエラーハンドリング

各バックエンドプロセスは、同時に１つしか I/O を起こせないようになっている。これは、io_in_progress ロックが、プロセスあたり１つしか取れないようにして実現している。
バッファの I/O 実行中にエラーが起きた場合に、I/O 実行中のバッファをクリーン処理ができるように、

[9] 具体的に何をやっているところかは読めていない。多分、ロールバックやリカバリ処理ではないかと思う。

I/O 実行中のバッファを記録している。バッファの I/O を開始する際には、StartBufferIO() を呼び出して、バックエンドのグローバル変数 InProgressBuf に、I/O 実行中のバッファを設定する。バッファの I/O が完了したら、TerminateBufferIO() を呼び出して I/O 実行中のバッファを NULL に設定する。エラー発生時にクリーン処理を行うのが、AbortBufferIO() である。グローバル変数からバッファを特定し、ロックの解除などを行う。

3.11. バッファマネージャの問題点

現行のバッファマネージャには、いくつかの明らかな問題点がある。

1. バッファの管理が LRU のため、大規模な Sequential SCAN が走るとバッファからデータが追い出されてしまう。
2. バッファの書き出しのタイミングが、チェックポイント実行時とキャッシュから追い出されるタイミングしかない。
3. バッファ管理のオーバヘッドが大きいため、バッファを大きく取っても性能が十分に出ない？
4. BufMgrLock というジャイアントロックのため、アーキテクチャ上、パフォーマンスの問題があると考えられる

1 に関しては、7.5 に向けて ARC(Adaptive Replacement Cache) の実装が行われている。2 に関しても 7.5 で、バックグラウンドプロセスでバッファを書き出す方法が検討されている。3 に関しては、どの程度なのか不明である。4 に関しては、Linux で実行している限り、kernel 2.4 で 4CPU くらいまでしかスケールしないという問題に引っかかるので、今のところあまり目立たない。しかし、将来問題になってくるものと思われる。

4. storage/freespace

storage/freespace 以下のコードは、フリースペースマップ(FSM)の実装である。FSM は、リレーション中の空き領域をすばやく見つけるために利用される。

FSM は、どのリレーションの、どのページに、どれだけの空き領域があるかを管理するための機能だが、全てのリレーションについて管理しているわけでもなく、また、管理しているリレーションについても、全ての空き領域のあるページを管理しているわけでもない。
管理するリレーションは、ページ要求が過去にあったもので、優先度のリストに残っているものに限られる。また、管理するページは、リレーションごとに定めたスレッシュホールドの値よりも空き領域の多いものに限られている。

4.1. データ構造

FSM を管理する構造体は3つある。1 つ目は、管理情報をまとめた FSMHeader で、共有メモリ中に 1 つだけ存在する。2 つ目は、管理するリレーションの情報のための構造体で、FSMRelation である。これは、双方向リストを使ってリレーション情報の優先度リストを構成する。また、FSMRelation は、高速に検索できるようにハッシュに登録する。3 つ目は、ページ情報を格納するためのチャンクのための構造体 FSMChunk である。チャンクは、細かくメモリアロケーションをするのを防ぐ目的で導入されている。

3 つの構造体とハッシュから構成される FSM の全体のデータ構造は、次の図のようになる。

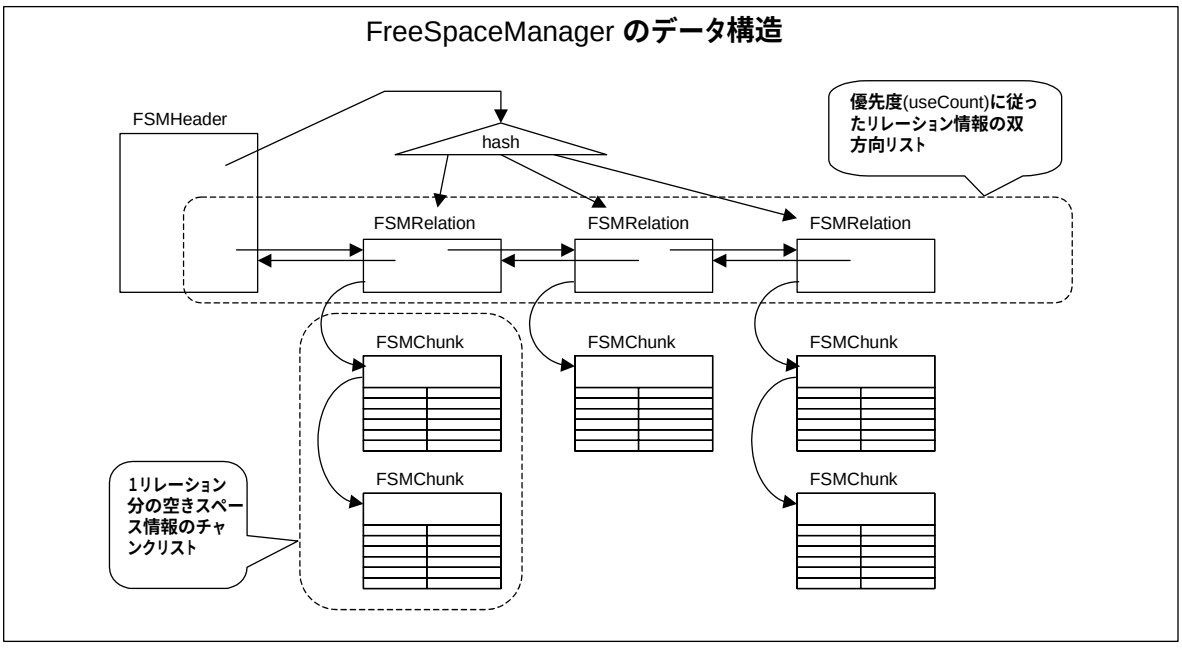


図 4.1

4.1.1. FSMHeader

FSM のオブジェクト全体を管理する構造体である。FSMRelation のリストのエントリ位置、チャンクのフリーリストの位置、ハッシュの位置を管理している。FSM に使われるデータは、全て共有メモリ上に配置されている。
FSMHeader 構造体は次のように定義されている。

```
/* Header for whole map */
struct FSMHeader
{
    HTAB       *relHash;        /* FSMRelation のハッシュテーブルのエントリポイント */
    FSMRelation *relList;       /* useCount 順に並んだ FSMRelations のリスト[優先度リスト] */
    FSMRelation *relListTail;   /* FSMRelation のリストの末尾へのポインタ */
    int         numRels;        /* 現在リストに保持している FSMRelations の数 */
    FSMChunk   *freeChunks;     /* チャンクのフリーリスト */
    int         numFreeChunks;  /* フリーリストのチャンク数 */
};
```

FSMRelation のハッシュテーブルのエントリポイント、FSMRelation の双方向リスト、チャンクのフリーリストをポインタで管理している。

4.1.2. FSM リレーション (FSMRelation)

リレーションごとの情報を管理する構造体である。優先度を示す双方向リストで繋いで管理するだけではなく、高速に検索する必要もあるのでハッシュにも登録する。ハッシュキーは RelFileNode の値である。従って、論理的 ID ではなく、物理的なリレーション ID を使うことになる。

```
struct FSMRelation
{
    RelFileNode key;            /* hash key (must be first) */
    FSMRelation *nextRel;       /* useCount 順のリストのポインタ */
    FSMRelation *priorRel;      /* useCount 順のリストの逆方向のポインタ */
    int         useCount;       /* use count for prioritizing rels */
    Size        threshold;      /* minimum amount of free space to keep */
    int         nextPage;       /* index (from 0) to start next search at */
    int         numPages;       /* total number of pages we have info
                                 * about */
    int         numChunks;      /* number of FSMChunks allocated to rel */
    FSMChunk   *relChunks;      /* linked list of page info chunks */
    FSMChunk   *lastChunk;      /* last chunk in linked list */
};
```

双方向の優先度リストを構成するために使用するポインタが、**nextRel** と **priorRel** である。
useCount は、そのリレーションに対する空き領域の要求があった数で、優先度を決める際に使用される。**threshold** は、空き領域として登録するサイズを決める際のスレッシュホールド値である。スレッシュホールド値に満たないサイズの空き領域しかないページは登録しない。

4.1.3. FSMChunk

チャンクの管理をする構造体である。リレーションの各ページの情報は、メモリアロケーションのオーバヘッドを減らすために、チャンクである程度まとめて保存している。
リストの一番後ろ以外のチャンクも、空きがあることを許している。これによって削除する速度を上げている。システム全体で、合計の空き容量が１チャンク分になったら、空き領域の回収を行う。それ以前に空き領域の回収を行うタイミングはない。

チャンクのデータ構造は次のようになっており、1 チャンクあたり 32 ページ分の情報を保存している。１ページ分のエントリは、ページのブロック番号と空き領域のサイズである。

```
#define CHUNKPAGES  32          /* each chunk can store this many pages */

struct FSMChunk
{
    FSMChunk   *next;           /* linked-list link */
    int         numPages;       /* number of pages described here */
    BlockNumber pages[CHUNKPAGES];      /* page numbers within relation */
    ItemLength  bytes[CHUNKPAGES];      /* free space available on each
                                         * page */
};
```

チャンクの物理的な配置は、図 4.2 のようになる。チャンク用の領域は、初期化時に１つだけまとめて取られる(図 4.2 では、中央の箱の集まりがまとめて１つで確保される)。実行中は、これ以上増えることはない。１つのまとめて確保されたメモリ領域は、FSMChunk 構造体のサイズごとに分けて使用される。

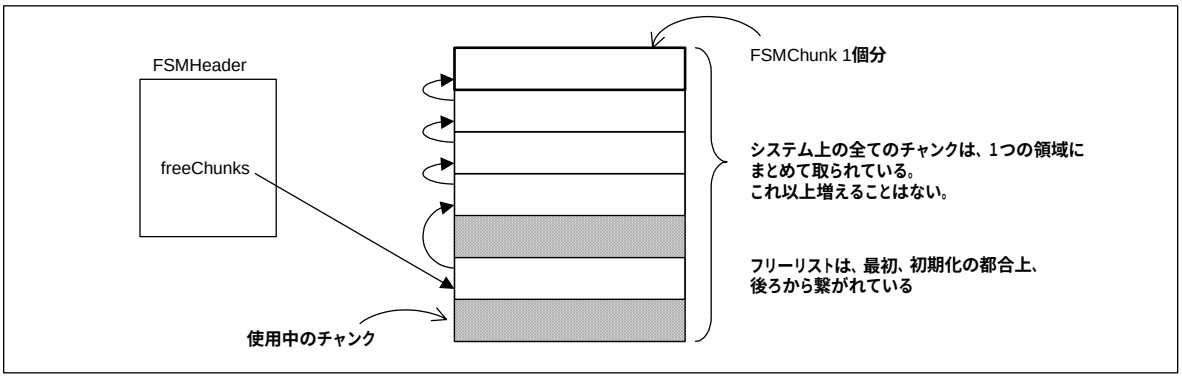


図 4.2

空いているチャンクは、FSMHeader の中の freeChunks からフリーリストとして繋がれている。

4.2. 全体の動作

リレーションのリストに関しては、新しいエントリを作るのが参照時であることが特徴的である。空き領域情報を追加する関数が呼ばれても、リスト中にそのリレーションのエントリが作られていなければ、その情報はそのまま捨てる。

管理するリレーション数は、コンフィギュレーションファイルの MaxFSMRelations で指定することができる。これを超えたら、優先度(useCount)を考慮した LRU で一番古いリレーションの情報を捨てる。はじめて登録されるリレーションは、FSM に useCount が 1 として登録される。もし useCount が 2 以上のエントリが、リレーションのリストの最後尾にあっても、それを捨てて、必ずリスト中に登録されることになっている。リストの最後尾にあるエントリを捨てることで、リストの最後でスラッシングを起こす可能性が出てくるが、こうしておかないと、一度リストが一杯になると新しいエントリを追加できなくなるからである。

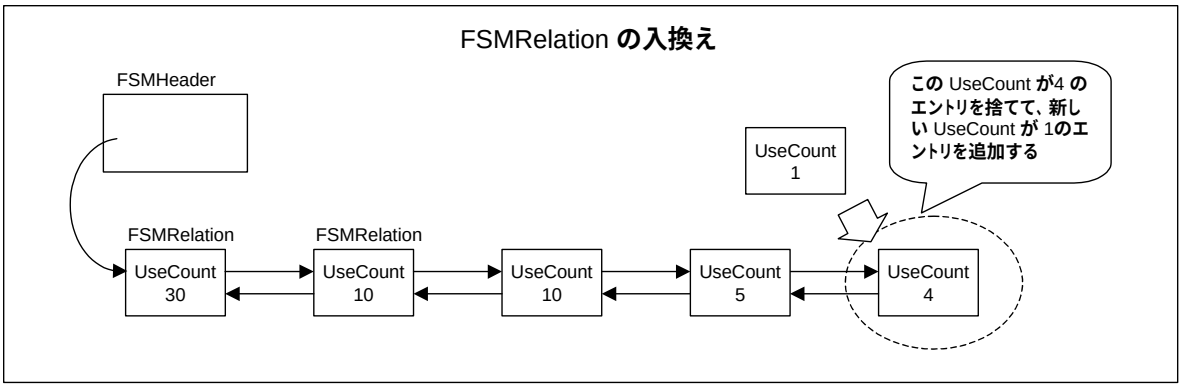


図 4.3

まだ空き領域の情報が１つもないリレーションの情報も、マップに入れることを許している。

監視している合計ページ数、つまり FSM のエントリ数も、コンフィギュレーションファイルの MaxFSMPages で設定できる。MaxFSMPages で指定された合計ページ数をリレーション間で動的に振り分けて利用する。よく利用されるリレーションと、タプルサイズが小さいリレーションほど優先度が高くなるようにできている。このしくみは、スレッシュホールドのメカニズムで実現されている。各リレーションは、ぞれぞれで、スレッシュホールドの値を管理している。そしてスレッシュホールド以上のサイズのページの情報のみがマップ中に保持される。新しく登録されたリレーションのスレッシュホールドは、BLCKSZ／2 から始まる。
GetFreeSpace() による参照のたびに、スレッシュホールドの値は要求されるサイズに向けて近づいていく。スレッシュホールドは、次のようにして要求サイズに近づけていく。

```
cur_avg = (int) fsmrel->threshold;
cur_avg += ((int) spaceNeeded - cur_avg) / 32;
fsmrel->threshold = (Size) cur_avg;
```

このような方法で、よく使われるリレーションは、スレッシュホールドの値が要求されるサイズの平均値(つまり平均のタプルサイズ)に近くなっていく。

FSM にページ情報を保存するスペースがなくなった場合、全てのリレーションのスレッシュホールドを２倍にする。ただし、最大でも BLCKSZ にすることで、スレッシュホールドが大きくなりすぎることは防止している。あまり使わないリレーションは、スレッシュホールドが大きくなっていくのである。

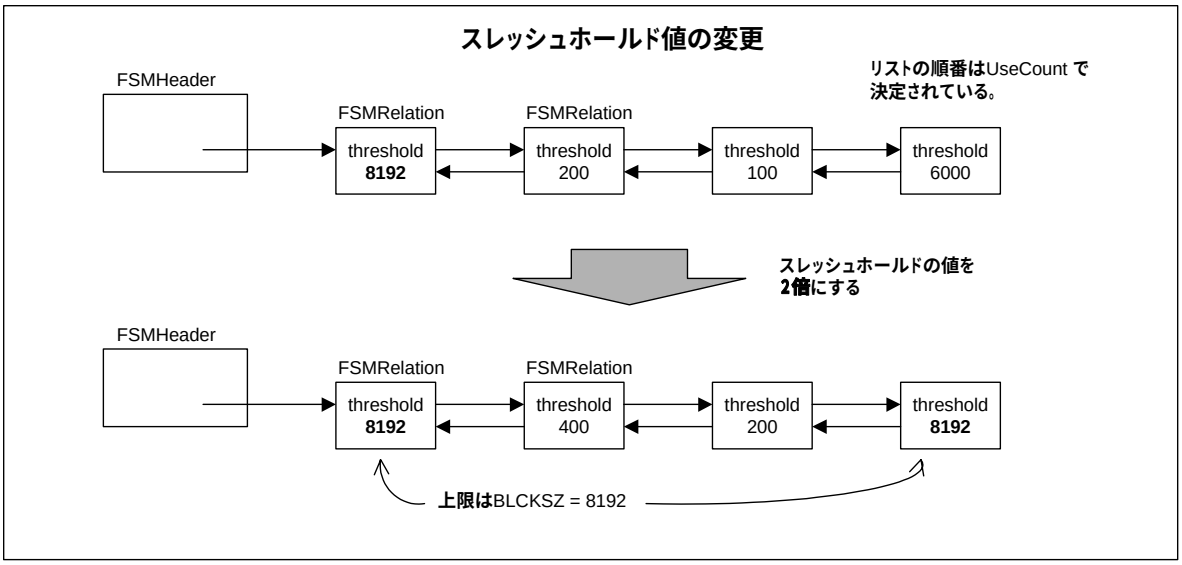


図 4.4

スレッシュホールドを２倍にする処理は、acquire_fsm_space() で実装されている。この中で、リレーションのリストをたどりながら、スレッシュホールドを２倍にして、新しいスレッシュホールド値でチャンクリストの回収を行う。この関数では、１回でどのくらいのチャンクが回収できるか保証されていないため、必要に応じて何回か呼び出す必要が出てくる場合がある。

4.3. 同時アクセスに対するページ割り当ての工夫

あるリレーションに対する空き領域の要求に対して、常に先頭から見つかったものを返していると、同じキュエリが同時実行されているような場合、複数のバックエンドに対して同じページを返すことになる。FSM は常に正しい空き容量を保証するものではないので、ページ内に要求したサイズがなかった場合、再度空き領域を要求する必要がある。同じページ要求が複数のバックエンドで行っているときは、ページの取り合いになり最悪である。

この問題に対処するために、空き領域を探し始める位置を、FSMRelation の nextPage で管理するようになっている。空き領域の検索は必ず nextPage からはじめる。必要な空き領域を持つページが見つかったら、nextPage を見つかったページエントリの次のエントリに設定する。チャンクの最後に来たら先頭に戻り、検索を開始した場所まで来たら、見つからなかったということになる。

4.4. 関数

storage/freespace/freespace.c に定義されている関数について、簡単に説明しておく。

FSM の外部インタフェースになる関数は、次のようなものである。

InitFreeSpaceMap()	フリースペースモジュールの初期化を行う
FreeSpaceShmemSize()	FSM のために必要な共有メモリのサイズの計算を行う
GetPageWithFreeSpace()	与えられたリレーションの中で、引数 spaceNeeded バイト分の空き領域のあるページを探す
RecordFreeSpace()	ページ内の利用可能な空き領域のサイズを登録する

RecordAndGetPageWithFreeSpace()	ロックを保持したまま、古いサイズの登録との要求サイズの取得を行う関数
MultiRecordFreeSpace()	複数ページ分の空き領域情報をまとめて登録する関数
FreeSpaceMapForgetRel()	特定リレーションの情報を全て削除する
FreeSpaceMapForgetDatabase()	特定データベースの情報を全て削除する

外部インタフェースを実現するための内部関数で、特徴的なものは次のようなものがある。

create_fsm_rel()	引数で指定されたリレーションが、FSM リレーションリストに登録されているかハッシュテーブルを使って検索する。もし見つからなければ、エントリを作って登録する。
find_free_space()	与えられた FSM リレーションのチャンクリストから、最低でも spaceNeeded バイト以上の空き領域のあるページを見つけて返す
lookup_fsm_rel()	ハッシュテーブルから該当リレーションの FSM リレーションのエントリを探す
lookup_fsm_page_entry()	FSM リレーションのチャンクリストから、指定したページのエントリを探してくる
fsm_record_free_space()	フリースペースを登録する
insert_fsm_page_entry()	FSM リレーションのチャンクリストに、新しいページのエントリを追加する
push_fsm_page_entry()	insert_fsm_page_entry() の補助ルーチン。エントリを chunk ÷ chunkRelIndex の位置に入れようとする
compact_fsm_page_list()	スレッシュホールド値以下のページ情報を捨てて、空き領域を回収する。
acquire_fsm_free_space()	リレーションをたどりながら、スレッシュホールド２倍にしていく。新しいスレッシュホールド値で各リレーションのチャンクリストに対して compact_fsm_page_list を実行する

参照時に、リレーションのエントリを追加するということを行っているため、インタフェースの GetPageWithFreeSpace() の中から create_fsm_rel() が呼び出される。create_fsm_rel() の中では、参照のあったリレーションに対して useCount をインクリメントして、同じ useCount の範囲で LRU の先頭に移動することで、変則的な LRU を実現している。

4.5. コードを読む練習

ここでは、free_chunk_chain(FSMChunk *fchunk) という、fchunk から始まるリストにつながれたチャンクを全て開放して、フリーリストにつなぐ関数の動作を説明する。

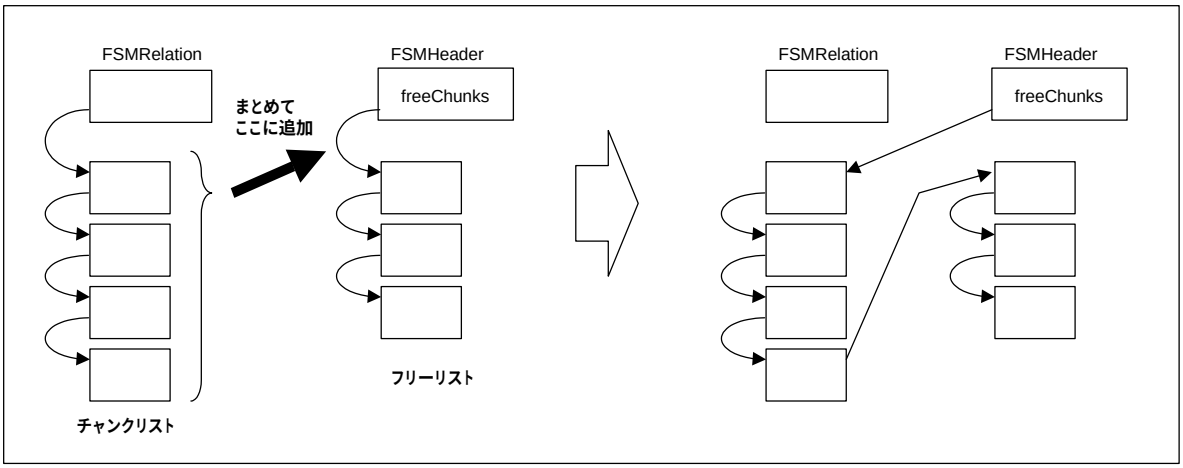


図 4.5

図のように FSMRelation につながれていたチャンクのリストが、FreeSpaceMapForgetRel() や LRU からあふれて、リレーションの情報ごと不要になった場合に利用される。これまで使っていたチャンクのリストを一気に開放する際に、図のようにまとめて freeChunks につないでしまうことで、高速化を測っている。

4.6. PostgreSQL 7.3.x までの FSM の問題点

PostgreSQL 7.3.4 までの FSM は、postmaster が起動されて以降の情報しか保持していない。従って、postmaster を再起動してしまうと、それまでの情報が失われてしまう。7.4 では、この情報を postmaster が再起動しても引き継ぐことができるようになっている。