PostgreSQL 解析資料

～ VACUUM の実装 ～

(株) NTT データ
基盤システム事業本部 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 における VACUUM の実装について説明する。
PostgreSQL は追記型のストレージを持つ DBMS である。従って、タプルの更新または削除の処理では、元のタプルに削除されたという印を付けるだけであり、不要になった元のタプルもデータベースの中に残っている。追記型のアーキテクチャのメリットは、トランザクション処理の実装が簡単になることや MVCC の処理が非常に軽くなることなどが挙げられる。しかし、一方で、不要となったタプルの使用している領域が増え続けるため、これらを回収する必要がある。この不要領域を回収して再利用可能にする処理が VACUUM である。

なお、本ドキュメントでは、VACUUM の使い方については説明しない。別途マニュアル等を参考にして頂きたい。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL 8.0.3 を対象にソースコードの調査を行ったものである。したがって、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. 対象ソースコード

本ドキュメントは、src/backend/commands ディレクトリにある vacuum.c と vacuumlazy.c を中心に、その他関連しているバッファマネージャ等のモジュールのソースコードについて解説している。

2. VACUUM の種類

ここでは、PostgreSQL で定義されている VACUUM の種類について説明する。VACUUM として意識しておくものは、CONCURRENT VACUUM, FULL VACUUM, VACUUM FREEZE, VACUUM ANLYZE の４種類である。

2.1. CONCURRENT VACUUM

CONCURRENT VACUUM とは、オプションなしで VACUUM を実行した場合の、一番よく使われる VACUUM 処理である。CONCURRENT VACUUM では、テーブルのページ単位での不要領域の回収と回収したタプルのインデックスからの削除を行う。ページ間にまたがってデータを移動させるよう

なリレーション全体での不要領域の回収は行わない。処理中は、テーブルの排他的ロックを取得しないため、通常のデータベース処理と並行して実行することができる。そのため、オンライン VACUUM と呼ばれることもある。

2.2. FULL VACUUM

FULLオプションを指定してVACUUMを実行した場合の処理である。基本は、ページ間でタプルを移動して前のほうに詰めることにより、不要領域の回収とリレーションファイルの縮小とを試みる。同時に、不要になったタプルをインデックスから削除する[1]。状況によっては、CONCURRENT VACUUM と同様のページ単位での不要領域の回収になる場合がある。

FULL VACUUM の処理中には、テーブルの排他的ロックが必要となり、システムの運用中での実行は難しい。

2.3. VACUUM FREEZE

FREEZE オプションを指定して VACUUM を実行した場合の処理である。CONCURRENT VACUUM と組み合わせて使用する。かつては、FULL VACUUM と組み合わせて使用できたようであるが、PostgreSQL 8.0.3 現在では、FULL VACUUM と組み合わせて使用しようとするとエラーになる。

FREEZE処理とは、XID[2]の周回問題を回避するため、古いXIDをFrozenTransactionIdという特殊なXIDに置き換える処理である。FREEZEオプション無しでVACUUMを実行した場合は、Xminが現在のXID (VACUUM処理のXID) より約 10 億以前のタプルに対してFREEZE処理を行う。FREEZEオプションを付けた場合は、現在実行されている最も古いトランザクションのXIDより古いXminを持つタプルの全てに対してFREEZE処理を行う。

2.4. VACUUM ANALYZE

VACUUM と同時に ANALYZE オプションを指定することである。基本的には、VACUUM の処理を実行した後、ANALYZE の処理を呼び出すだけである。本ドキュメントでは、ANALYZE は調査の対象外のため、VACUUM のオプションとして ANALYZE が指定された場合のエントリポイントの説明にとどめる。

2.5. VACUUM 対象のリレーションについて

VACUUM では、特定のリレーションを指定して VACUUM を実行することができる。リレーションを指定しない場合は、pg_class に登録されている全てのリレーションに対して VACUUM が実行される。

VACUUM の処理は、リレーション単位で実行されることになるが、そのときのエントリポイントとなるリレーションの種類は、通常のテーブルのみである。インデックスについては、テーブルの VACUUM を行う際に、VACUUM 対象のタプルが決まった時点で、インデックスからの削除が行われる。そして、TOAST も、その TOAST の元となるテーブルの VACUUM が終わった時点で、TOAST の VACUUM が実行される。

---

[1] 今回の調査では、btree インデックスについて、ページ間を移動させてページを詰める作業を行っているかどうかまでは判っていない。

[2] トランザクション ID のこと。PostgreSQL の中では、トランザクションを X で省略することがある。XID 以外の例では、トランザクションログが XLOG となるなどである。

インデックスや TOAST は、それらが関連付けられているテーブルからタプルが回収されない限り、VACUUM により回収できるエントリは存在しない。また、VIEW にはテーブルの実体がない。従って、リレーションを指定した VACUUM にインデックス、TOAST、VIEW などを指定しても、内部的にスキップされる。

3. VACUUM のソースコード

これまで説明した各種の VACUUM であるが、ソースコード中では、vacuum.c と vacuumlazy.c、そして analyze.c に含まれている。vacuum.c には、FULL VACUUM のソースコードと、各種 VACUUM で共通に使用されるソースコードが含まれている。そして、vacuumlazy.c には、CONCURRENT VACUUM のソースコードが、analyze.c には ANALYZE 処理のソースコードが含まれている。また、VACUUM FREEZE は、CONCURRENT VACUUM のオプションとして機能するため、専用のソースコードはない。これらのソースコードに含まれる関数は、リファレンスとして別紙にまとめたので、参照願いたい。

3.1. ソースコード上での VACUUM の処理の流れ

どのオプションが指定されているかに関わらず、構文解析を終わった時点でのクエリ木では、VACUUM は、VacuumStmt というステートメント構造体になっている。VACUUM の各種のオプションは、このステートメント構造体の中に保持されている。そして、VACUUM の処理は、tcop/utility.c の ProcessUtility() 内で処理が分岐され、VACUUM の処理が書かれている関数 vacuum()が呼び出されることで開始される。

vacuum()の引数には、ステートメント構造体 VacuumStmt が渡される。そして、さらにオプションにより、各種の VACUUM の処理に分岐される。

3.2. Vacuum ステートメント構造体

VacuumStmt 構造体は、次のように定義されている。このように、VACUUM コマンドで指定したオプションを引き渡せるようになっている。

```
typedef struct VacuumStmt
{
    NodeTag     type;
    bool        vacuum;       /* VACUUM コマンド時に true、ANALYZE のみの場合は false */
    bool        full;         /* VACUUM コマンドの FULL オプション時に true */
    bool        analyze;      /* VACUUM コマンドの ANALYZE オプション、
                                 または ANALYZE コマンド時に true */
    bool        freeze;       /* VACUUM コマンドの FREEZE オプション時に true */
    bool        verbose;      /* VACUUM または ANALYZE コマンドの VERBOSE オプション時に true */
    RangeVar   *relation;     /* テーブル指定時のリレーション情報.
                                 指定無し(全テーブルが対象)の場合は NULL */
    List       *va_cols;      /* VACUUM ANALYZE または ANALYZE コマンドのカラム指定時のカラム名
                                 リスト. 指定無し(全カラムが対象)の場合は NULL */
} VacuumStmt;
```

なお、VacuumStmt 構造体は ANALYZE コマンドと共用であり、ANALYZE コマンドのときも VacuumStmt 構造体が使用される。従って、vacuum() 関数は ANALYZE コマンドの処理も含んでい

る。

3.3. vacuum()での処理概要

vacuum() では、コマンドのオプションのチェックやメモリコンテキストの初期化などを行った後、メインループの中で、リレーション単位で VACUUM および ANALYZE 処理を実行する。対象リレーションの VACUUM 処理が全て終了すると、全リレーション対象の VACUUM の場合に限り、データベース単位の統計情報の更新と不要な CLOG セグメントの削除を行う。

ソースコードで見ていくと、主なシーケンスは次のようになっている。

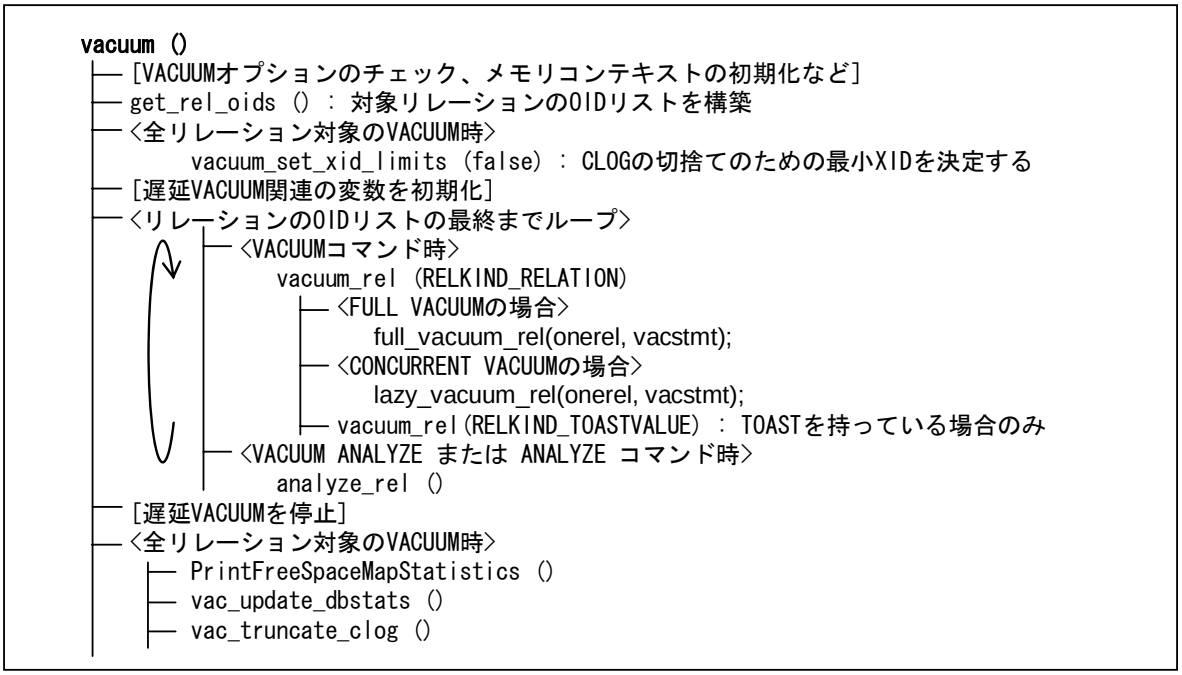

図 3-1 VACUUM 処理の分岐

1. get_rel_oids() : VACUUM 対象となるリレーションの OID リストを構築する。VACUUM のオプションとして、リレーションが指定されている場合は、それらのリレーションの OID リストを構築する。リレーションが指定されていない場合は、全リレーション対象の VACUUM となるため、pg_class から通常のテーブルだけを取得して、OID リストを構築する。全リレーション対象の場合、VIEW, INDEX, TOAST などは、このリストに含まれない。リレーションが指定されていた場合は、VIEW, INDEX, TOAST などの OID も変換し、後の処理で WARNING を出し、処理を省略する。
2. vacuum_set_xid_limits() : 全リレーション対象の VACUUM の場合、後のデータベースの統計情報を更新と CLOG の切り詰めのため、同一データベースで実行されているトランザクション中で一番古いトランザクションの XID の取得と、FREEZE のカットオフポイントの取得を行う。FREEZE オプションが指定されていない場合は、FREEZE のカットオフポイントは現在の XID より約 10 億古い XID にする。FREEZE オプションが指定されている場合は、FREEZE のカットオフポイントは、同一データベースで実行している全トランザクション中で一番古いトランザクションの XID と同じ値に設定される。

3. postgresql.conf ファイルで、遅延 VACUUM のパラメータが設定してある場合、ここで遅延 VACUUM になるように値を設定する。
4. リレーションのOIDリストに対するループ： 1のget_rel_oids()で取得したOIDリストから順にリレーション単位のVACUUMおよびANALYZEの処理を実行する。
   i. vacuum_rel() ：リレーション単位での VACUUM 処理。この関数の中で、FULL VACUUM と CONCURRENT VACUUM の処理が分岐する。
      TOAST を持っている場合は、さらにここから TOAST に対する VACUUM 処理を呼び出す。
   ii. analyze_rel() ：リレーション単位での ANALYZE 処理
5. PrintFreeSpaceMapStatistics() : 全リレーション対象の VACUUM の場合、フリースペースマップ情報を DEBUG2 レベルでログに出力する。なお、VERBOSE オプションを指定した場合には INFO レベルで出力されるので、フロントエンド側で情報の確認が可能となっている。
6. vac_update_dbstats() : 全リレーション対象の VACUUM の場合、処理中のデータベースの統計情報を更新する。pg_database カタログの datvacuumxid と datfrozenxid を更新する。
7. vac_truncate_clog() : 全リレーション対象の VACUUM の場合、接続不可の DB を除いたシステム中で最も古い pg_database カタログの datvacuumxid より古い CLOG の削除を試みる。また XID の周回に関するチェックを行い、必要に応じて警告を出している。実際の CLOG の削除は access/transam/clog.c の TruncateCLOG() で行われる。

上記のシーケンスから分かるように、リレーションを指定していない全リレーション対象の VACUUM 処理においても、リレーション単位の VACUUM 処理の繰り返しである。

3.4. 遅延 VACUUM について

遅延 VACUUM とは、VACUUM 処理中における通常処理の性能低下を抑えるため、VACUUM 処理を定期的 sleep させて遅延させることで、VACUUM の処理の負荷を下げる機能である。postgresql.conf のパラメータ vacuum_cost_delay に 0 以外の値が設定ている場合に、遅延 VACUUM を行う。
VACUUMの遅延処理は、コストベースで行われる。コストカウンタへの見積もりコストの累積は、storage/buffer/bufmgr.cのReadBufferInternal()とwrite_buffer()で行われる。ReadBufferInternal()では、バッファキャッシュにヒットすれば、vacuum_cost_page_hit[3] の値を、ヒットしなかった場合はvacuum_cost_page_miss[4] の値をコストカウンタに加算する。write_buffer()では、バッファがダーティになるときにvacuum_cost_page_dirty[5] の値をコストカウンタに加算する。そして、コストの上限値vacuum_cost_limit[6]を超えた場合にプロセスを一定時間スリープする。スリープから覚めるとコストカウンタを 0 にリセットする。

コストの上限値のチェックは、VACUUM 処理の各々の主要なループ内（通常はページ毎の処理）から vacuum_delay_point() を呼び出すことによって行われる。vacuum_delay_point()が呼び出されると、コストカウンタの累積値が vacuum_cost_limit を超えていないかチェックする。vacuum_cost_limit を超えていた場合は、遅延時間を計算して、遅延時間分スリープする。遅延時間の計算方法は以下の

[3] postgresql.conf のパラメータ。バッファキャッシュにヒットした場合のコスト。デフォルト値は 1。
[4] postgresql.conf のパラメータ。バッファキャッシュにヒットしなかった場合のコスト。デフォルト値は 10。
[5] postgresql.conf のパラメータ。バッファがダーティになるときに加算するコスト。デフォルト値は 20。
[6] postgresql.conf のパラメータ。デフォルト値は 200。

とおりである。

```
遅延時間(msec) = vacuum_cost_delay * <コストの累積値> / vacuum_cost_limit
        (ただし、遅延時間は最大でも vacuum_cost_delay * 4 まで)
```

3.5. トランザクションブロックの制御

VACUUM の処理では、各リレーション単位の VACUUM が、１つのトランザクションとなる。ソースコード中では、リレーション単位での VACUUM 処理である vacuum_rel() の始めの方でトランザクションが開始され、リレーション１つ分の VACUUM 処理が終わった時点でコミットされる。

4. VACUUM の内部処理

ここでは、VACUUM の内部で使われる、個別の技術について説明する。

4.1. VACUUM 処理の流れ

はじめに、VACUUM 処理の流れについて説明する。
CONCURRENT VACUUM、FULL VACUUM ともに、大きく次のような流れになっている。

1. テーブルをスキャンして、VACUUM対象タプルのエントリリスト[7]を構築する。また、同時に、FREEZEの必要があるタプルに対しては、FREEZE処理を行う。
2. index から VACUUM 対象のタプルを削除する。
3. テーブルから VACUUM 対象のタプルを削除する。
4. TOAST を持っていたら、その TOAST の VACUUM を実行する。

このように､テーブルに対しては 2 フェーズで処理が実施される｡CONCURRENT VACUUM と FULL VACUUM の大きな違いは、３のテーブルから VACUUM 対象のタプルを削除するところで、ページ単位で VACUUM を行うか､テーブル全体でタプルを移動しながら空き領域を詰めるかの違いである。FULL VACUUM では、1～4 を 1 ステップずつ、リレーション単位で VACUUM 処理を行っていく。CONCURRENT VACUUM では、若干トリッキーなことを行っており、１のテーブルのスキャンを行っていく中で、タプルのエントリリストの蓄積用の領域が一杯になったら、その時点で２と３の処理を行うので、ソースコードを読む際には注意が必要である。

4.2. VACUUM 対象となるタプル状態の検査

VACUUM で回収できるタプルであるかどうかは、utils/time/tqual.c の HeapTupleSatisfiesVacuum() によって判断される。HeapTupleSatisfiesVacuum()の戻り値として取得できるタプル状態は、以下の通りである。

```
typedef enum
{
    HEAPTUPLE_DEAD,                 /* タプルは死んでいて、削除可能 */
    HEAPTUPLE_LIVE,                 /* タプルは生きている */
    HEAPTUPLE_RECENTLY_DEAD,        /* タプルは死んでいるが, まだ削除可能でない */
```

[7] CONCURRENT VACUUM 時と FULL VACUUM 時では、異なる構造体になっている。

```
    HEAPTUPLE_INSERT_IN_PROGRESS,       /* INSERT しているトランザクションがまだ進行中 */
    HEAPTUPLE_DELETE_IN_PROGRESS        /* DELETE しているトランザクションがまだ進行中 */
} HTSV_Result;
```

タプルの状態は、HeapTupleHeaderの情報とトランザクションのコミット状況、OldestXmin[8]との前後関係で決定される。

タプルの状態が HEAPTUPLE_DEAD の場合は、VACUUM 対象となるタプルであるため、VACUUM 対象タプルのエントリリストにそのタプルの位置を登録する。

タプルの状態がHEAPTUPLE_LIVEの場合は、VACUUM対象のタプルには該当しないので、エントリリストへの登録は行わない。HEAPTUPLE_LIVEの場合は、FREEZE処理の対象となる可能性がある。そのタプルのt_xminがFreezeLimit[9]より古ければ、t_xminをFrozenTransactionId (XID=2) に変更してタプルをFREEZE する。FreezeLimitは、前述のようにVACUUM FREEZEでない場合は現在のXIDより約 10 億古いものとなり、VACUUM FREEZEの場合はOldestXminとなる。

タプルの状態が HEAPTUPLE_RECENTLY_DEAD の場合は、事前に vacuum_set_xid_limits()によって得られている OldestXmin よりも新しい t_xmax を持つタプルが対象となっており、そのタプルは「ごく最近死んだ」状態のものと考えられ、オープンしている幾つかのトランザクションでまだ見える状態であるかもしれない。たとえ VACUUM 処理でその削除トランザクションがコミットされていることが確認できたとしても、この状態のタプルは VACUUM 対象に入れてはならない。

HEAPTUPLE_INSERT_IN_PROGRESS と HEAPTUPLE_INSERT_IN_PROGRESS の状態のタプルは VACUUM の対象とせず、テーブルのスキャンを続行する。この状態は、リレーションに ACCESS EXCLUSIVE ロックを掛けている FULL VACUUM では発生すべきでない。しかし、システムカタログでは、コミットの前に書き込みロックを解放するので、この状態が発生することがある。FULL VACUUM でこのタプル状態を見つけた場合には NOTICE メッセージを出力し、そのリレーションは、後の処理ではタプルの移動による不要領域の回収ではなく、CONCURRENT VACUUM 同様のページ単位の VACUUM 処理を行う。

4.3. StrategyHintVacuum()

バッファキャッシュの交換ストラテジに対して、VACUUMがアクティブである事を伝える。通常時のフリーリストへのバッファの登録では、リストのMRU[10]側に最後に使ったバッファは配置される。しかし、VACUUMがアクティブの場合は、シーケンシャルスキャンが主であるので、フリーリストのLRU[11]側に配置してバッファ管理の最適化を行っている。

4.4. ロック

ここでは、CONCURRENT VACUUM と FULL VACUUM でのロックの違いについて説明する。

[8] VACUUM 開始時に実行中の全トランザクション中での一番古い t_xmin。
[9] FREEZE のカットオフポイントを示すのに使われる変数。
[10] most recently used：最近に使用されたもの
[11] least recently used：最も以前に使用されたもの

4.4.1. CONCURRENT VACUUM のロック

CONCURRENT VACUUM では、基本的にリレーションに対するロックは ShareUpdateExclusiveLock、バッファに対するロックは BUFFER_LOCK_SHARE を取得する。しかし、ページ内の VACUUM を実行する際のバッファロックでは、排他ロックを取得後に自分以外のバッファの PIN が無くなるまで待つという、特殊な排他ロック(LockBufferForCleanup())が行われている。また、インデックス VACUUM 時のインデックスのロックは、インデックス VACUUM のアクセスメソッドが同時更新をサポートしていれば RowExclusiveLock で、サポートされていなければ AccessExclusiveLock となっている。他に lazy_vacuum_rel()の最後に行われる、末尾の空ページの切詰めの際にも、AccessExclusiveLock でのリレーションロックの取得を試みてる。この場合、ロックが取得できなければ切詰めを諦める。

4.4.2. FULL VACUUM のロック

FULL VACUUM では、基本的にリレーションのロックは AccessExclusiveLock で、バッファのロックは BUFFER_LOCK_EXCLUSIVE が使用されている。

4.5. ページ単位の VACUUM -- lazy_vacuum_page() / vacuum_page ()

ここでは、1 ページ単位での VACUUM 処理について説明する。

VACUUM対象タプルのエントリリストは、事前のヒープのスキャン処理にて、CONCURRENT VACUUM時にはLVRelStats構造体のdead_tuplesに、FULL VACUUM時にはVacPageData構造体のoffsetsに構築される。各構造体の詳細については、5.2. 情報収集とページ内VACUUMのためのテーブルスキャン -- lazy_scan_heap() および、6.2. 情報収集のためのヒープスキャン-- scan_heap() を参照してほしい。

lazy_vacuum_page() / vacuum_page()では、まず最初に該当ページのVACUUM対象タプルのエントリリストより順にタプル情報を取得して、そのItemIdDataのlp_flagsに未使用(~LP_USED [12])と印を付けていく（図 4-1の ①)。

ページ内の印付けが全て終了したら、次にstorage/page/bufpage.c のPageRepairFragmentation() を呼び出して、ページ上のフラグメンテーションを除去する。PageRepairFragmentation()ではページ内の全てのItemIdDataをスキャンして、lp_flagsが未使用あるいは削除で印付けされているItemIdDataが見つかった場合に、そのlp_flagsを未使用と印付けした後にデータ領域を解放して、生きているデータはページ領域の後ろから詰めている（図 4-1の ②)。

[12] LP_USED は、そのエントリが使用中であることを示すビットフラグ。~は NOT を表すので、~ LP_USED は LP_USED ビットをクリアすること。つまり、そのエントリが未使用という状態を示すようにしている。

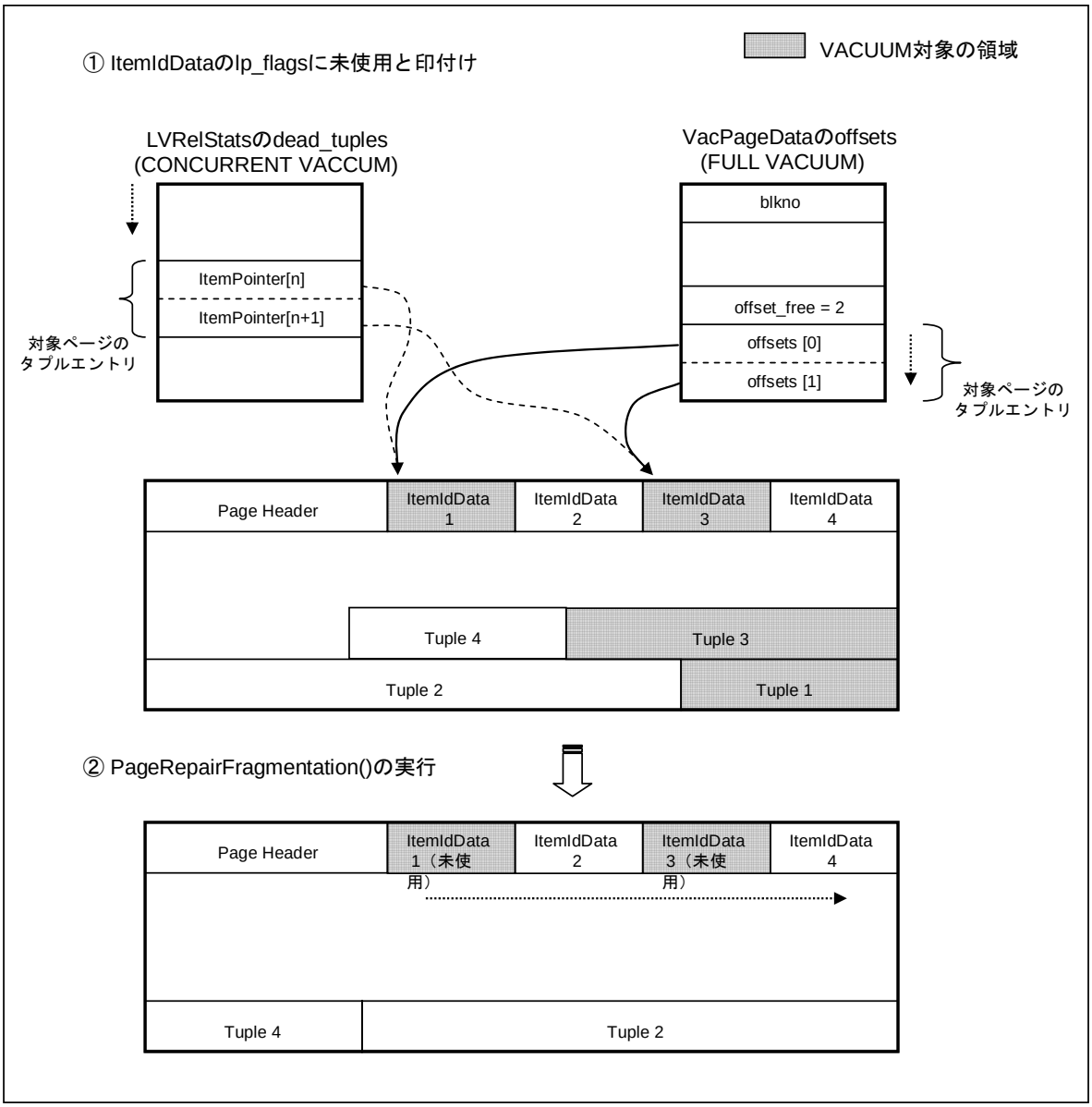

図 4-1 ページ内の VACUUM 処理

最後にaccess/heap/heapam.cのlog_heap_clean()[13] を呼び出して、XLOGに未使用アイテムを除去したことを記録する。そしてページヘッダにLSNとWALのTLIをセットする。

4.6. インデックスの VACUUM -- lazy_vacuum_index() / lazy_scan_index / vacuum_index () / scan_index ()

インデックスの VACUUM 処理は、VACUUM 対象ページが見つかったかどうかによって、処理が分かれる。VACUUM 対象のページが見つかった場合は、インデックスも VACUUM を行う必要があるため、lazy_vacuum_index() または vacuum_index() を呼び出して VACUUM 対象タプルに関連したインデックスをまとめて削除している。VACUUM 対象ページが見つからなかった場合、インデックスの VACUUM を行う必要はないが、統計情報の収集と共有フリースペースの更新を行うために lazy_scan_index() または scan_index()を呼び出している。

vacuum_index() と scan_index() の違いは、実際にインデックスエントリの削除を行うかどうかであ

[13] 最終的には XLogInsert(XLOG_HEAP_CLEAN) を呼び出している。

り、他はほとんど同じ処理となっている。また、先頭に lazy_ のついた CONCURRENT VACUUM 用の関数は、ロックの取り方などが若干異なるだけで、それぞれほとんど同じ処理を行っている。
ここでは、vacuum_index() を対象として説明を進める。

vacuum_index() では、最初にindex_bulk_delete()を呼び出して、VACUUM 対象のタプルを指しているインデックスエントリを探し出してバルク削除[14]を行う。実際のバルク削除処理は、インデックスの種類に従ったバルク削除用アクセスメソッド(pg_amカタログのambulkdelete) が実行される。インデックスの種類がbtreeの場合[15]では、btbulkdelete() がバルク削除用のアクセスメソッドとして指定されている。btbulkdelete()では、インデックスの全件スキャン[16]が走り、インデックスエントリの削除対象はindex_bulk_delete()の引数で指定したコールバックルーチンを介して判定される[17]。バルク削除の最後には、インデックスページ、タプルに関する統計が計算され、その情報は IndexBulkDeleteResult構造体で返却される。
インデックスの種類によっては、インデックスエントリの削除時にログを生成しない場合もある[18]。

バルク削除が終了すると、次に VACUUM 後のクリーンアップ処理 index_vacuum_cleanup() を呼び出して、インデックスの VACUUM の後処理を行う。バルク削除と同様の手法で、実際は VACUUM 後のクリーンアップ用アクセスメソッド(pg_am の amvacuumcleanup)が実行されることになり、インデックスの種類が btree の場合は btvacuumcleanup() がクリーンアップ用のアクセスメソッドに指定されている。
最後に共有フリースペースマップに対してインデックスのフリースペース情報の更新を行っている。

クリーンアップ処理後は、インデックスの統計情報（pg_class カタログ）の更新を行っている。

4.7. リレーション単位の VACUUM 処理 -- vacuum_rel()

ここではリレーション単位で実行される VACUUM 処理 vacuum_rel() の概要について説明する。

vacuum_rel() では、リレーション単位でトランザクションの開始、ロックの確保を行った後、FULL VACUUM 用あるいは CONCURRENT VACUUM 用の VACUUM 処理を呼び出す。もし、リレーションに派生する TOAST リレーションを持つ場合は、vacuum_rel() 自身の再起呼び出しを行って TOAST リレーションの VACUUM も行っている。

リレーション単位の VACUUM 処理の主なシーケンスは次のようになっている。

1. StartTransactionCommand()： トランザクションの開始。
2. StrategyHintVacuum(true)： バッファマネージャに対して、VACUUM がアクティブである事を伝える。

[14] 削除するタプルの情報をまとめて渡し、一気に不要なタプルのエントリを削除する方法。
[15] 今回は、btree インデックスの VACUUM 方法について詳しく調べてないため、関数のエントリポイントまでの紹介にとどめる。
[16] btbulkdelete()の全件スキャン中のバッファロックでは、排他ロックを取得後、自分以外のバッファの PIN が無くなるまで待つという、特殊な排他ロック（LockBufferForCleanup()）が行われている。
[17] scan_index() ではコールバックルーチンが常に false を返却するので、実際にインデックスエントリを削除することはない。ここが vacuum_index() との大きな違いである。
[18] このようなインデックスは、データベースの異常終了時にリカバリされない。従って、インデックスの再構築が必要になる。

3. ロックモードを決定する。FULL VACUUM の場合は AccessExclusiveLock で、CONCURRENT VACUUM の場合は ShareUpdateExclusiveLock となっている。
4. relation_open()： 3. で求めたロックモードでリレーションをオープンする。
5. LockRelationForSession()： 3. で求めたロックモードで、リレーションのセッションレベルのロックも確保する。これは 10. の TOAST リレーションの VACUUM 処理のために、親リレーションをロックしてある。
6. full_vacuum_rel() / lazy_vacuum_rel()： 実際の FULL / CONNCURRENT VACUUM 処理の実行。
7. relation_close(NoLock)： リレーションのクローズ。しかしコミットするまでロックは保持している。
8. StrategyHintVacuum(false)： バッファマネージャに VACUUM が非アクティブである事を伝える。
9. CommitTransactionCommand()： トランザクションのコミット。
10. 対象リレーションに派生した TOAST リレーションを持つならば、自身を再起呼び出しして TOAST リレーションを VACUUM する。
11. UnlockRelationForSession()：マスターリレーションのセッションレベルのロックを解除する。

5. CONCURRENT VACUUM 処理部の概要

この章では、リレーション単位での CONCURRENT VACUUM 処理について説明を行う。
CONCURRENT VACUUM では、前述のページ単位での VACUUM、インデックスの VACUUM、TOAST の VACUUM を行う。

CONCURRENT VACUUM での基本的な処理の流れは、VACUUM 対象テーブルのスキャン、index の VACUUM、ページ単位での VACUUM である。しかし、実際の処理は、一定量のスキャンを行ったところで、index の VACUUM、ページ単位での VACUUM を行い、また、同じサイクルを繰り返す。VACUUM 処理が実行されるタイミングは、スキャン中に収集している VACUUM 対象タプルのエントリ情報が、最大 maintenance_work_mem サイズ分のメモリ領域に格納できないと判断された時点である。スキャン中にはページのフリースペース情報も収集されるが、最大 max_fsm_pages 分までのフリースペース情報を収集するようになっており、運用中のリソースを考慮した作りとなっている。
対象リレーションのスキャンを終えると、最後に共有フリースペースマップと pg_class カタログの統計情報を更新する。

5.1. リレーション単位の CONCURRENT VACUUM 処理 -- lazy_vacuum_rel ()

ここではリレーション単位で実行される CONCURRENT VACUUM 処理 lazy_vacuum_rel () の概要について説明する。

ソースコードを見ていくと、主なシーケンスは次のようになっている。lazy_vacuum_rel()は、リレーション単位の VACUUM 処理である vacuum_rel() から呼び出される。

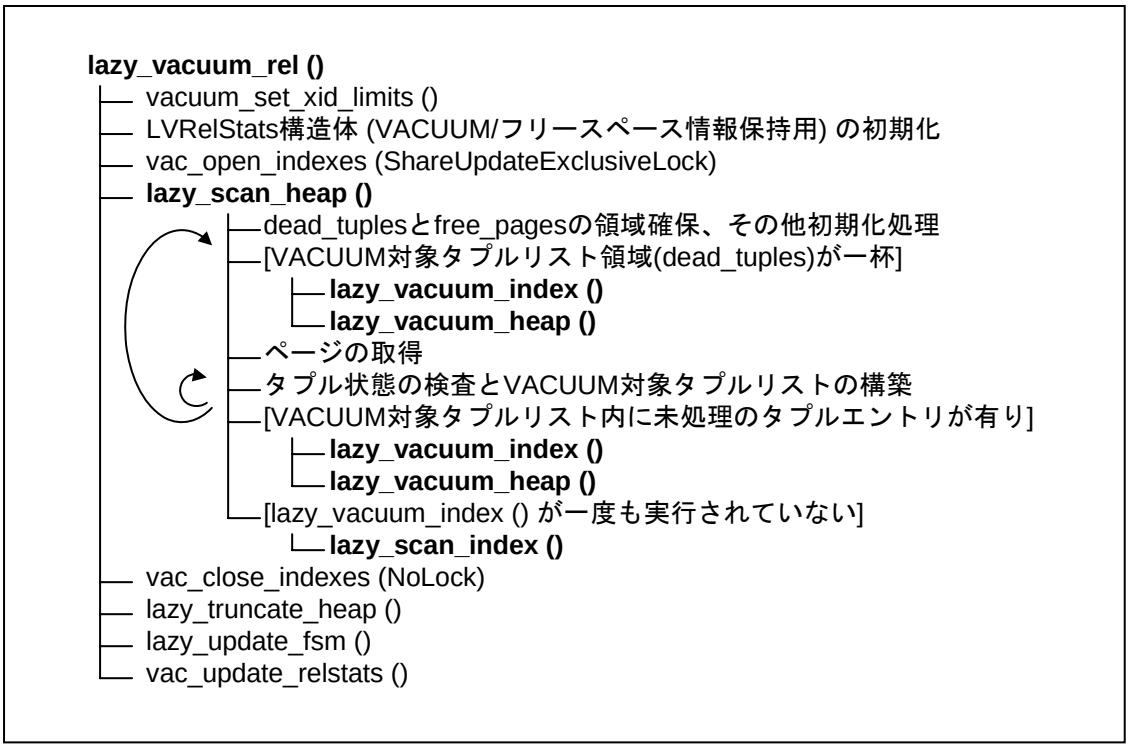

図 5-1 CONCURRENT VACUUM の処理の流れ

1. vacuum_set_xid_limits(): 実行中の全トランザクション中での一番古い xmin の取得と、FREEZE する XID のカットオフポイントの計算を行う。VACUUM の対象となるリレーションが共有リレーションの場合は、全バックエンドのトランザクションが対象となり、そうでない場合は、同一データベース上の全バックエンドのトランザクションが対象となる。
2. vac_open_indexes():リレーションが所有している全インデックスを ShareUpdateExclusiveLock でオープンする。
3. lazy_scan_heap(): テーブルをスキャンして、ページ単位のVACUUM、インデックスのVACUUM などを行う。また、スキャン中にページのフリースペース情報[19]も構築していく。詳細は5.2節で述べる。
4. lazy_truncate_heap(): テーブルの末尾に空のページが存在していれば[20]、空のページ内をスキャンして、VACUUM中に他のバックエンドが空の終了ページに対してタプルを作成していないことを確認した上で、RelationTruncate()でファイルの切詰めを試みる。この際、ブロッキングなしで AccessExclusiveLock ロックの取得を試み、ロックが取得できた場合のみファイルの切詰めを行う。ロックを取得できなければ切詰めは諦める。
5. lazy_update_fsm (): 現在持っているフリースペース情報で、共有フリースペースマップを更新する。storage/freespace/freespace.c の RecordRelationFreeSpace()で共有フリースペースマップに記録している。
6. vac_update_relstats(): リレーションのための統計情報の更新を行う。pg_class カタログの relpages (ページ数の推測値)、reltuples (タプル数の推測値)、relhasindex (インデックス所有フラ

[19] FSM に登録するための情報を収集している。
[20] ある程度の空の終了ページが連続していないと、切詰めの対象とならない。この判定を行うためのトレードオフパラメータ REL_TRUNCATE_MINIMUM と REL_TRUNCATE_FRACTION は、今のところ vacuumlazy.c にてハードコーディングされているが、コメントによると GUC 化も検討されているらしい。

グ)、relhaspkey (主キー所有フラグ)を更新している。

5.2. 情報収集とページ内 VACUUM のためのテーブルスキャン -- lazy_scan_heap()

lazy_scan_heap() では、対象のテーブルをスキャンしながら全タプルを検査し、VACUUM 対象の ItemPointerData を最大 maintenance_work_mem サイズ分のメモリ領域に格納していく。スキャン中にこのメモリ領域が足りなくなると、そのタイミングで対象タプルに関連したインデックスの VACUUM と、ページ単位の VACUUM 処理が行われる。そして VACUM 対象タプルの収集用の領域をクリアしてスキャンを再開する。
またスキャン中には、GetAvgFSMRequestSizez()で取得したリレーションの平均 FSM リクエストサイズ(閾値)よりも多くのフリースペースを持つページのフリースペース情報を構築していく。この格納領域は max_fsm_pages 分のメモリ領域に格納されるが、このメモリ領域が足りなくなると、格納済みの情報よりも多くのフリースペースサイズを持つページ情報と入れ替えている。lazy_scan_heap()で構築されたフリースペース情報は、最終的に lazy_vacuum_rel()で呼び出される lazy_update_fsm ()により、共有フリースペースマップに反映される。

lazy_scan_heap()で構築される VACUUM 対象のタプル情報、フリースペース情報は、以下の LVRelStats 構造体で定義されている。

```
typedef struct LVRelStats
{
    /* リレーション全体についての統計 */
    BlockNumber    rel_pages;                  /* ページ数 */
    double              rel_tuples;            /* 有効タプル数 */
    BlockNumber    pages_removed;              /* 削除ページ数 */
    double              tuples_deleted;        /* 削除タプル数 */
    BlockNumber    nonempty_pages;             /* 有効タプルが存在する最終ページ + 1 */
    Size                    threshold;         /* 平均FSMリクエストサイズ(閾値) */

    /* 削除予定のタプルのTIDsリスト */
    /* このリストはTIDアドレスによって順序付けされている */
    int                      num_dead_tuples;  /* 現在の削除タプル数 # */
    int                      max_dead_tuples;  /* 配列に割り当てられたタプル数 */
    ItemPointer       dead_tuples;             /* ItemPointerDataの配列 */

    /* ページ当たりのフリースペースについての情報の配列、またはヒープ */
    /* 一杯になる時までシンプルな配列を使い、それからヒープに変更する */
    bool                    fs_is_heap;        /* ヒープの構成を使っているか? */
    int                      num_free_pages;   /* フリースペースを持つ現在のページ数 # */
    int                      max_free_pages;   /* 配列に割り当てられたページ数 */
    PageFreeSpaceInfo  *free_pages;            /* blknoとavailの配列またはヒープ */
} LVRelStats;
```

タプルの VACUUM 手順と、フリースペース情報の構築手順は以下のようになっている。

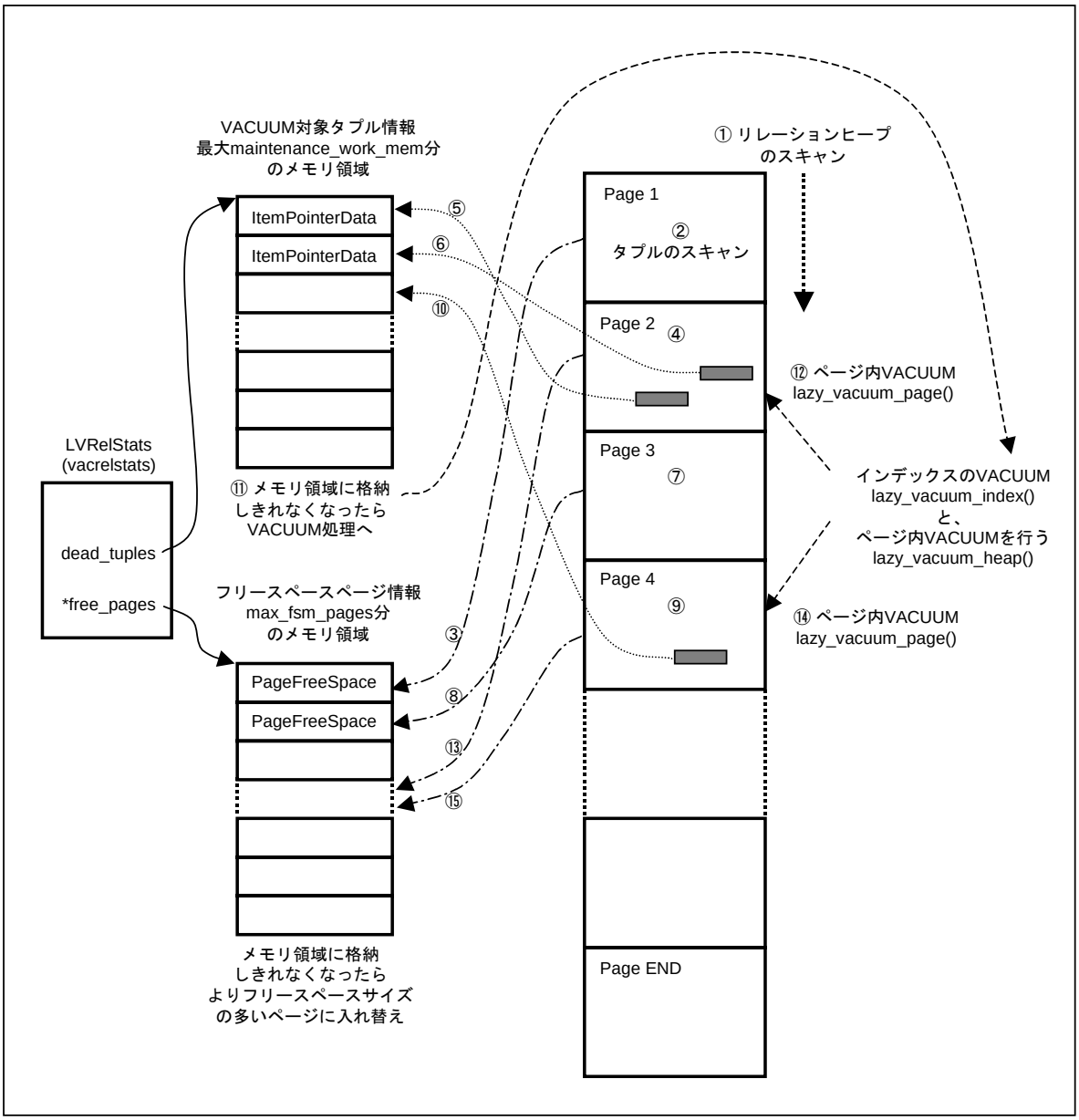

図 5-2 タプルの VACCUM とフリースペース情報の構築

1. lazy_space_alloc()を呼び出し、LVRelStats 構造体の VACUUM 対象タプルのエントリリスト dead_tuples とフリースペース情報 free_pages のメモリ領域を確保する。dead_tuples には最大 maintenance_work_mem 分の領域を、free_pages には max_fsm_pages のページ数分のフリースペース情報の領域をメモリに割り当てる。ただし、free_pages にはテーブルが持つブロック数より多くのページ情報の領域を割り当てる必要はない。
2. テーブルの前方のページから順にスキャンを開始する。(図 5-2 ①)
3. ページ内のタプルの状態を検査する。タプルの状態がHEAPTUPLE_LIVEの場合は、そのタプルがFREEZE対象なら、FREEZE処理を行う。もしページ内にHEAPTUPLE_DEADの状態のタプルが存在していなければ(図 5-2 ②⑦)、フリースペース情報free_pagesにそのページのブロックNoとフリースペースサイズの情報を格納する。(図 5-2 ③⑧)ただし、リレーションの平均FSMリクエストサイズ(閾値)よりも多くのフリースペースを持たない場合は、格納の対象とならない。テーブルのスキャンが進み、ページのフリースペース情報のfree_pagesの領域が足りなくなると、格

納済みの情報よりも多くのフリースペースサイズを持つページ情報の場合のみ入れ替えを行っている[21]。

4. ページ内のタプルの状態を検査して、ページ内にHEAPTUPLE_DEADの状態のタプルが存在していれば(図 5-2 ④⑨)、dead_tuplesにそのタプルのItemPointerDataを格納する（図 5-2 ⑤⑥⑩）。テーブルのスキャンが進み、dead_tuplesの領域が足りなくなると、テーブルのスキャンを一旦休止してdead_tuplesのエントリ情報を元にインデックスのVACUUM (lazy_vacuum_index())と、ページ単位でページ内VACUUM処理（ lazy_vacuum_heap() / lazy_vacuum_page() 図 5-2 ⑪⑫⑭) を行う。そして、そのページのVACUUM後のフリースペース情報をfree_pagesへ格納を試みる。(図 5-2 ⑬⑮)
5. dead_tuples 内のすべての対象タプルの VACUUM が終了すると、dead_tuples の情報をクリア後、一旦休止していたテーブルのスキャンを再開する。テーブルの最後まで 2. ～ 5. の処理を繰り返す。
6. テーブルの最後までスキャンが終了したら、dead_tuples に残っている未処理分の対象タプルの VACUUM 処理を実行する。

5.3. インデックスの VACUUM -- lazy_vacuum_index () / lazy_scan_index ()

4.6節で説明したインデックスのVACUUM処理である。CONCURRENT VACUUM時のインデックスのロックは、インデックスVACUUMのアクセスメソッドが同時更新をサポートしていればRowExclusiveLockで、サポートされていなければAccessExclusiveLockとなっている。

5.4. 収集された VACUUM 対象タプルの VACUUM -- lazy_vacuum_heap()

テーブルのスキャンで収集した maintenance_work_mem 領域分の VACUUM 対象タプルのエントリリストを元に、lazy_vacuum_page()を呼び出してページ単位で VACUUM 処理を行う。そして、lazy_record_free_space()で LVRelStats 構造体の free_pages にフリースペースページ情報を構築していく。

5.5. ページ内の VACUUM -- lazy_vacuum_page ()

4.5節で説明したページ内のVACUUM処理のとおりである。CONCURRENT VACUUM時のページ内のVACUUM処理中のバッファロックは、排他ロックを取得後、自分以外のバッファのPINが無くなるまで待つという、特殊な排他ロック(LockBufferForCleanup()) で行われている。

6. FULL VACUUM 処理部の概要

この章では、リレーション単位での FULL VACUUM 処理について説明を行う。

6.1. リレーション単位の FULL VACUUM 処理 -- full_vacuum_rel()

ここではリレーション単位で実行される FULL VACUUM 処理 full_vacuum_rel() の概要について説明する。

[21] フリースペース情報の格納は lazy_record_free_space()を呼び出すことによって行われ、バッファ入れ替えのアルゴリズムは、クヌースのヒープベースの優先順位付き待ちクエリ（heap-based priority queues sec 5.2.3）を用いている。

full_vacuum_rel() では、まず最初に対象リレーションのヒープをスキャンして、FULL VACUUM に関するのページとタプル情報を収集する。スキャンの結果、タプルの移動が可能なフリースペースを持つページが存在していなければ、収集した VACUUM 対象タプルの ItemIdData を未使用状態にして、そのページ内の Item データ領域のフラグメンテーションを取り除いていくだけである。フリースペースを持つページが存在している場合には、ページ内の VACUUM 処理に加え、ヒープを後ろからスキャンしながら有効タプルをフリースペースを持つページに移動し、ヒープ全体の切詰めを試みる。最後に共有フリースペースマップと pg_class カタログの統計情報を更新している。なお VACUUM 対象タプルがインデックスを持っている場合には、関連するインデックッスリレーションの VACUUM も行われる。また、移動された有効タプルがインデックスを持っている場合には、インデックスのエントリの更新が行われる。

ソースコードを見ていくと、主なシーケンスは次のようになっている。

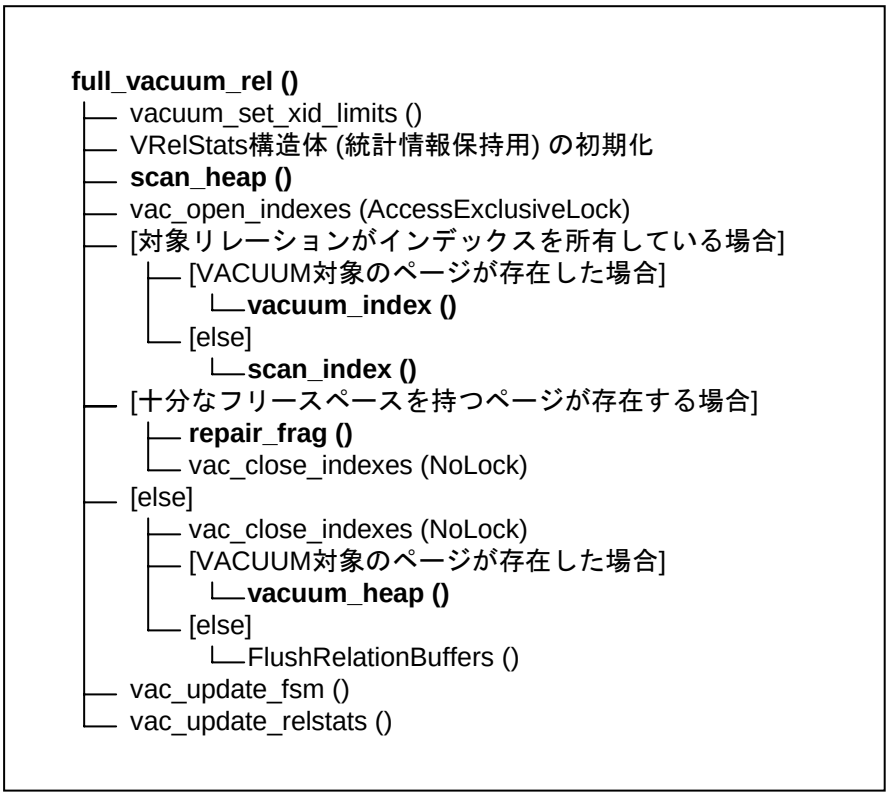

図 6-1 FULL VACUUM の処理の流れ

1. vacuum_set_xid_limits(): 実行中の全トランザクション中での一番古い xmin の取得と、FREEZE する XID のカットオフポイントの計算を行う。VACUUM の対象となるリレーションが共有リレーションの場合は、全バックエンドのトランザクションが対象となり、そうでない場合は、同一データベース上の全バックエンドのトランザクションが対象となる。
2. scan_heap()：リレーションヒープをスキャンしてタプルの状態を判定し、VACUUM 対象ページとフリースペースを持つページのリストを構築する。
3. vac_open_indexes()：リレーションが所有している全インデックスを AccessExclusiveLock でオ

ープンする。

4. scan_heap() の結果、インデックスを持っており、かつ VACUUM 対象のタプルが見つかった場合には、vacuum_index() でインデックスの VACUUM 処理を実行する。インデックスを持つが VACUUM 対象のタプルが見つからない場合には、scan_index() でインデックスのスキャンを行う。これは単に統計情報を更新するために行われている。
5. scan_heap() の結果、タプルの移動が可能なフリースペースを持つページが存在していれば、repair_frag()でテーブル全体に渡るフラグメンテーションの修繕と切詰めを試みる。インデックスを持つタプルが別ページに移動された場合には、タプルの移動後にインデックスの VACUUM 処理を行う必要があるので、repair_frag()の呼び出し後に vac_close_indexes()を行うようにしている。
6. scan_heap() の結果、タプルの移動が可能なフリースペースを持つページが見つからず、VACUUM 対象のタプルを持つページは存在していれば、vacuum_heap()でページ内でのデータ領域のフラグメンテーションを取り除くが、他のページへタプルを移動することはない。なお、VACUUM 対象のタプルを持つページもフリースペースを持つページも見つからなかった場合には、FlushRelationBuffers()を呼び出してリレーションのダーティページをディスクにフラッシュしている。これは scan_heap() 時にタプルの t_infomask ビットフラグの追加などの変更が行われている場合があるためである。
7. vac_update_fsm()： 現在持っているフリースペース情報で、共有フリースペースマップを更新する。フリースペースマップの平均リクエストサイズを求め、それより大きなサイズのフリースペースの情報を storage/freespace/freespace.c の RecordRelationFreeSpace()で共有フリースペースマップに記録する。
8. vac_update_relstats()：リレーションのための統計情報の更新を行う。pg_class カタログの relpages (ページ数の推測値)、reltuples (タプル数の推測値)、relhasindex (インデックス所有フラグ)、relhaspkey (主キー所有フラグ)を更新している。

6.2. 情報収集のためのヒープスキャン-- scan_heap()

scan_heap() では、対象リレーションのヒープをスキャンして全タプルの状態を検査し、VACUUM 対象ページ情報のリストと、ヒープを移動するのに十分なフリースペースを持つページ情報のリストを構築する。なおヒープのスキャン中には、必要に応じてタプルの t_infomask ビットフラグの更新も行われている。ここで更新されるのは、主に、MVCC でタプルの生死を確認するときに使う XID が、コミットされているかどうかを判断するためのフラグである。

なお、scan_heap() 中において、ページ中のデータ領域のフラグメンテーションを取り除く PageRepairFragmentation() が呼び出されているが、除去の後のフリースペースの値を求めるための目的であり、これは複製したページを対象に実行している。この時点では本当のページはまだ変更されていない。

scan_heap() で構築する VACUUM 対象ページ情報、および十分なフリースペースを持つページ情報のリストは、次の構造体で定義されている。

```
typedef struct VacPageListData
{
    BlockNumber empty_end_pages;    /* リレーションの連続する空の終了ページの数 */
    int         num_pages;                          /* pagedesc で使用されているページ数 */
    int         num_allocated_pages;          /* pagedesc に割り当てられているページ数 */
    VacPage    *pagedesc;                        /* ページ(VacPage)の配列ポインタ */
} VacPageListData;

typedef VacPageListData *VacPageList;
```

```
typedef struct VacPageData
{
    BlockNumber blkno;           /* このページの BlockNumber */
    Size        free;                       /* このページのフリースペースサイズ */
    uint16      offsets_used;      /* VACUUM で使用された OffNums の数 */
    uint16      offsets_free;        /* 空きか空きになる OffNums の数 */
    OffsetNumber offsets[1];    /* 空き OffNums の配列 */
} VacPageData;

typedef VacPageData *VacPage;
```

VacPageListData 構造体はリレーション単位の情報であり、pagedesc にはヒープのスキャンで収集されたページ情報(VacPageData)の配列がポインタで格納される。empty_end_pages は、後に実行されるリレーションの切詰め処理の情報である。

VacPageData 構造体はページ単位の情報であり、offsets[]にはヒープのスキャンで収集された空き(未使用、削除等)タプルの OffsetNumber が配列で格納されていく。offsets_used はタプルの回収が行われた数であり、scan_heap() 終了時点ではまだ 0 である。

最終的に、引数 vacuum_pages には VACUUM 対象ページ情報のリストが構築され、引数 fraged_pages にはタプルを移動するための十分なフリースペースがあるページ情報のリストが構築される。fraged_pages は、後にリレーション全体のフラグメンテーションの修繕を試みる repair_frag() で利用されることになる。

ページ内に十分なフリースペースがあるかどうかの基準は、そのリレーションの全 VACUUM 対象外のタプルの最小タプルサイズより大きなフリースペースを持っているということである。しかしながら、実際には、ヒープをスキャンしながら最小タプルサイズも求めているので、スキャン当初においては正確な最小タプルサイズの判定が行えない。そこで、もう一つの判定条件として、BLCKSZ / 10 よりも大きなフリースペースを持っていれば、十分なフリースペースを持つページとしている。

scan_heap()で構築される、VACUUM 対象ページ情報リスト(引数 vacuum_pages)と、フリースペースページ情報リスト(引数 fraged_pages)の関連は、次のようになる。

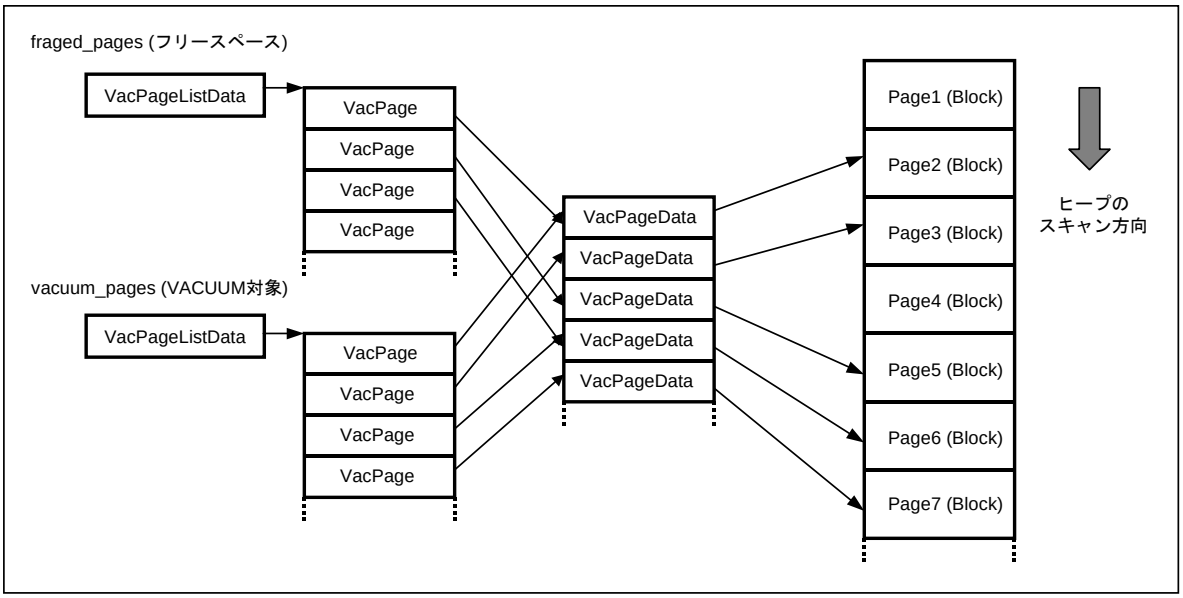

図 6-2 VACUUM 対象のページ情報とフリースペースページ情報の関連

次の図に、VACUUM 対象となるページ情報とタプルの関連を示す。

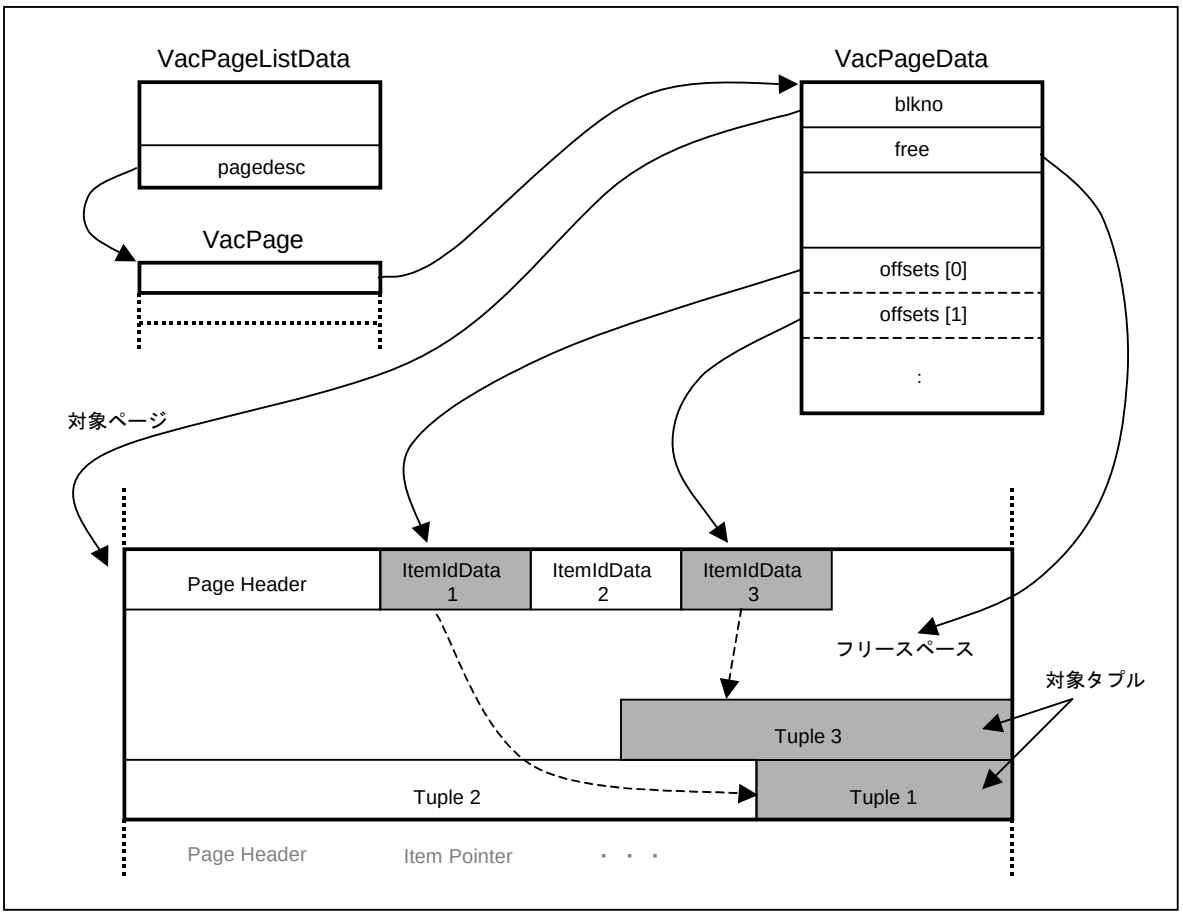

図 6-3 VACUUM 対象となるページ情報とタプルの関連

FULL VACUUM では、タプルが HEAPTUPLE_RECENTLY_DEAD の状態で、かつそのタプルは更新されて死んだタプルと判断すれば、タプルの依存状態（自身の TID と更新先の TID）を VTupleLinkData 構造体で保持しておき、関連するタプル同士が近くに配置されるようにタプルチェ

ーン全体の移動を試みる。ヒープのスキャンによって収集された VTupleLinkData は、リレーションの統計情報を保持している VRelStats 内にポインタを持ち、後に呼び出される repair_frag() タプルチェーン全体の移動処理で参照されることになる。
以下が VTupleLinkData の定義である。

```
typedef struct VTupleLinkData
{
    ItemPointerData new_tid;          /* 更新されている新しいタプルの TID */
    ItemPointerData this_tid;         /* タプル自身の TID */
} VTupleLinkData;

typedef VTupleLinkData *VTupleLink;
```

ヒープのスキャン処理時には、後に共有フリースペースマップの更新や VACUUM VERBOSE オプションで表示する際の、リレーション単位の統計情報も収集される。以下がその構造体である。

```
typedef struct VRelStats
{
    BlockNumber   rel_pages;          /* リレーションのページ数 */
    double        rel_tuples;         /* 有効タプル数 */
    Size          min_tlen;           /* 有効タプルの最小タプルサイズ */
    Size          max_tlen;           /* 有効タプルの最大タプルサイズ */
    bool          hasindex;           /* インデックス所有フラグ */
    int           num_vtlinks;        /* vtlinks 中の VTupleLinkData の数 */
    VTupleLink    vtlinks;            /* VTupleLinkData の配列ポインタ*/
} VRelStats;
```

6.2.1. タプル移動を行わないリレーションヒープの VACUUM -- vacuum_heap ()

ここでは、ページ単位の VACUUM に留まり、タプルを他のページに移動することのないテーブルの FULL VACUUM 処理について説明する。scan_heap()の結果、VACUUM 対象タプルが見つかっているが移動可能なフリースペースを持つページが存在しない場合や、タプルの検査で HEAPTUPLE_INSERT_IN_PROGRESS や HEAPTUPLE_INSERT_IN_PROGRESS の状態が見つかった場合などの VACUUM 処理がこれにあたる。

vacuum_heap()では、VACUUM 対象ページ情報のリスト(VacPageListData)から順にページ情報(VacPageData)を取り出して行き、VACUUM 対象となるページをバッファへ読込んで vacuum_page()を実行するだけである。
全ての VACUUM 対象ページに対して vacuum_page() が実行し終わったら、最後に空の終了ページが存在しているかチェックして、存在していたら RelationTruncate() でリレーションの切詰めを行なっている。

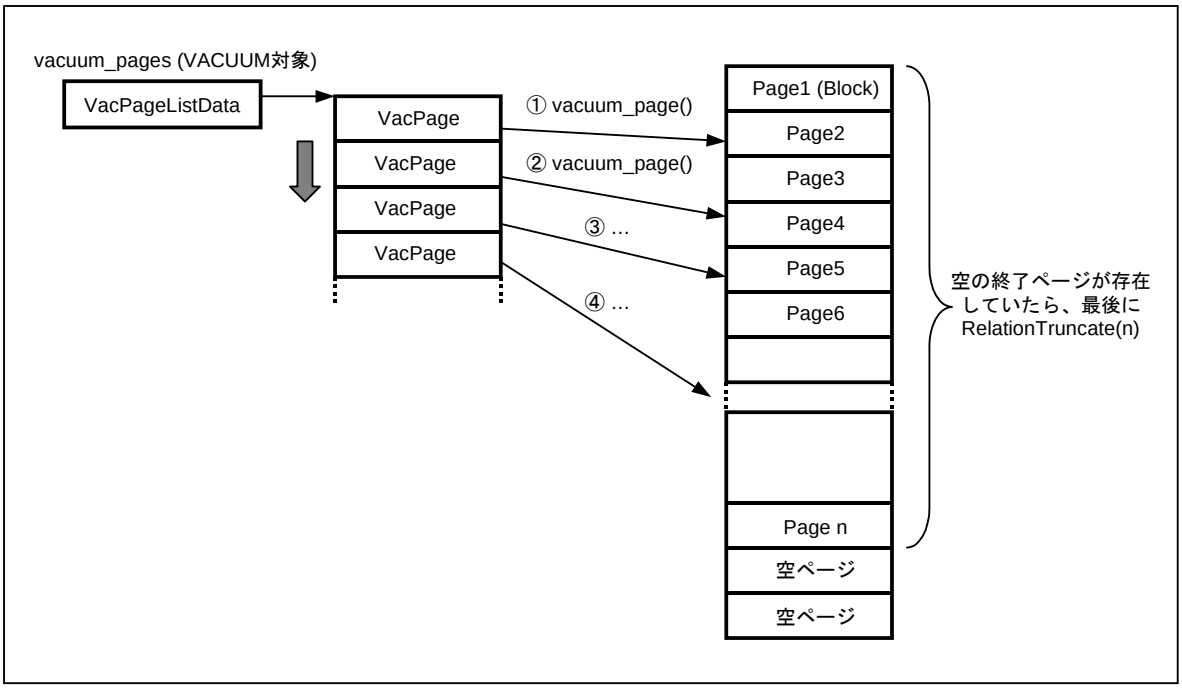

図 6-4 タプル移動のない場合の FULL VACUUM のデータ構造

6.2.2. タプルの移動を行うリレーションヒープの VACUUM -- repair_frag ()

ここでは、生きている有効なタプルをフリースペースを持つページに移動し、リレーションヒープ全体に渡るフラグメンテーションの修繕を試みる VACUUM 処理について説明する。
scan_heap()の結果、タプルの移動可能なフリースペースを持つページが存在している場合の VACUUM 処理がこれにあたる。

タプルの移動手順は以下のようになっている。

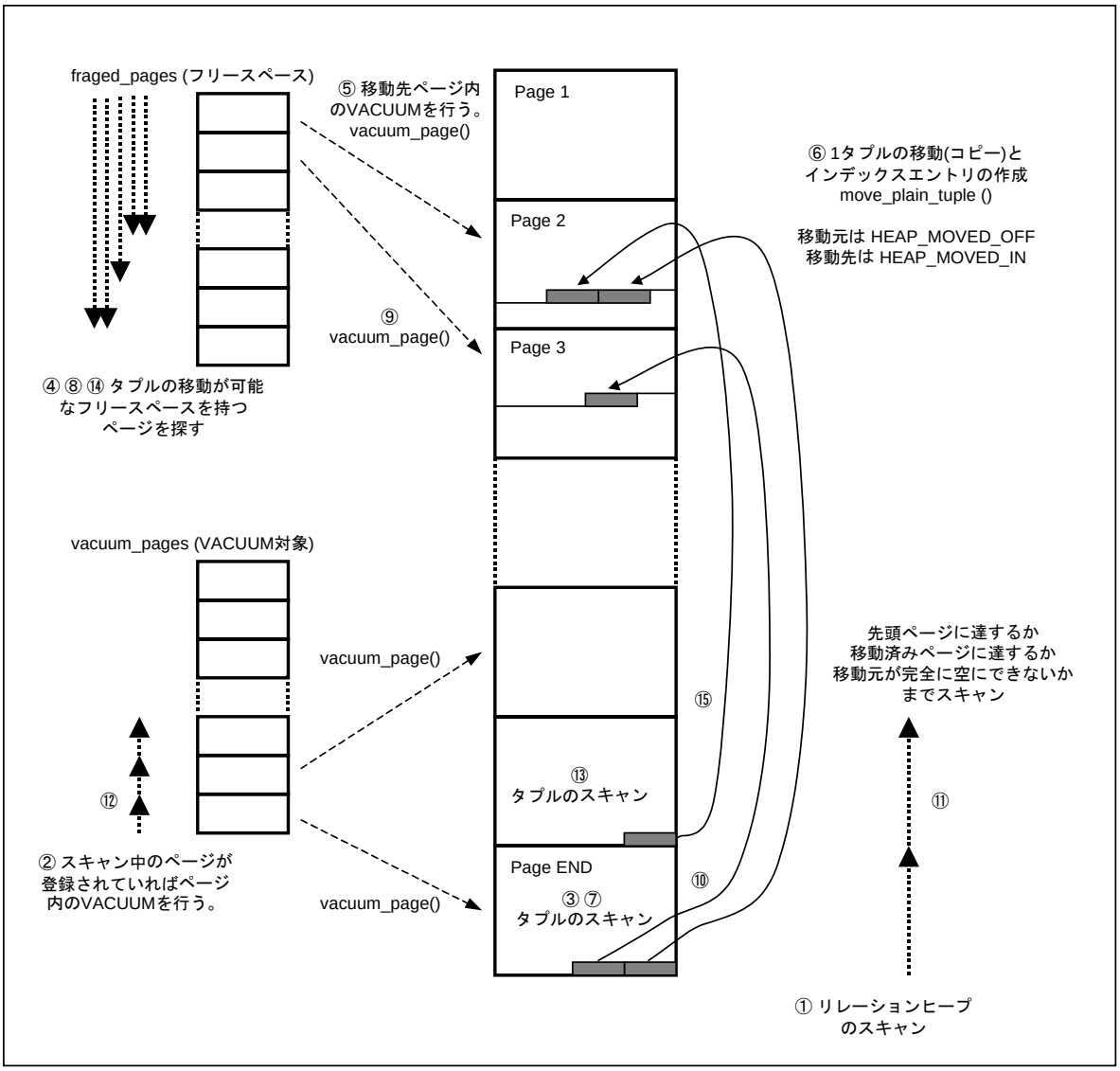

図 6-5 リレーション全体に渡るフラグメンテーションの修繕

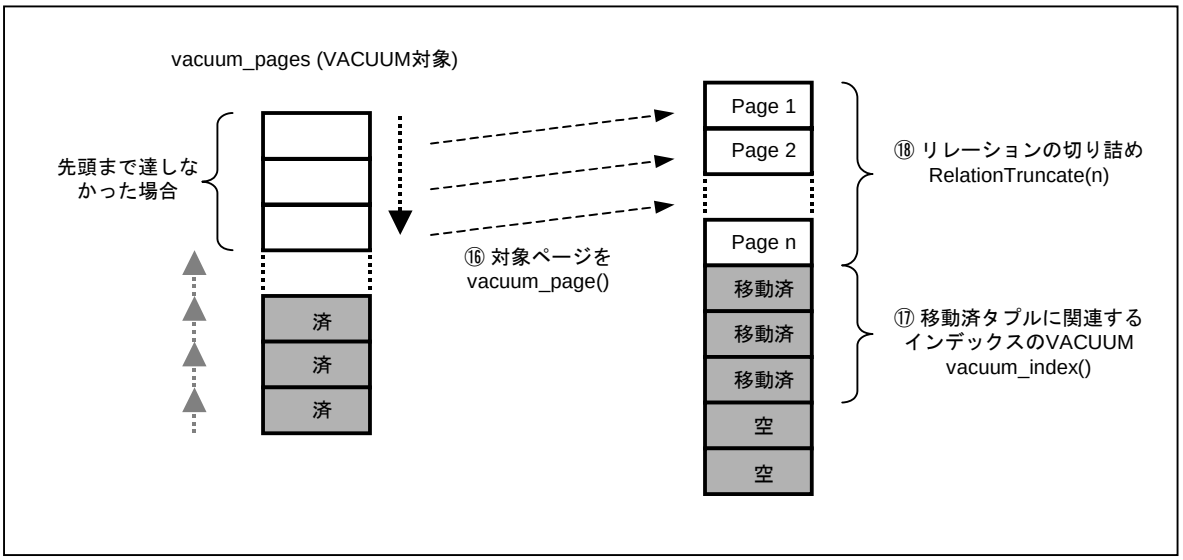

図 6-6 リレーション後部の切詰め

1. タプルの移動を目的としたリレーションヒープのスキャンを後方のページから開始(図 6-5 ①)し、VACUUM対象のページ情報リスト(vacuum_pages)の後方より、現在スキャン中のページと同じページが登録されているかチェックする。VACUUM対象ページとして登録されている場合は、このページ内のVACUUM処理 vacuum_page() を実行する。(図 6-5 ②)
2. 現在のページ中のタプルのスキャンを実行する。(図 6-5 ③) ページ内の先頭のItemIdDataより順に、有効なタプルの移動を試みる。FULL VACUUMの場合はリレーションにACCESS EXCLUSIVEロックを持っているので、移動を試みるタプルにはt_infomaskにHEAP_XMIN_COMITTEDビットが立っているはずであるが、例外としてscan_heap()で判断されたタプルチェーンの一部[22] の場合も想定されている。なおここでは、通常のタプルの移動処理について説明を進めていく。
3. 次にフリースペースがあるページ情報リスト(fraged_pages)より、移動タプルサイズより大きなフリースペースを持つ移動先のページ情報を取得する。(図 6-5 ④) 移動先ページは移動元ページより低位置のページとなる。もし移動先のページが一度もVACUUMによるタプル移動がなされていない場合には、移動先ページ内のVACUUM処理 vacuum_page() を実行する。(図 6-5 ⑤) なお、移動先のページ情報の取得時には、既に全タプルの移動が終了しているページ以降の検索は行わない。
4. 移動元ページから移動先ページへタプルを移動し、移動先のページ情報(fraged_pagesが持っているVacPageData構造体)のフリースペースサイズ等の情報を更新する。(図 6-5 ⑥) タプルの移動処理は、move_plain_tuple()[23] を呼び出すことによって行われ、移動元のt_infomaskは、HEAP_XMIN_COMMITTED / HEAP_XMIN_INVALID / HEAP_MOVED_IN をクリア後、HEAP_MOVED_OFFを立て、t_xvacに現在のトランザクションIDをセットしている。移動先では、t_infomaskのHEAP_XMIN_COMMITTED / HEAP_XMIN_INVALID / HEAP_MOVED_OFFをクリアした後、HEAP_MOVED_INを立て、同様にt_xvacをセットする。なおXLOGバッファには、access/heap/heapam.c のlog_heap_move()[24] を呼び出すことによってヒープバッファの移動を知らせ、PageHeaderDataにLSNとWALのTLIをセットしている。また、必要に応じて移動したタプルのためのインデックスエントリが作成される。インデックスの作成にはエグゼキュータのインデックスエントリ生成機構を利用し、ExecStoreTuple() / ExecInsertIndexTuples() / ResetPerTupleExprContext() を呼び出すことによって実現している。
5. 1タプルの移動が終了したら、次のタプルの移動を行う。同ページ内の最終ItemIdDataまでタプルの移動を続ける。処理内容は上記 2. ～ 4. と同様である。(図 6-5 ⑦⑧⑨⑩)
6. ページ中の全タプル移動が終了したら、次(一つ前)のページのタプル移動を行う。処理内容は上記 1. ～ 5. と同様 (図 6-5 ⑪⑫⑬⑭⑮) である。タプルの移動に伴うリレーションヒープの移動元ページのスキャンは、既にタプル移動されているページに達するか、リレーションヒープの最初のページに達するか、あるいは完全に空にできないページが見つかった場合まで続けられる。
7. リレーションヒープのスキャンから抜けた後、一つでもタプルの移動が行われていれば、後でリレーションの切詰めを試みるためにRecordTransactionCommit() を呼び出して現在のトランザクションの状態を記録する。

[22] 更新処理によってタプルの状態が HEAPTUPLE_RECENTLY_DEAD と判断されたチェーンタプルの一部。
[23] タプルチェーンの一部である 1 タプルを移動する場合は、move_chain_tuple()を呼び出すことになる。
[24] 最終的には XLogInsert(XLOG_HEAP_MOVE) を呼び出している。

8. タプル移動処理の際に、VACUUM対象ページ情報リスト(vacuum_pages)の先頭まで達せずにVACUUM処理が実行されていないページが残っている場合には、残っているVACUUM対象ページに対してページ内のVACUUM処理 vacuum_page() を実行する。(図 6-6 ⑯)
9. タプルの移動先の対象ページに対してをスキャンを行い、t_infomask に HEAP_MOVED_IN が立っているタプルに、HEAP_XMIN_COMMITTED のセットと HEAP_MOVED_IN と HEAP_MOVED_OFF のクリアを行う。この処理は update_hint_bits() で行われ、将来のトランザクションにおけるタプル状態の検査時間の節約を目的としている。
10. CommandCounterIncrement()を呼び出し、システムカタログの変更を可視状態にする。
11. 移動元ページ中の移動済タプルに関連している、古くなった移動前のインデックスを vacuum_index() でVACUUMする。(図 6-6 ⑰)
12. 最後に、リレーションの全てのダーティページをディスクにフラッシュした後、空の終了ページや、タプルの移動によって必要のなくなった空の移動元ページを、RelationTruncate() を呼び出してリレーションの後ろの切詰めを行っている。(図 6-6 ⑱)

7. 付録：VACUUM 処理関連の関数リファレンス

VACUUM と ANALYZE に共通のコード

vacuum()	VACUUM および ANALYZE コマンドのエントリポイント
get_rel_oids()	リレーション OID のリストを vac_context 内で構築
vacuum_set_xid_limits()	最古の xmin および FREEZE する XID のカットオフポイントを求める
vac_update_relstats()	1 リレーションのための統計情報の更新
vac_update_dbstats()	1 データベースのための統計情報の更新
vac_truncate_clog()	コミットログの切詰めを試みる

FULL VACUUM と CONNCURRENT VACUUM に共通のコード

vacuum_rel()	1 つのリレーションヒープの VACUUM 処理

FULL VACUUM のためのコード

full_vacuum_rel()	1 つのリレーションヒープの FULL VACUUM 処理
scan_heap()	リレーションヒープのスキャン
repair_frag()	リレーションのフラグメンテーションの修繕を試みる
move_chain_tuple()	タプルチェーンの一部である 1 つのタプルを移動する
move_plain_tuple()	タプルチェーンの一部でない 1 つのタプルを移動する
update_hint_bits()	タプル中のステータスビット(t_infomask)の更新
vacuum_heap()	死んでいるタプルを解放
vacuum_page()	指定ページの上の死んでいるタプルを解放して、データ領域のフラグメンテーションを修繕する
scan_index()	1 インデックスをスキャンして統計を更新する
vacuum_index	1 インデックスを VACUUM して統計を更新する
tid_reaped()	index_bulk_delete()の判定用。特定の tid は VACUUM 対象か？
dummy_tid_reaped()	scan_index のためのダミー用の判定ファンクション
vac_update_fsm()	マップが持つかもしれない幾らかの古い情報を破棄して、現在持っているリレーションのフリースペースに関する情報で、共有フリースペースマップを更新する。
copy_vac_page()	VacPage 構造体の複製を作る
vpage_insert()	VacPageList の pagedesc に VacPage を追加する
vac_bsearch()	VACUUM 用バイナリサーチ関数。bsearch()と同等
vac_cmp_blk()	qsort()と bsearch()用のための比較用ルーチン。BlockNumber の比較用ファンクション。
vac_cmp_offno()	qsort()と bsearch()用のための比較用ルーチン。OffsetNumber の比較用ファンクション。

vac_cmp_vtlinks()	qsort()と bsearch()用のための比較用ルーチン。VTupleLink の new_ tid でソートする際の比較用ファンクション。
vac_open_indexes()	指定された種類のロックを得て、与えられたリレーションの全インデックスをオープンする。
vac_close_indexes()	vac_open_indexes によって得られたリソースを解放する。 任意でロックを解放する。(ロックを保つには NoLock を指定)
vac_is_partial_index()	部分インデックスかどうかの判定。 部分インデックスの場合 true。
enough_space()	ページ中にタプルを格納する十分なフリースペースがあるかをチェックする
vac_init_rusage()	カレントリソースの使用に関する情報と時刻を取得
vac_show_rusage()	vac_init_rusaged()で取得した情報を文字列に変換
vacuum_delay_point()	割り込みとコストベースの VACUUM 遅延についてのチェック
ExecContext_Init()	エグゼキュータのインデックスエントリ生成機構を使うことができるように ExecContextData を初期化する
ExecContext_Finish()	ExecContextData の終了、解放

CONCURRENT VACUUM のためのコード

lazy_vacuum_rel ()	1 つのリレーションヒープの CONCURRENT VACUUM 処理
lazy_scan_heap()	リレーションヒープのスキャンと VACUUM 処理
lazy_vacuum_heap()	収集した VACUUM 対象タプルのエントリリストを元に lazy_vacuum_page()を呼び出す
lazy_vacuum_page()	ページ内の上の死んでいるタプルを解放して、データ領域のフラグメンテーションを修繕する
lazy_scan_index()	1 インデックスをスキャンして統計を更新する
lazy_vacuum_index()	1 インデックスを VACUUM して統計を更新する
lazy_truncate_heap()	リレーションの空の終了ページを切詰める
count_nondeletable_pages()	空の終了ページのヒープをスキャンして、現在でも空のページであるか確認する
lazy_space_alloc()	LVRelStats 構造体に maintenance_work_mem 分の領域と max_fsm_pages 分のメモリ領域を割り当てる
lazy_record_dead_tuple()	1 つの VACUUM 対象タプルを LVRelStats 構造体に保持する
lazy_record_free_space()	1 つのフリースペース情報を LVRelStats 構造体に保持する
lazy_tid_reaped()	index_bulk_delete()の判定用。特定の tid は VACUUM 対象か？
dummy_tid_reaped()	lazy_scan_index のためのダミー用の判定ファンクション
lazy_update_fsm()	マップが持つかもしれない幾らかの古い情報を破棄して、現在持っているリレーションのフリースペースに関する情報で、共有フリースペースマップを更新する。
vac_cmp_itemptr()	qsort()と bsearch()用のための比較用ルーチン。

	ItemPointer 中の BlockNumber の比較
vac_cmp_page_spaces()	qsort()と bsearch()用のための比較用ルーチン。 PageFreeSpaceInfo 中の BlockNumber の比較