PostgreSQL ハッシュインデックス解析（7.4 系編）

NTT サイバースペース研究所 寺本純司

目 次

図 目 次

表 目 次

1 はじめに

著者らは、オープンソースで公開されているデータベース管理システム PostgreSQL の内部構造の解析を行っている。特に、PostgreSQL のサブシステムのうち、データへのアクセス手段の一つである、ハッシュインデックスの内部構造と動作原理について解析を進めてきた。

この資料では、ハッシュインデックス解析資料として、インデックスアクセスメソッドに関するシステムカタログ（PostgreSQL が管理する内部情報）についての説明（2章）と、ハッシュインデックスのデータ構造の説明（3章）を行う。さらに、PostgreSQL が実装しているハッシュアルゴリズム「線形ハッシュ法」についての説明（4章）を行う。

また、ハッシュインデックス実装に必要な関数のうち、新規インデックスの構築をする `hashbuild()`（5章）、インデックスにタプルを挿入する `hashinsert()`（6章）、インデックスから次の有効なタプルを得る `hashgettuple()`（7章）、インデックスの新規スキャンを開始する `hashbeginscan()`（8 章）、インデックスのスキャンを再開する `hashrescan()`（9章）、インデックスのスキャンを終了する `hashendscan()`（10章）、現在のインデックススキャン位置を記録する `hashmarkpos()`（11章）、記録したインデックススキャン位置を復元する `hashrestrpos()`（12章）、インデックスをバルク削除する `hashbulkdelete()`（13章）について説明する。

なお、この資料が対象とする PostgreSQL の版は、7.4.3 である。

2 インデックスアクセスメソッドに関するシステムカタログ

システムカタログとは、リレーショナルデータベースの管理システムがテーブルや列の情報などのメタデータの概要と内部的な情報を格納する場所である。

インデックスアクセスメソッドに関係するシステムカタログは、`pg_am` である。ここに必要な情報を格納することによって、インデックスアクセスメソッドを登録することができる。

`pg_am` が格納する情報を表 1に示す（[1] より引用）。また、ハッシュインデックスの場合何が登録されているかもあわせて示す。

表 1: pg_am の列

名前	型	参照先	説明	ハッシュの値
amname	name		アクセスメソッド（AM）の名前	hash
amowner	int4	pg_shadow.usesysid	所有者のユーザ ID（現時点では未使用）	1
amstrategies	int2		この AM 用の演算子ストラテジの数	1
amsupport	int2		この AM 用のサポートルーチンの数	1
amorderstrategy	int2		インデックスがソート順を提供しない場合はゼロ、その他の場合はソート順を記述する戦略演算子の戦略番号	0
amcanunique	bool		AM が一意性インデックスをサポートするかどうか	FALSE
amcanmulticol	bool		AM が複数列インデックスをサポートするかどうか	FALSE
amindexnulls	bool		AM がインデックスエントリに NULL を許すかどうか	FALSE
amconcurrent	bool		AM が同時更新をサポートするかどうか	TRUE
amgettuple	regproc	pg_proc.oid	“つぎの有効なタプル” 関数	hashgettuple
aminsert	regproc	pg_proc.oid	“このタプルを挿入” 関数	hashinsert
ambeginscan	regproc	pg_proc.oid	“新規スキャンを開始” 関数	hashbeginscan
amrescan	regproc	pg_proc.oid	“このスキャンを再開” 関数	hashrescan
amendscan	regproc	pg_proc.oid	“このスキャンを終了” 関数	hashendscan
ammarkpos	regproc	pg_proc.oid	“現在のスキャン位置を記録” 関数	hashmarkpos
amrestrpos	regproc	pg_proc.oid	“記録したスキャン位置を復元” 関数	hashrestrpos
ambuild	regproc	pg_proc.oid	“新規インデックスをビルド” 関数	hashbuild
ambulkdelete	regproc	pg_proc.oid	バルク削除関数	hashbulkdelete
amvacuumcleanup	regproc	pg_proc.oid	VACUUM 後クリーンアップ関数	Invarid 値
amcostestimate	regproc	pg_proc.oid	インデックススキャンのコストの推測値を概算する関数	hashcostestimate

amstrategies と amsupport の値については次のような意味がある。

amstrategies ハッシュインデックスの場合、演算子として必要なのは、ビット毎の類似性を表す「等しい」という演算子のみなので、値として 1 が登録されている。

amsupport ハッシュインデックスの場合、「キーのハッシュ値を計算」するサポートルーチンが 1 つ必要なので、値として 1 が登録されている。ただし、ハッシュ値を計算するにあたって、対象となるデータ型に合わせて別個のサポートルーチンが必要となるので、サポートルーチンの実体は型の数だけ複数存在する。

以上からわかるように、ハッシュインデックスアクセスメソッドの実装には、演算子ルーチンを 1 つと、ハッシュ値計算ルーチンをサポートするデータ型の数だけ用意し[1]、そして表 1の後半に示された 11 個の関数のうち、「VACUUM 後クリーンアップ関数」を除いた 10 個を用意する必要がある。ちなみに、VACUUM 後クリーンアップ関数は 7.4 から新設された関数で、現状 BTree インデックスアクセスメソッドのみに実装されている。

3 ハッシュインデックスのデータ構造

この章では、ハッシュインデックスがどのようにデータベースファイルに格納されているかについて示す。

一般的なデータベースファイルの構造は文献 [1] の「VII. 内部情報 第 51 章 “ページファイル”」にも記されているので参照されたい。

以下、ハッシュインデックス固有の情報を含めた構造の解説をする。なお、各種数値はデフォルト値の場合を想定している。

3.1 ページファイル

PostgreSQL において、表やインデックスなどを格納するファイルをページファイルと呼ぶ。ページファイルの中は、デフォルトでは 8kbyte のページが複数入っている。

各ページにはブロック番号が振られており、そのブロック番号単位でアクセスする。ブロック番号の範囲は 0 から 0xFFFFFFFE（4294967294）までである[2]。

3.2 ページ

ページにはデータと付帯情報が格納されている。

ハッシュインデックスの場合、

- メタページ
- バケツページ
- オーバーフローページ
- ビットマップページ

の 4 種類のページが使われている。まず、ページの基本構造について説明し、その後 4 種類のページそれぞれについて説明する。

3.2.1 ページの基本構造

ページの基本構造は、大きく 3 つのブロックからなっている。

[1] `backend/access/hash/hashfunc.c` 参照のこと。これらの関数は、Postgres Function Manager 経由で、データ型に応じ適切な関数が呼ばれて使用される。詳しくはインデックスアクセスメソッド共通の関数 `index_getprocinfo()` を参照のこと。

[2] 0xFFFFFFFF は Invalid 値として定義されている。

1. ページヘッダ領域
2. データ格納領域
3. スペシャルデータ領域

構造を図示すると図 1のようになっている。

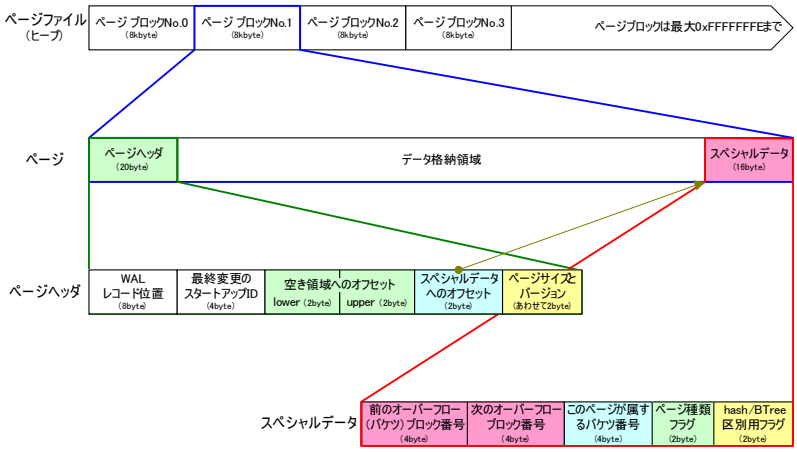

図 1: ページの基本構造

以下、各領域について概説する。

3.2.1.1　ページヘッダ領域　ページヘッダ領域には、

1. WAL（Write Ahead Log）レコード位置
2. 最終変更のスタートアップ ID
3. データ格納領域における空き領域の開始点へのオフセット
4. データ格納領域における空き領域の終了点へのオフセット
5. スペシャルデータ領域へのオフセット
6. ページサイズとレイアウトバージョン情報

が格納されている。このうち、最初の 2 つのフィールドは WAL に関する項目であり、ハッシュインデックスでは使用しない。

3.2.1.2　データ格納領域　ページヘッダ領域とスペシャルデータ領域にはさまれた部分に各種データが格納される。この部分の利用の仕方には特に決まりがない。ハッシュインデックスの場合も、バケツページ・オーバーフローページに関しては標準のヒープタプルと同様の構造を用いているが、メタページやビットマップページはまったく別の構造を用いている。

3.2.1.3　スペシャルデータ領域　スペシャルデータ領域には、アクセスメソッド固有の情報を格納するための場所である[3]。ハッシュインデックスの場合、

1. 前のオーバーフロー（またはバケツ）ブロック番号
2. 次のオーバーフローブロック番号

[3]通常のテーブルの場合、利用されない領域のために存在しない。

3. このページが属するバケツ番号
4. ページ種類フラグ
5. hash か BTree かを区別するためのフラグ

が格納されている。

このうち最後のフラグは、本来将来使用するために未使用としている領域に、pg_filedump コマンド（PostgreSQL のファイルをダンプして内部を解析する、RedHat 社作成のツール）向けに、このスペシャルデータが hash 用のものか BTree 用のものか区別しやすいようなフラグを暫定的に入れているものである。今後変更の可能性があるので注意する。

3.2.2 バケツページとオーバーフローページ

インデックスエントリを格納しているページがバケツページとオーバーフローページである。両者はページ種類フラグの違いこそあるが同じ構造をしている。ページレイアウトは、通常の表におけるタプルを格納するページのレイアウトとほぼ同等である。

ページの構成は図 2のようになっている。

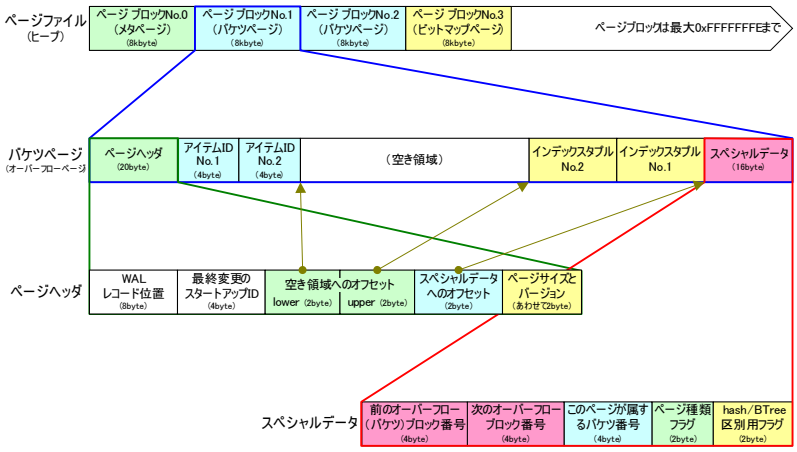

図 2: バケツページ・オーバーフローページの構成図

データ格納領域に、インデックスタプルが格納されていく。インデックスタプルはアイテム ID と対になっている。アイテム ID は、ページヘッダに続いてデータ格納領域の先頭から割り当てられる。それに対応するインデックスタプルは、データ格納領域の最後から割り当てられる。

インデックスタプルとアイテム ID が格納されるたびに、ページヘッダの空き領域のオフセットが更新される。

スペシャルデータは次のような値が入る。

前のオーバーフロー（またはバケツ）ブロック番号

次のオーバーフローブロック番号 以上 2 つはバケツの双方向チェーンを表現するために用いられる

このページが属するバケツ番号 バケツページの場合は自分の番号、オーバーフローページの場合は自分が属しているバケツ番号

ページ種類フラグ バケツページの場合 LH_BUCKET_PAGE、オーバーフローページの場合 LH_OVERFLOW_PAGE

アイテム ID とインデックスタプルの構造についてさらに詳しく示したものが図3である。

アイテム ID にはインデックスタプルオフセットが情報として含まれている。この情報で、インデックスタプルの位置を把握する。また、フラグにはタプルの状態が格納されている（使用中の LP_USED か削除済の LP_DELETE）。データサイズには、インデックスタプルの大きさが格納されている。

インデックスタプルには、

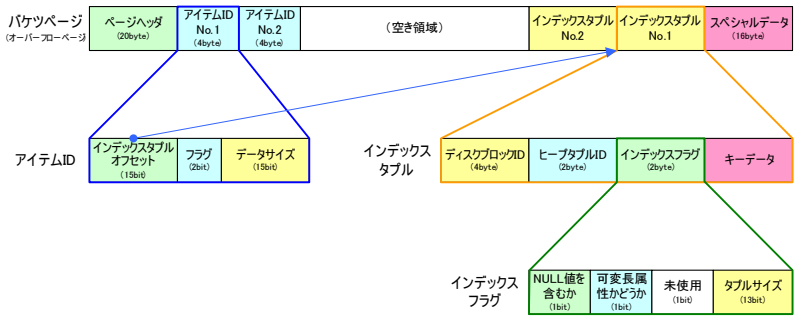

図 3: アイテム ID とインデックスタプルの構造

1. インデックスしたヒープタプルへのリファレンス
2. インデックスタプルの情報
3. キーデータ

が格納されている。この構造は各種インデックス共通である。複数列インデックスの場合、キーデータの後にサブキーデータが続くが、ハッシュインデックスでは複数列インデックスをサポートしないのでキーデータは 1 つである。

ヒープタプルへのリファレンスには、タプルを含むページの、ディスク上の位置を示すディスクブロック ID と、そのページ内でのタプルの位置を示すオフセット番号であるヒープタプル ID が含まれている。

インデックスタプルの情報はインデックスフラグに格納されている。この中には、NULL 値を含んでいるかどうか、可変長属性かどうかのフラグと、インデックスタプルのサイズが含まれている。

最後に、実際のキーデータがつながっている。

3.2.3 メタページ

ハッシュインデックスページファイル中の先頭ページ（ブロック番号 0）はメタページと呼ばれる特殊なページである。ページの構成は図 4のようになっている。

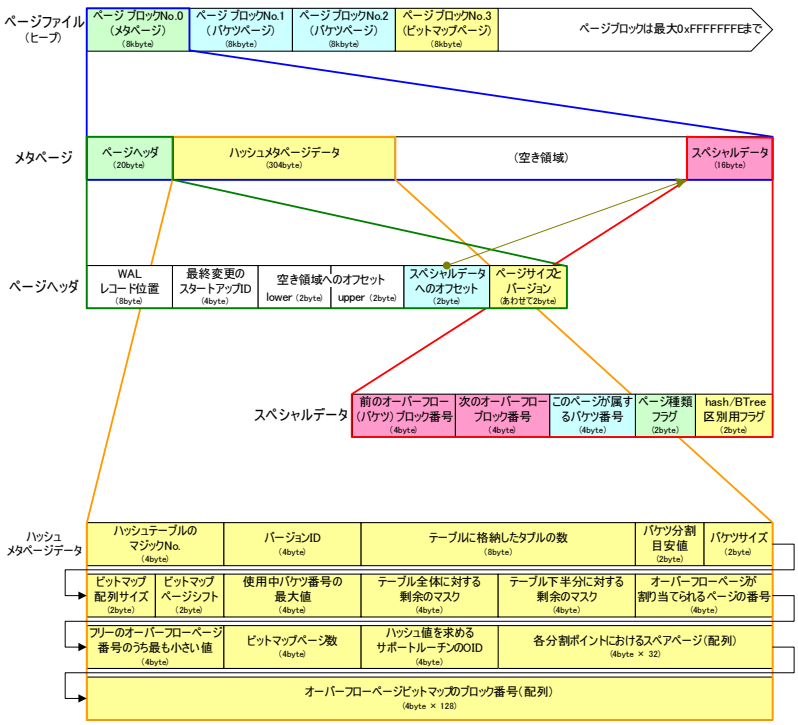

図 4: メタページの構成図

データ格納領域に、ハッシュメタページデータが 1 つ格納されている。これは固定長のデータなので、ページヘッダ内の「空き領域へのオフセット」情報は初期値のまま更新されない。

スペシャルデータには、ページ種類フラグとしてメタページを示す LH_META_PAGE が入り、前／次のオーバーフローブロック番号、このページが属するバケツ番号にはすべて invalid 値が入る。

ハッシュメタページデータには表 2のような情報が格納されている（HashMetaPageData 構造体参照のこと）。

表 2: ハッシュメタページデータ

構造体メンバー名称	説明
hashm_magic	ハッシュテーブルのマジック ID（0x6440640）
hashm_version	バージョン ID（7.4 以降は 1）
hashm_ntuples	インデックステーブルに格納したタプルの数
hashm_ffactor	バケツ分割の目安値
hashm_bsize	バケツサイズ
hashm_bmsize	ビットマップ配列サイズ
hashm_bmshift	ビットマップページシフト値
hashm_maxbucket	使用中バケツ番号の最大値
hashm_highmask	テーブル全体に対する剰余のマスク
hashm_lowmask	テーブル下半分に対する剰余のマスク
hashm_ovflpoint	オーバーフローページが割り当てられるページの番号
hashm_firstfree	フリーのオーバーフローページ番号のうち最も小さいもの
hashm_nmaps	ビットマップの数
hashm_procid	ハッシュ値を求めるサポートルーチンの OID
hashm_spares[]	各分割ポイントにおけるスペアページを記録した配列
hashm_mapp[]	オーバーフローページビットマップのブロック番号を記録した配列

3.2.4 ビットマップページ

ビットマップページは、各分割ポイントにおけるオーバーフローページの使用状況を記録したビットマップ配列を格納した、特殊なページである。

ページの構成は図5のようになっている。

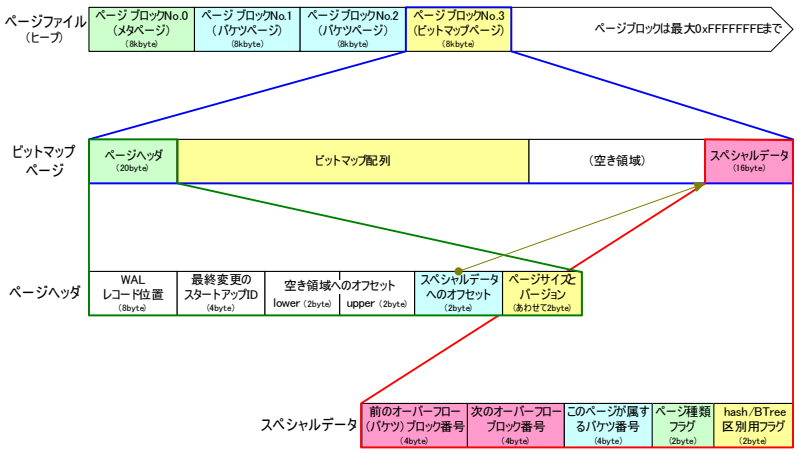

図 5: ビットマップページの構成図

データ格納領域に、ビットマップ配列が 1 つ格納されている。これは固定長のデータなので、ページヘッダ内の「空き領域へのオフセット」情報は初期値のまま更新されない。

スペシャルデータには、ページ種類フラグとしてビットマップページを示す LH_BITMAP_PAGE が入り、前／次のオーバーフローブロック番号、このページが属するバケツ番号にはすべて invalid 値が入る。

4 線形ハッシュ法について

実際に関数を説明する前に、ハッシュインデックスのアルゴリズムについて述べておく。

PostgreSQL で用いているハッシュインデックスは、Margo Seltzer らによる UNIX 向けのハッシュパッケージ [2] のインプリメンテーションであり、中身は Witold Litwin の線形ハッシュ法 [3] の実装である。そのアルゴリズムの解説は [4] が詳しい。

簡単に概説すると、以下のようになる。

今、C をキー空間とする。

C を N 個のバケツに分配する基本ハッシュ関数を

$$h_0 : C \to \{0, 1, \ldots, N-1\}$$

とする。

さらに、h_0 のスプリット関数として $h_1, h_2, \ldots, h_i$ を定義する。スプリット関数は以下の条件を満たす。

1. $h_i : C \to \{0, 1, \ldots, 2^i N - 1\}$
2. 全ての c に対し、以下のいずれかの条件を満たす。
   (a) $h_i(c) = h_{i-1}(c)$
   (b) $h_i(c) = h_{i-1}(c) + 2^{i-1}N$

これはつまりどういうことかというと、スプリット関数 h_i は、その 1 世代前の関数 h_{i-1} よりも 2 倍のバケツに分配できる関数であるということである。

このような条件を満たす関数として一般的なのは、

$$h_i(c) = c \bmod 2^i N$$

である。

一例を挙げる。N が 10 だったとし、基本ハッシュ関数として $h_0(c) = c \bmod 10$ を用いるとする。

その 10 個のバケツの中にデータを $h_0(c)$ で振り分け、順調に格納していた（図6）。

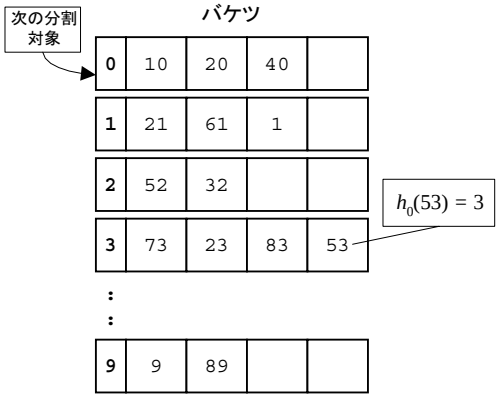

図 6: 線形ハッシュの例

さて、あるバケツにデータを挿入しようとしたら、そのバケツがオーバーフローしたとする。その時、バケツの分割を行い、格納できるバケツを増やすわけだが、ここで重要なのは、オーバーフローしたバケツを分割するのではなく、それとはまったく無関係に、0 番のバケツから順々に分割していくというところである。

この場合、一番最初なので、0 番のバケツが分割対象となる。0 番のバケツの中を確認し、スプリット関数 $h_1(c) = c \bmod 20$ を用いて、$h_1(c) = 0$ のものは 0 番バケツに、$h_1(c) = 10$ のものは 10 番バケツに入れる。つまり、分割によって 11 個めのバケツ（10 番バケツ）が付加され、0 番バケツの約半数を 10 番バケツに移すことになる（図7）。

そして以後、$h_0(c) = 0$ の時（分割済みのバケツに格納する）にはハッシュ関数として $h_1(c)$ を使い、そうでない（まだ分割されていないバケツに格納する）時には従来どおり $h_0(c)$ を用いて格納先バケツを決定する。

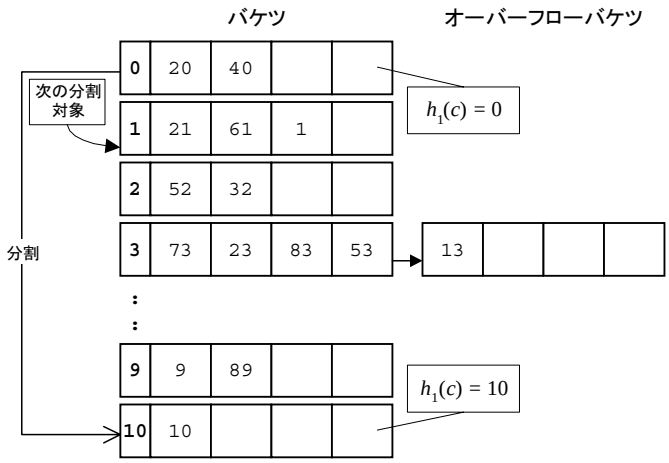

図 7: c=13 を挿入したとき

実際にオーバーフローしたバケツはどうするかというと、オーバーフローバケツをつなげてそこにあふれたデータを格納する。バケツは順々に分割されていくので、オーバーフローバケツを持つバケツもそのうち再構築される。通常の使用であれば[4]、オーバーフローバケツはそれほど必要としない。

分割が進み、最初の 10 個のバケツが分割され総バケツ数が 20 個になったら、i がインクリメントされ、基本ハッシュ関数として h_1、スプリット関数として h_2 を用いるようにする。このように、バケツの数が倍になったら、ハッシュ関数も次の世代に進む。そして分割対象バケツも 0 番に戻す。

なお、バケツの分割は、オーバーフローバケツが生成された時だけではなく、何かしらのタイミング（たとえば、バケツの数に対するタプルの数の割合が一定の値を超えたら、等）に合わせて分割してもかまわない。そのほうが性能が向上する場合がある。

以上のような処理を繰り返すことで、インデックスが構築される。

5 新規ハッシュインデックスの構築

以下、7.4.3 における該当箇所を欄外に示す。

ハッシュインデックスの構築は、`hashbuild()` 関数によって行われる。

関数の内部呼び出し構成は図8のようになっている。

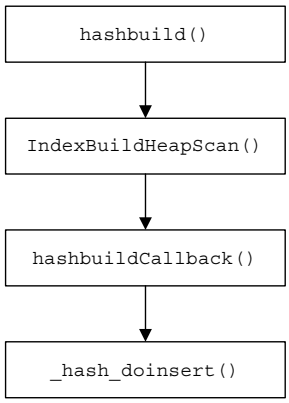

図 8: ハッシュインデックス構築関数の構成図

5.1 `hashbuild()` 関数の動作

`backend/access/hash/hash.c` *l.*44 ~ 102

`hashbuild()` 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres

[4]データに偏りがあると、この限りではなくなる。このアルゴリズムは、データの偏りに弱いと言う弱点がある。

Function Manager を経由して以下の情報を入力する。

heap（**Relation 型**）インデックスするタプルの入ったヒープリレーション

index（**Relation 型**）構築したインデックスを入れるインデックスリレーション

*indexInfo（**indexInfo へのポインタ型**）構築したインデックスに関する情報を入れる構造体へのポインタ

この関数の戻り値はない[5]。
この関数の動作は大きく次のように構成されている。

1. すでに存在するインデックスかどうかチェックする
2. ハッシュインデックスメタページの初期化
3. インデックス構築
4. （Bootstrap 時でなければ）統計量をアップデートする

以下、その詳細について説明する。

5.1.1 すでに存在するインデックスかどうかチェックする

l.61 ~ 67

この関数は、インデックスリレーションごとに 1 回だけ呼ばれることを前提にしている。そこで、インデックスリレーション index の中にいくつページがあるか調べ、そのページ数が 0 かどうか調べる。
もし 0 でない場合は、すでに何かが存在していると言うことなのでエラー。

5.1.2 ハッシュインデックスメタページの初期化

l.69 ~ 70

インデックスリレーション index を、_hash_metapinit() 関数で初期化する。
ハッシュインデックスメタページには、このインデックスのメタデータが格納される（3.2.3節を参照のこと）。

5.1.3 インデックス構築

l.72 ~ 77

まず、HashBuildState 構造体型変数 buildstate の、動作ステータスを保持する buildstate.indtuples を 0 にする。この変数には、インデックスしたタプルの数が代入される。
そして、IndexBuildHeapScan() 関数を呼び出し、ヒープリレーションをスキャンしながらハッシュインデックスを構築する。この関数の詳しい動作については後述（5.2節）する。
この関数から戻り値としてヒープタプルの数を受け取り、変数 reltuples に代入する。

5.1.4 統計量をアップデートする

l.79 ~ 99

作成したインデックスをプランナが利用する際には、pg_class カタログに格納された統計値が参考にされる。インデックス構築において、インデックスしたタプルの数とヒープタプルの数を計上したので、そのデータを新しい統計値として用いるために UpdateStats() 関数で pg_class カタログを更新する。
ただし、統計量をアップデートするのは通常のインデックス定義の間だけで、ブートストラップ処理の時に生成されるシステムカタログのインデックスの場合はアップデートしない。
UpdateStats() で統計値を更新する際、リレーションが閉じている必要があるため、heap_close(),index_close() で各リレーションを閉じておく。ただし、リレーションのロックはリリースしない。トランザクションの終了まで保持される。
以上でハッシュインデックスの構築が完了する。

[5]正確には、Postgres Function Manager 互換の関数作成用マクロによる VOID（Datum 型で 0）が返される。

5.2 IndexBuildHeapScan() 関数の動作

backend/
catalog/index.c
*l.*1323 ~ 1583

インデックス構築には IndexBuildHeapScan() 関数を用いる。この関数は、ハッシュインデックスに限らず、他のインデックス構築関数からも同様に呼び出される。実際のインデックス処理は、各インデックス構築関数から指定されたコールバック関数を呼び出すことで実現するので、この関数の主な機能は

- インデックス構築のための準備
- コールバック関数の呼び出し
- 後処理

であるといえる。
IndexBuildHeapScan() 関数は、以下の引数を持つ。

heapRelation（**Relation 型**）インデックスするタプルの入ったヒープリレーション

indexRelation（**Relation 型**）構築したインデックスを入れるインデックスリレーション

*indexInfo（**indexInfo へのポインタ型**）構築したインデックスに関する情報を入れる構造体へのポインタ

callback（**indexBuildCallback 型**）コールバックする関数を指定する関数名実引数

*callback_state（**void へのポインタ型**）コールバック関数からの結果を受け取るための変数へのポインタ

この関数の戻り値は、ヒープタプルの総数（double 型）である。
この関数の動作は大きく次のように構成されている。

1. 値のサニティチェック
2. ヒープタプルデスクリプタの取得
3. テンポラリメモリコンテキストの構築
4. 評価関数用の変数の初期化
5. 各種初期化
6. ベースリレーションの全てのタプルをスキャンし処理
7. 各種後処理
8. ヒープタプルの総数を出力

以下、その詳細について説明する。

5.2.1 値のサニティチェック

*l.*1362 ~ 1365

インデックスリレーションのオブジェクト ID が不正かどうか確認する。

5.2.2 ヒープタプルデスクリプタの取得

*l.*1367

インデックスするタプルに関する情報（ヒープタプルデスクリプタ）を、RelationGetDescr() マクロを用いて引数 heapRelation から取得し、変数 heapDescriptor に格納する。

5.2.3 テンポラリメモリコンテキストの構築

*l.*1369 ~ 1374

部分インデックスの述語評価と、関数インデックスの関数評価に用いるメモリコンテキストを用意する。

部分インデックスとは、テーブルの部分集合に構築されるインデックスのこと。ある条件式（述語）を満たすテーブル行のエントリのみをインデックスする場合に用いられる。

関数インデックスとは、1 つのテーブルの 1 つ以上の列に適用された関数の結果に定義したインデックスである。関数の呼び出し結果からなるデータに高速にアクセスするために用いられる。

具体的なメモリコンテキストの構築は、エクゼキュータ呼び出しのために必要な作業領域となる EState 型の変数 estate を CreateExecutprState() 関数で初期化することで行う。そして、作業領域のうち述語評価・関数評価に用いるメモリコンテキストへのポインタを GetPerTupleExprContext() マクロで取得し、変数 econtext に格納する。

5.2.4 評価関数用の変数の初期化

*l.*1376 ~ 1401

対象となるインデックスが部分インデックスか、もしくは関数インデックスかを確認する。

もしいずれかであれば、述語・関数評価に用いる関数（ExecQual() 関数）用のタプルテーブル変数 TupleTable とタプルスロット変数 slot を初期化する。そして、メモリコンテキスト変数 econtext の中でスキャン用タプルスロットを格納する econtext->ecxt_scantuple に scan を代入し、評価作業領域変数 estate を関数 ExecPrepareExpr() で評価式リスト predicate に加える。

そうでなければ、各変数にヌルポインタを代入する。

5.2.5 各種初期化

*l.*1403 ~ 1424

各種の初期化（5.2.5.1節〜5.2.5.3節）を行う。

*l.*1403 ~ 1417 **5.2.5.1 スナップショットの種類を選択** ブートストラップ時かどうかで処理が変わる。

もしブートストラップ時であればスナップショットとして SnapshotNow、そうでなければ SnapshotAny を用いる。SnapshotAny を用いる時には、一番古いトランザクション ID を変数 OldestXmin に格納する。

*l.*1419 ~ 1422 **5.2.5.2 ヒープスキャンデスクリプタの初期化** ヒープスキャンデスクリプタ変数 scan を heap_beginscan() 関数で初期化する。

*l.*1424 **5.2.5.3 ヒープタプル総数カウンタの初期化** 処理したヒープタプルの総数をカウントする変数 reltuples を初期化する（0 を代入する）。

5.2.6 ベースリレーションの全てのタプルをスキャンし処理

*l.*1426 ~ 1569

ヒープからタプルを取り出せなくなるまで、以下の処理（5.2.6.1 節〜5.2.6.8節）を繰り返し行う。

*l.*1429 **5.2.6.1 ヒープからタプルを取り出す** ヒープからタプルを取り出し、変数 heapTuple に格納する。

*l.*1433 **5.2.6.2 インタラプト命令が来ているかどうかチェック** 来ていたらインタラプト。

*l.*1435 **5.2.6.3 スナップショットが SnapshotAny かチェック** スナップショットの種類を確認する（つまり、ブートストラップ時かどうかで処理が変わる）。

(a) SnapshotAny **の場合**

*l.*1441 ~ 1450 もし SnapshotAny なら、HeapTupleSatisfiesVacuum() 関数を用いてヒープタプルの他のトランザクションから見た状態を測定する。この **(a)** の処理の間、現在のヒープスキャンバッファは共有ロックされる。また、この時点でのヒープタプルの状態フラグ heapTuple->t_data->t_infomask を変数 sv_infomask に保存しておく。HeapTupleSatisfiesVacuum() 関数がフラグを更新するかもしれないためである。

*l.*1452 ~ 1509 ヒープタプルの状態に応じて、フラグを以下のように設定する。

表 3: ヒープタプルの状態と対応するフラグ

タプル状態	indexIt	tupleIsAlive
HEAPTUPLE_DEAD	FALSE	FALSE
HEAPTUPLE_LIVE	TRUE	TRUE
HEAPTUPLE_RECENTLY_DEAD	TRUE	FALSE
HEAPTUPLE_INSERT_IN_PROGRES	TRUE	TRUE
HEAPTUPLE_DELETE_IN_PROGRES	TRUE	FALSE
それ以外	FALSE	FALSE

表3のうち、HEAPTUPLE_INSERT_IN_PROGRES の場合と HEAPTUPLE_DELETE_IN_PROGRES の場合は、Insert もしくは Delete が、この処理自身のトランザクションによるものかどうかチェックし、そうでなければエラーとなる。ただし、この Insert・Delete が、システムカタログの再インデックスの場合を除く。

タプル状態が「それ以外」の場合も、エラーとなる。

*l.*1511 ~ 1513 ここで、以前保存しておいた、ヒープタプルの状態フラグ sv_infomask と、現在のフラグ heapTuple->t_data->t_infomask を比較し、HeapTupleSatisfiesVacuum() 関数がフラグを更新したかどうかを確認する。もし更新されていたら、SetBufferCommitInfoNeedsSave() 関数を用いて、ヒープスキャンバッファscan->rs_cbuf がダーティであるとマークする。

*l.*1515 そして、現在のヒープスキャンバッファに 対する共有ロックを解放する。

*l.*1517 ~ 1518 indexIt フラグが FALSE の場合、ループの先頭（5.2.6.1 節）に戻って次のタプルに処理を移す。

(b) SnapshotAny **でない場合**

*l.*1520 ~ 1524 もし SnapshotAny でなければ、heap_getnext() 関数がタプルの状態を調べているので、tupleIsAlive フラグを TRUE にする。

*l.*1526 **5.2.6.4 ヒープタプル総数カウンタをインクリメント** 変数 reltuples に 1 を加える。

*l.*1528 **5.2.6.5 式評価用のメモリコンテキストを開放** 変数 econtext の中の、式評価用のメモリコンテキスト econtext->ecxt_per_tuple_memory を MemoryContextReset() 関数で開放する。

*l.*1534 ~ 1546 **5.2.6.6 部分インデックスの場合、述語を満たさないタプルを捨てる** また、ヒープタプルの状態が「最近削除された（recently-dead）」なものも捨てる。なぜなら、部分インデックスの場合、VACUUM がタプルカウント不一致に関して関知しないためである。

述語の評価には ExecQual() 関数を用いる。

*l.*1548 ~ 1558 **5.2.6.7 新しいインデックスタプルの構築** FormIndexDatum() 関数を用いて、attdata 配列と nulls 配列を新しいインデックスタプルのために構築・初期化する。attdata 配列は、タプルの属性を示す配列である。nulls 配列は、タプルが NULL かどうかを示す配列である。

インデックスが関数インデックスの場合、これは関数の評価としても作用する[6]。

*l.*1560 ~ 1568

5.2.6.8　アクセスメソッドが指定したコールバック関数を呼ぶ　ハッシュインデックスの場合、hashbuildCallback() 関数をコールバックする。

このコールバック関数によって、実際にハッシュインデックスへの格納が行われる。このコールバック関数の詳しい動作については後述（5.3節）する。

コールバック関数から戻ってきたら、ループの先頭（5.2.6.1節）に戻る。

5.2.7　各種後処理

*l.*1571 ~ 1580

後処理として、下記のように、各種テンポラリ変数の後始末を行う。

1. heap_endscan() 関数で、ヒープスキャンデスクリプタ変数 scan を後始末して開放する。
2. もし部分インデックス・関数インデックスだったら、ExecDropTupleTable() 関数で、評価用に作成したタプルテーブル変数 TupleTable を後始末して開放する。
3. FreeExecutorState() 関数で、econtext 変数を後始末して開放する。
4. 構築したインデックスに関する情報を格納する indexInfo のうち、関数評価や述語評価のための情報を格納している indexInfo->ii_ExpressionState と indexInfo->ii_PredicateState を NIL 値にする。

5.2.8　ヒープタプルの総数を出力

*l.*1582

ヒープタプルの総数 reltuples を戻り値として返す。

5.3　hashbuildCallback() 関数の動作

backend/
access/hash/
hash.c
*l.*104 ~ 142

タプル毎に IndexBuildHeapScan() 関数からコールバック呼び出しを受ける hashbuildCallback() 関数について説明する。

hashbuildCallback() 関数は、以下の引数を持つ。

index（**Relation 型**）構築したインデックスを入れるインデックスリレーション

htup（**HeapTuple 型**）インデックスするタプル

*attdata（**Datum へのポインタ型**）タプルの属性を示す配列

*nulls（**char へのポインタ型**）タプルが NULL かどうかを示す配列

tupleIsAlive（**bool 型**）タプルの状態フラグ

*state（**void 型**）コールバック関数の結果を返すための変数へのポインタ

この関数の戻り値はない。

この関数の動作は大きく次のように構成されている。

1. インデックスタプルの作成
2. インデックスタプルをヒープタプルに向ける
3. インデックスタプルが NULL 値を持っているか確認

[6]初期化に用いる FormIndexDatum() 関数の中で、関数の評価もおこなっている。

4. インデックスタプルのコピーを作成
5. インデックスにコピーしたインデックスタプルを挿入
6. 各種後処理

以下、その詳細について説明する。

5.3.1 インデックスタプルの作成

l.120 ~ 121

インデックスするタプルを処理するテンポラリ変数 itup を index_formtuple() 関数で作成する。

5.3.2 インデックスタプルをヒープタプルに向ける

l.122

具体的には、ヒープタプルの ID が変数 htup->t_self に入っているので、それを変数 itup->t_tid に入れて参照できるようにする。

5.3.3 インデックスタプルが NULL 値を持っているか確認

l.124 ~ 129

ハッシュインデックスでは NULL 値をインデックスしないので、変数 itup を確認し、NULL だったら変数 itup を開放し呼び出し元に戻る。

ハッシュインデックスが NULL 値をインデックスしない理由は次のとおりである。

ハッシュインデックスのスキャンでは、ビット毎の類似性だけが演算される。よってサポートする演算子は “=” だけである。関係代数において、A または B のどちらかが NULL 値の時、$A = B$ は NULL 値を返す。つまり、NULL 値がインデックスに入っていても、TRUE になることはありえない。ゆえに、ハッシュインデックスは NULL 値をインデックスしない。

5.3.4 インデックスタプルのコピーを作成

l.131

_hash_formitem() 関数を用いて、インデックスタプルのコピーを作成し、変数 hitem に格納する。

5.3.5 インデックスにコピーしたインデックスタプルを挿入

l.133

指定したインデックスリレーション index に対し、hitem を _hash_doinsert() 関数で挿入する。
この関数の詳しい動作については後述（5.4節）する。

5.3.6 各種後処理

l.135 ~ 141

インデックスしたタプルの数を数えるカウンタ buildstate->indtuples に 1 を加え、各種テンポラリ変数を開放して終了する。

5.4 _hash_doinsert() 関数の動作

backend/
access/hash/
hashinsert.c
l.26 ~ 199

テーブルへの単一 HashItem 挿入を処理する _hash_doinsert() 関数について説明する。
_hash_doinsert() 関数は、以下の引数を持つ。

rel（**Relation 型**）構築したインデックスを入れるインデックスリレーション

hitem（**HashItem 型**）インデックスするタプルのコピー

この関数の戻り値は、インデックスにタプルを挿入した結果（InsertIndexResult 型）である。
この関数の動作は大きく次のように構成されている。

1. インデックスメタページの取得
2. インデックス検索用スキャンキーの構築
3. インデックスを検索し挿入するバッファを取得
4. ロック交換
5. インデックスにタプルを挿入
6. スキャンキーの開放
7. 戻り値の出力

以下、その詳細について説明する。

5.4.1 ハッシュキーの計算

l.53 ~ 62

挿入するデータのハッシュキーを計算する。

まず、これからインデックスするタプル（hitem->hash_itup）へのポインタを変数 itup に入れる。また、インデックスリレーションの属性のうち、インデックスキーの数（rel->rd_del->relanatts）の数が 1 以外でないかを確認する。なぜなら、ハッシュインデックスはインデックスキーを 1 つしかサポートしていないからである。

次に、タプルから index_getattr() 関数を用いてデータそのものを取り出し、変数 datum に格納する。そして、_hash_datum2hashkey() 関数を用いて datum のハッシュキーを計算し、変数 hashkey に格納する。
_hash_datum2hashkey() 関数は、datum のデータ型に応じた適切なハッシュ値計算ルーチンを呼び出す機能を持っている。

5.4.2 アイテムサイズの計算

l.64 ~ 69

挿入するデータのサイズをここで計算し、変数 itemsz に格納する。その際、実際のサイズそのものではなく、型に合わせて切り上げをする。

5.4.3 インデックスメタページに共有ロックをかける

l.71 ~ 75

挿入対象となるバケツを安全に探す処理ができるように、インデックスリレーション rel のメタページに対し、_hash_getlock() 関数で共有ロックをかける[7]。こうすることにより、他のトランザクションによるバケツページの分割が抑止できる。

5.4.4 インデックスメタページの取得

l.77 ~ 80

まずインデックスリレーションのメタページ用のバッファを_hash_getbuf() 関数で取得し、変数 metabuf に入れる。この時このバッファにリードロックがかかる。前の節で行ったロックはロックマネージャによる通常ロック（“heavyweight” locks）で、ここで行うロックはバッファに対する軽量ロック（LWLocks）という違いがある。詳しくは README を参照されたい。

次に、metabuf に関連するメタページを BufferGetPage() 関数で取得し、変数 metap に入れる。

取得したメタページは、_hash_checkpage() 関数でサニティチェックをかける。

[7] メタページに対するロックを分割ロック（split lock または split-control lock）とソース中では呼んでいる。処理対象のバケツを探すだけのときは共有ロックの分割ロック、実際にバケツを分割する時は排他ロックの分割ロックが必要になると README に記述がある。

5.4.5 アイテムがハッシュページに入るかどうかチェック

*l.*82 ~ 92

ハッシュページに入る最大のサイズを HashMaxItem() マクロで確認し、先ほど取得した変数 itemsz と比較する。もし itemsz の方が大きい場合はエラーを出力する。

5.4.6 挿入先バケツ番号と対応するブロック番号を計算

*l.*94 ~ 102

_hash_hashkey2bucket() 関数で、ハッシュキー hashkey から挿入先バケツ番号を計算し変数 bucket に格納する。そして、バケツ番号に対応するブロック番号を BUCKET_TO_BLKNO() マクロで求め、変数 blkno に格納する。

5.4.7 インデックスメタページへのリードロックを解放

*l.*104 ~ 105

_hash_chgbufaccess() 関数で、メタページを取得したときにかけたリードロックを解放する。ただし、後ほど必要となるのでピンは保持したままにしておく。

5.4.8 バケツページに共有ロックをかける

*l.*107 ~ 110

挿入対象バケツに対して、_hash_getlock() 関数で共有ロックをかける。

5.4.9 インデックスメタページへの共有ロックを解放

*l.*112

_hash_droplock() 関数で、メタページにかけた共有ロックを解放する。

5.4.10 タプルを挿入するバッファを取得

*l.*114 ~ 119

タプルを挿入するバケツのバッファを _hash_getbuf() 関数で取得し、変数 buf に入れる。この時このバッファにライトロック（ここで行うロックはバッファに対する軽量ロック (LWLocks)）がかかる。次に、buf に関連するページを BufferGetPage() 関数で取得し、変数 page に入れる。

取得したページは、_hash_checkpage() 関数でサニティチェックをかける。

ページの中に含まれるスペシャルデータ（アクセスメソッド特有のデータ。3.2.1.3節を参照のこと）へのポインタを PagegetSpecialPointer() マクロで取り出し変数 pageopaque に入れる。スペシャルデータのうち、このページが属するバケツ番号 pageopaque->hasho_bucket を変数 bucket と比較し、正しいバケツ番号のページを取得したか確認する。

5.4.11 対象ページにタプルを挿入する余裕があるか確認

*l.*122

挿入するページの空き容量を PageGetFreeSpace() 関数で取得し、それを先ほど求めた挿入するタプルのサイズ変数 itemsz と比較する。

もし itemsz のほうが小さい場合は、次の処理（5.4.12節）へ移る。

逆に itemsz のほうが大きい場合、このページには空き容量が残っていないことになるので、このページにオー
*l.*127 バーフローページのブロック番号を取得し、変数 nextblkno に入れる。

*l.*129 ここで変数 nextblkno が有効な値かどうか、つまりオーバーフローページが存在するかどうかを BlockNumberIsValid() マクロで確認する。

もしオーバーフローページが存在するなら、このページが持つスペシャルデータの中に、次のオーバーフロー
*l.*131 ~ 137 ブロック番号が記載されているので、それをたぐる。具体的には、以下のような処理をする。

1. 現在のバッファbuf を_hash_relbuf() 関数でライトロックから開放する。

2. _hash_getbuf() 関数でオーバーフローバッファ（ブロック番号 nextblkno）を取得しライトロックをかけ、変数 buf にいれる。

3. 変数 buf（オーバーフローバッファが入っている）から、ページを BufferGetPage() 関数で取得し、変数 page に入れる。

逆にオーバーフローページが存在しない場合、新しいオーバーフローページを割り当てる。具体的には、以下の
*l.*141 ~ 158 ような処理をする。

1. _hash_chgbufaccess() 関数で、現在のページに対応するバッファのライトロックを解放する。この時、バッファへの変更は反映されない[8]。

2. _hash_addovflpage() 関数で、オーバーフローバッファを現在変数 buf が指し示しているバッファに付け加える。そのバッファを変数 buf にいれる。

3. 変数 buf（オーバーフローバッファが入っている）から、ページを BufferGetPage() 関数で取得し、変数 page に入れる。

4. 新しいページの空き容量を PageGetFreeSpace() 関数で取得し、それを先ほど求めた挿入するタプルのサイズ変数 itemsz と比較する。

*l.*156 ~ 158 以上のような処理でオーバーフローページを取得し、そのページを_hash_checkpage() 関数でサニティチェックをかける。そしてページの中に含まれるスペシャルデータへのポインタを PagegetSpecialPointer() マクロで取り出し変数 pageopaque に入れる。これで、オーバーフローページが新しい挿入先ページとして設定される。

ここでこの処理の先頭に戻り、もう一度ページの空き容量と挿入するタプルサイズを比較する。そして、タプルを挿入することができるオーバーフローページを見つけられるまでループする。

5.4.12 タプルの挿入

*l.*161 ~ 169

実際にタプルを挿入する。

まず_hash_pgaddtup() 関数で、インデックスリレーション rel の中の、先ほど求めた挿入先ページを含むバッファbuf にインデックスするタプル hitem を付加する。この関数は、戻り値としてタプルを挿入したページのオフセット値を返すので、それを変数 itup_off に代入する。

次に BufferGetBlockNumber() 関数でバッファbuf に割り当てられたブロック番号を求め、変数 itup_blkno に代入する。

以上で処理したバッファを_hash_wrtbuf() 関数で書き出し、ライトロックを解放する。そして、_hash_dropscan() 関数でバケツへの共有ロックも解放する。

5.4.13 ページの拡張

*l.*171 ~ 190

バケツページの追加（ページ拡張）を必要に応じて行う。

*l.*175 ~ 177 まず、メタページに記録している格納したタプル数をインクリメントする。_hash_chgbufaccess() 関数で、メタページに対応するバッファ（metabuf）のライトロックを取得する。そして、変数 metap->hashm_ntuples の数を 1 増やす。

*l.*180 ~ 181 次に、格納したタプルの数とバケツの総数の比が一定の値（変数 metap->hashm_ffactor）を超えていたら、バケツページの追加をする必要があるので、変数 do_expand を真にする。

*l.*184 計算が終了したら、_hash_chgbufaccess() 関数で、メタページに対応するバッファのライトロックを解放し書き出す。ただしピンは保持したまま。

*l.*187 ~ 188 そして、変数 do_expand が真のときは、_hash_expandtable() 関数でバケツを追加する。

[8] ここの_hash_chgbufaccess() 関数で、バッファのリードロックを解放するように引数が指定されている。実際にはバッファにはライトロックがかかっているのだが、わざとリードロック解放の指定をすることによって、バッファへの変更を反映せずにロックを解放することができる。関数のコメントを参照のこと。

l.191 以上すべて終了したところで、_hash_dropbuf() 関数でメタページに対応するバッファのピンを抜く。

5.4.14 各種後処理

l.193 ~ 198

ここで、タプルを挿入した結果を呼び出し元に返すためのメモリを確保し、変数 res にポインタを代入する。
次に ItemPointerSet() 関数で、結果としてタプルを挿入したページのディスク上の位置を示す変数 itup_blkno と、そのページ内でのタプル位置のオフセット変数 itup_off を変数 res の中に格納する。
最後に、戻り値として変数 res を返す。

6 ハッシュインデックスにタプルを挿入

ハッシュインデックスに指定したタプルを挿入する処理は、hashinsert() 関数によって行われる。
関数の内部呼び出し構成は図9のようになっている。

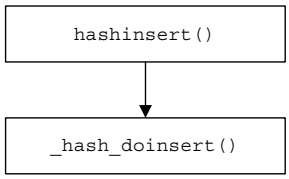

図 9: ハッシュインデックスへのタプル挿入関数の構成図

このうち、hashinsert() 関数が呼び出す_hash_doinsert() 関数については、hashbuild() 関数で呼び出しているものと同じである。

6.1 hashinsert() 関数の動作

backend/
access/hash/
hash.c
l.144 ~ 194

hashinsert() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

rel（**Relation 型**）構築したインデックスを入れるインデックスリレーション

*datum（**Datum へのポインタ型**）インデックスするタプルの属性を示す配列

*nulls（**char へのポインタ型**）タプルが NULL かどうかを示す配列

ht_ctid（**ItemPointer 型**）インデックスするタプルの ID

この関数は、Postgres Function Manager 互換の関数作成用マクロにより、インデックスにタプルを挿入した結果（InsertIndexResult 型）へのポインタを戻り値として呼び出し元に返す。
この関数の動作は大きく次のように構成されている。その構成は、5.3節で説明した hashbuildCallback() 関数によく似ている。

1. インデックスタプルの作成
2. インデックスタプルを指定したタプルに向ける
3. インデックスタプルが NULL 値を持っているか確認
4. インデックスタプルのコピーを作成
5. インデックスにコピーしたインデックスタプルを挿入
6. 各種後処理

以下、その詳細について説明する。

6.1.1 インデックスタプルの作成

l.167 ~ 168

インデックスするタプルを処理するテンポラリ変数 itup を index_formtuple() 関数で作成する。

6.1.2 インデックスタプルを指定したタプルに向ける

l.169

具体的には、指定したタプルの ID が変数*ht_ctid に入っているので、それを変数 itup->t_tid に入れて参照できるようにする。

6.1.3 インデックスタプルが NULL 値を持っているか確認

l.171 ~ 184

ハッシュインデックスでは NULL 値をインデックスしないので、変数 itup を確認し、NULL だったら変数 itup を開放する。そして、戻り値として NULL を定義し、呼び出し元に戻る。

6.1.4 インデックスタプルのコピーを作成

l.186

_hash_formitem() 関数を用いて、インデックスタプルのコピーを作成し、変数 hitem に格納する。

6.1.5 インデックスにコピーしたインデックスタプルを挿入

l.188

指定したインデックスリレーション index に対し、hitem を _hash_doinsert() 関数で挿入する。
この関数の詳しい動作については前述（5.4節）のとおりである。
挿入した結果を変数 res に代入する。

6.1.6 各種後処理

l.190 ~ 193

各種テンポラリ変数を開放し、_hash_doinsert() 関数の結果を格納した変数 res を戻り値として返す。

7 ハッシュインデックスから次の有効なタプルを得る

ハッシュインデックスから次の有効なタプルを得る処理は、hashgettuple() 関数によって行われる。
関数の内部呼び出し構成は図10のようになっている。

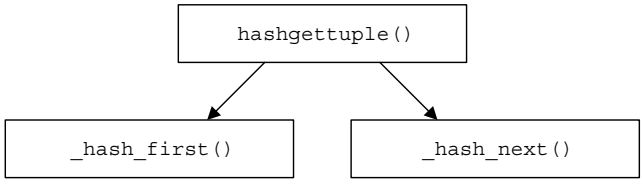

図 10: ハッシュインデックスから次の有効なタプルを得る関数の構成図

7.1 hashgettuple() 関数の動作

backend/
access/hash/
hash.c
l.197 ~ 274

hashgettuple() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

scan（**IndexScanDesc 型**）インデックススキャンに関する情報

dir（**ScanDirection 型**）インデックススキャンの方向

この関数は、Postgres Function Manager 互換の関数作成用マクロにより、インデックスをスキャンした結果（bool 型）を戻り値として呼び出し元に返す。
この関数の動作は大きく次のように構成されている。

1. 現在スキャン中のバッファを確認
2. カレントバッファへのリードロックを取得
3. インデックススキャンが初期化されているかどうか確認
   - スキャンが初期化済みならばそのスキャンを続行し次のタプルを得る
   - スキャンが初期化済みでなければ初期化し最初のタプルを得る
4. 必要に応じ、削除済みタプルを無視する
5. カレントバッファへのリードロックを解放
6. 各種後処理

以下、その詳細について説明する。

7.1.1 現在スキャン中のバッファを確認

*l.*205 まず、どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque）、変数 so に格納する。

7.1.2 カレントバッファへのリードロックを取得

*l.*211 ~ 216 現在スキャン中のバッファは、ピンは保持しているものの、ハッシュアクセスメソッドの外ではロックしていない。そのため、まずスキャン中のバッファ（so->hashso_curbuf）が正当な値を示しているかどうか確認し、正当だった場合、_hash_chgbufaccess() 関数で、バッファのリードロックを再要求する。

7.1.3 インデックススキャンが初期化されているかどうか確認

*l.*223 変数 scan に記録されている、現在のスキャン済タプルへのポインタ（scan->currentItemData）が正当な値を示しているかどうか確認する。もしこれが正当な値でない場合は、まだスキャンが初期化されていないことを意味する。
もしスキャンが初期化されていた場合は7.1.3.1節に進み、初期化されていなかった場合は7.1.3.2節に進む。

*l.*225 ~ 249 **7.1.3.1 スキャンが初期化されていた場合** スキャンがすでに初期化されていた場合は、まず、現在のスキャン済タプルを削除するべきかどうか（変数 scan->kill_prior_tuple が TRUE かどうか）を確認する。
もしそうなら、変数 offnum にスキャン済タプルのオフセット番号を代入し、変数 page に現在スキャンしているバッファのページを入れる。そして、ページ page 内のオフセット番号 offnum の位置にあるタプルに、削除されることになっていることを示すフラグを立てる。
そして、_hash_next() 関数で、このスキャンの次のタプルを得て、その結果を変数 res に代入する。この関数の詳しい動作については後述（7.3節）する。

*l.*251 ~ 252 **7.1.3.2 スキャンが初期化されていなかった場合** スキャンが初期化されていない場合は、 _hash_first() 関数で、スキャンを初期化し、スキャンの一番最初のタプルを得て、その結果を変数 res に代入する。この関数の詳しい動作については後述（7.2節）する。

7.1.4 必要に応じ、削除済みタプルを無視する

l.254 ~ 267

ここで、スキャンが削除済タプルを無視すべきかどうか（変数 scan->ignore_killed_tuples が TRUE かどうか）を確認する。

もしそうなら、変数 res が TRUE の間、以下の処理（7.1.4.1節〜7.1.4.3節）を繰り返し、削除済タプルでないものを探す。

l.261 ~ 262

7.1.4.1 スキャンしたタプルの位置を調べる 変数 offnum にスキャン済タプルのオフセット番号を代入し、変数 page に現在スキャンしているバッファのページを入れる。

l.263 ~ 264

7.1.4.2 タプルの削除フラグを確認 ページ page 内のオフセット番号 offnum の位置にあるタプルに、削除されることになっていることを示すフラグが立っているかどうかを確認する。もし立っていなければ、このタプルで問題ないので、このループから脱出する。

l.265

7.1.4.3 次のタプルをスキャンする 削除フラグが立っているようであれば、_hash_next() 関数で、さらに次のタプルを得て、その結果を変数 res に代入する。

7.1.5 カレントバッファへのリードロックを解放

l.269 ~ 271

_hash_chgbufaccess() 関数で、スキャン中のバッファへのリードロックを解放する。ただし、ピンは保持したままにしておく。

7.1.6 各種後処理

l.273

_hash_next() 関数または _hash_first() 関数の結果を格納した変数 res を戻り値として返す。有効なタプルは、変数 scan を通じて呼び出し元に渡されることになる。

7.2 _hash_first() 関数の動作

backend/
access/hash/
hashsearch.c
l.106 ~ 223

スキャンデスクリプタに関連する制限（スキャンキー）を満たす最初のタプルをインデックスの中から取得する _hash_first() 関数について説明する。

_hash_first() 関数は、以下の引数を持つ。

scan（**IndexScanDesc 型**）インデックススキャンに関する情報

dir（**ScanDirection 型**）インデックススキャンの方向

この関数の戻り値は、インデックスをスキャンした結果（bool 型）である。

この関数の動作は大きく次のように構成されている。

1. 各種変数初期化
2. カレントインデックスポインタの初期化
3. スキャンキーの数・フラグを確認
4. ハッシュキーの計算
5. バケツ分割抑止ロックの取得
6. インデックスメタページの取得

7. スキャン対象バケツの計算
8. 対象バケツへの共有ロックの取得と分割抑止ロックの解放
9. スキャン中バッファ情報の更新
10. スキャン対象バケツのバッファを取得
11. バッファを確認し、対象となるタプルを探す
12. 戻り値の出力

以下、その詳細について説明する。

7.2.1 各種変数初期化

*l.*118 ~ 119

各種変数を初期化する。

スキャン対象のインデックスリレーションを変数 scan から取り出し（ scan->indexRelation)、変数 rel に格納する。

どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque)、変数 so に格納する。

7.2.2 カレントインデックスポインタの初期化

*l.*133 ~ 134

カレントインデックスポインタ（変数 scan->currentItemDataopaque）のアドレスを変数 current に格納する。

取得したポインタは、ItemPointerSetInvalid() マクロで初期化する（Invarid 値にする)。

7.2.3 スキャンキーの数・フラグを確認

*l.*136 ~ 152

*l.*136 ~ 145 ここでスキャンキーの数（scan->numberOfKeys）が 1 以上であるかどうか確認する。もしこれが 0 であれば、エラーコードを出力し処理を中断する。

スキャンキーが 1 つも設定されていないということは、どんなタプルであってもスキャンキーにヒットするという意味になるので、インデックス全体をスキャンすることになる。7.4 以前では、そのようなスキャンを認めるような実装になっていたが、7.4 よりハッシュインデックスに対するロック方法がより厳格なものに変更されたため、インデックス全体のスキャンを許可すると多数の問題を引き起こす可能性が出てきた。そのため、この機能は使用できないように変更された。

また、スキャンキーの内容についてのフラグ（scan->keyData[0].sk_flags）を調べ、スキャンキーが NULL

*l.*151 ~ 152 値かどうかも確認する。もし NULL 値なら、どんなタプルもスキャンにヒットしないという意味になるので、インデックススキャンの結果は失敗だったとして、この関数の戻り値を FALSE にして呼び出し元に戻る。

7.2.4 ハッシュキーの計算

*l.*154 ~ 158

スキャンキーが正当なものだと確認できたので、ここで_hash_datum2hashkey() 関数でスキャンキーからハッシュキーを計算し、変数 hashkey に格納する。この関数の中で、ハッシュ値を計算するサポートルーチンを呼び出している。

7.2.5 バケツ分割抑止ロックの取得

*l.*160 ~ 164

_hash_getlock() 関数で、ページブロック 0（メタページ）への通常ロックを ShareLock モードで取得する。

これは、スキャン対象バケツの計算を正当なものとするため、計算中に他のトランザクションによってバケツを分割したりされないようにするためのロックである。

7.2.6 インデックスメタページの取得

*l.*166 ~ 169

インデックスリレーションのメタページ用のバッファを_hash_getbuf() 関数で取得し、変数 metabuf に入れる。
次に、metabuf に関連するメタページを BufferGetPage() 関数で取得し、変数 metap に入れる。
取得したメタページは、_hash_checkpage() 関数でサニティチェックをかける。

7.2.7 スキャン対象バケツの計算

*l.*171 ~ 182

先ほど計算したハッシュキー hashkey と、メタページから得た情報（使用中バケツの最大値、テーブル全体に対する剰余のマスク、テーブル下半分に対する剰余のマスク）を用いて、_hash_hashkey2bucket() 関数でスキャンキーを満たすバケツを計算し、変数 bucket に入れる。
得られたバケツ番号から、BUCKET_TO_BLKNO() マクロでブロック番号を求め、変数 blkno に入れる。
最後に、メタページの情報はもういらないので、_hash_relbuf() 関数でメタバッファを解放する。

7.2.8 対象バケツへの共有ロックの取得と分割抑止ロックの解放

*l.*184 ~ 189

_hash_getlock() 関数で、スキャン対象バケツブロックへの通常ロックを ShareLock モードで取得する。
取得後、先ほど取得したページブロック 0 への通常ロック（分割抑止ロック）を_hash_droplock() 関数で解放する。

7.2.9 スキャン中バッファ情報の更新

*l.*191 ~ 194

どのバッファがスキャンされているかという情報についての変数 so のうち、以下の 3 つを更新する。

- 現在スキャン中のバッファ（so->hashso_bucket）に変数 bucket を代入
- スキャン中のバッファが正当かどうかの値（so->hashso_bucket_valid）を TRUE に
- スキャン中のバケツに通常ロックを ShareLock モードで取得している場合に、そのブロック番号を記録する変数（so->hashso_bucket_blkno）に変数 blkno を代入

7.2.10 スキャン対象バケツのバッファを取得

*l.*196 ~ 201

スキャン対象バケツ用のバッファを_hash_getbuf() 関数で取得し、変数 buf に入れる。
次に、buf に関連するページを BufferGetPage() 関数で取得し、変数 page に入れる。
取得したページは、_hash_checkpage() 関数でサニティチェックをかける。
取得したページの中に含まれるスペシャルデータ（アクセスメソッド特有のデータ）へのポインタを PagegetSpecialPointer() マクロで取り出し変数 opaque に入れ、このページのバケツ番号（opaque->hasho_bucket）が変数 bucket と符合するかどうか確認する。

7.2.11 バッファを確認し、対象となるタプルを探す

*l.*203 ~ 212

検索対象となるタプルは、ここまでで取得したバケツを先頭に、オーバーフローバケツが連なったバケツチェーンの中のどこかにあるので、このバケツチェーンをたぐって探す。

インデックススキャンの方向（変数 dir に入っている）が後方の場合は、オーバーフローバケツを _hash_readnext() 関数を用いてたぐり、最後のバケツを探す。

そして、_hash_step() 関数で、バケツチェーンをスキャン方向 dir に沿ってたぐっていき、その中から対象となるタプルを探す。探すのに失敗した場合（関数の戻り値が FALSE だった場合）は、戻り値を FALSE として呼び出し元に戻る。成功した場合は次の処理へ移る。

7.2.12 戻り値の出力

*l.*214 ~ 222

変数 scan を経由して、呼び出し元にタプルを返す。そのために以下の処理をする。

1. カレントインデックスポインタ current のディスクアイテムポインタのオフセット番号を ItemPointerGetOffsetNumber() マクロで取得し、変数 offnum に入れる。
2. buf に関連するページを BufferGetPage() 関数で取得し、変数 page に入れる。
3. 取得したページを _hash_checkpage() 関数でサニティチェックをかける。
4. ページ page のオフセット番号 offnum からタプルを取得し、変数 hitem に入れる。
5. 取得したタプルの情報が格納されたアドレスを変数 itup に入れる。
6. 取得したタプルの情報のうち、タプル ID（iutp->t_tid）を変数 scan->xs_ctup.t_self に入れる。

最後に、この関数の結果として、戻り値を TRUE にして呼び出し元に戻る。

7.3 _hash_next() 関数の動作

backend/
access/hash/
hashsearch.c
*l.*21 ~ 62

スキャンデスクリプタに関連する制限を満たす次のタプルをインデックスの中から取得する _hash_next() 関数について説明する。

_hash_next() 関数は、以下の引数を持つ。

scan（**IndexScanDesc 型**）インデックススキャンに関する情報

dir（**ScanDirection 型**）インデックススキャンの方向

この関数の戻り値は、インデックスをスキャンした結果（bool 型）である。

この関数の動作は大きく次のように構成されている。その構成は、7.2節で説明した _hash_first() 関数から初期化部分を抜いたようなものになっている。

1. 各種変数初期化
2. 現在のバッファを取得
3. 対象となるタプルを探す
4. 戻り値の出力

以下、その詳細について説明する。

7.3.1 各種変数初期化

*l.*33 ~ 34

各種変数を初期化する。

スキャン対象のインデックスリレーションを変数 scan から取り出し（ scan->indexRelation)、変数 rel に格納する。

どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque)、変数 so に格納する。

7.3.2 現在のバッファを取得

l.42 ~ 44

現在スキャン中のバッファを変数 so から取り出し（ so->hashso_curbuf）、変数 buf に入れる。取得したバッファはサニティチェックする。

7.3.3 対象となるタプルを探す

l.46 ~ 50

_hash_step() 関数で、バケツの中から次の対象となるタプルを探す。探すのに失敗した場合（関数の戻り値が FALSE だった場合）は、戻り値を FALSE として呼び出し元に戻る。

7.3.4 戻り値の出力

l.52 ~ 61

変数 scan を経由して、呼び出し元にタプルを返す。そのために以下の処理をする。

1. カレントインデックスのポインタ（変数 scan->currentItemData）のアドレスを変数 current に格納する。
2. current のディスクアイテムポインタのオフセット番号をマクロ ItemPointerGetOffsetNumber で取得し、変数 offnum に入れる。
3. buf に関連するページを BufferGetPage() 関数で取得し、変数 page に入れる。
4. 取得したページを_hash_checkpage() 関数でサニティチェックをかける。
5. ページ page のオフセット番号 offnum からタプルを取得し、変数 hitem に入れる。
6. 取得したタプルの情報が格納されたアドレスを変数 itup に入れる。
7. 取得したタプルの情報のうち、タプル ID（iutp->t_tid）を変数 scan->xs_ctup.t_self に入れる。

最後に、この関数の結果として、戻り値を TRUE にして呼び出し元に戻る。

8 ハッシュインデックスの新規スキャン開始

ハッシュインデックスの新規スキャン開始の手続き（インデックススキャンデスクリプタの初期化処理）は、hashbeginscan() 関数によって行われる。

関数の内部呼び出し構成は図 11のようになっている。

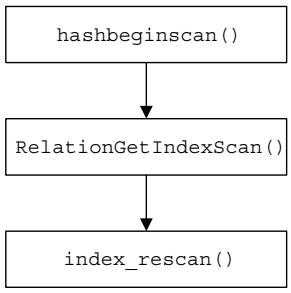

図 11: ハッシュインデックスの新規スキャン開始手続き関数の構成図

8.1 hashbeginscan() 関数の動作

backend/
access/hash/
hash.c
l.277 ~ 300

hashbeginscan() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

`rel`（**Relation 型**）スキャンするインデックスリレーション

`keysz`（**int 型**）スキャンキーの数

`scankey`（**ScanKey 型**）インデックススキャンを限定するスキャンキーの配列

この関数は、Postgres Function Manager 互換の関数作成用マクロにより、初期化設定したインデックススキャンデスクリプタ（IndexScanDesc 型）へのポインタを戻り値として呼び出し元に返す。
この関数の動作は大きく次のように構成されている。

1. インデックススキャンデスクリプタの初期化
2. スキャン中のバッファを記録する変数の初期化
3. インデックススキャンの登録
4. 戻り値の出力

以下、その詳細について説明する。

8.1.1 インデックススキャンデスクリプタの初期化

*l.*289

インデックススキャンデスクリプタ `scan`（インデックススキャンに関係する情報を格納する `IndexScanDesc` 構造体）を、`RelationGetIndexScan()` 関数で初期化する。この関数の詳しい動作については後述（8.2節）する。

8.1.2 スキャン中のバッファを記録する変数の初期化

*l.*290 ~ 294

ハッシュインデックススキャンにおいて、どのバッファが現在スキャン中なのかを記録する変数は、ハッシュに特有な情報なので `RelationGetIndexScan()` 関数では初期化されない。
そこで、まずそれを記録する `HashScanOpaque` 構造体 `so` を `palloc()` マクロで確保する。そして構造体の内容を以下のように初期化する。

- スキャン中のバッファが正当かどうかの値（`so->hashso_bucket_valid`）を FALSE に
- スキャン中のバケツに通常ロックを ShareLock モードで取得している場合に、そのブロック番号を記録する変数（`so->hashso_bucket_blkno`）に 0 を代入
- スキャン中バッファを記録している変数（`so->hashso_curbuf`）を invalid 値に
- 記録したバッファを記録している変数（`so->hashso_mrkbuf`）を invalid 値に

最後に、初期化した内容を変数 `scan->opaque` に代入する。

8.1.3 インデックススキャンの登録

*l.*296 ~ 297

インデックススキャンが使用しているページが書き換えられたりしないように、`_hash_regscan()` 関数でスキャン `scan` を登録する。この関数は、グローバル変数 `HashScans` に、スキャンをリスト構造でつなげていく。

8.1.4 戻り値の出力

*l.*299

初期化したインデックススキャンデスクリプタ `scan` へのポインタを戻り値として返す。

8.2 RelationGetIndexScan() 関数の動作

backend/access/index/genam.c *l.*51 ~ 118

インデックススキャンデスクリプタを初期化する RelationGetIndexScan() 関数について説明する。この関数は、ハッシュインデックスに限らず、他のインデックスアクセスメソッドからも同様に呼び出される。

RelationGetIndexScan() 関数は、以下の引数を持つ。

indexRelation（**Relation 型**）スキャンするインデックスリレーション

nkeys（**int 型**）スキャンキーの数

key（**ScanKey 型**）インデックススキャンを限定するスキャンキーの配列

この関数の戻り値は、初期化したインデックススキャンデスクリプタ（IndexScanDesc 型）である。
この関数の動作は大きく次のように構成されている。

1. インデックススキャンデスクリプタ用メモリの確保
2. 各種初期化
3. 統計情報収集器のための各種初期化処理呼び出し
4. アクセスメソッド固有の各種初期化処理呼び出し
5. 戻り値の出力

以下、その詳細について説明する。

8.2.1 インデックススキャンデスクリプタ用メモリの確保

*l.*73

palloc() マクロを用いて IndexScanDescData 構造体の大きさでメモリを確保し、そのメモリへのポインタを IndexScanDesc 型変数 scan に代入する。

8.2.2 各種初期化

*l.*75 ~ 108

確保した構造体に初期値を設定する。
この構造体の詳細と設定される初期値を表 4に示す。

8.2.3 統計情報収集器のための各種初期化処理呼び出し

*l.*110

統計情報収集器が用いる変数 scan->xs_pgstat_info を pgstat_initstats() 関数で初期化する。
統計情報収集器（Statistics Collector）は、サーバの活動状況に関する情報を収集し、報告するサブシステムである。機能としては、

- テーブルとインデックスへのアクセス数のカウント（ディスクブロックおよび行単位）
- 他のサーバプロセスによって実行中のクエリを決定する

といったものがある。この収集・報告のために変数 scan->xs_pgstat_info は用いられる。

8.2.4 アクセスメソッド固有の各種初期化処理呼び出し

*l.*112 ~ 115

アクセスメソッドにスキャンキーやメソッド固有の情報を準備させるため、 index_rescan() 関数を呼び出す。この関数の詳しい動作については後述（8.3節）する。

表 4: IndexScanDescData 構造体

型	変数名	説明	初期値
Relation	heapRelation	ヒープリレーションデスクリプタ	NULL（但し後で設定される）
Relation	indexRelation	インデックスリレーションデスクリプタ	変数 indexRelation を代入
Snapshot	xs_snapshot	スナップショットの種類	SnapshotNow（但し後で設定される）
int	numberOfKeys	スキャンキーの数	変数 nkeys を代入
ScanKey	keyData	スキャンキーの配列	nkeys> 0 の時確保したメモリへのポインタ、さもなくば NULL
bool	kill_prior_tuple	最後に返したタプルが死んでいるかどうか	FALSE
bool	ignore_killed_tuples	死んでいるタプルエントリを返すかどうか	TRUE
bool	keys_are_unique	スキャンキーがインデックスのユニーク性制限を満たすかどうか	FALSE（満たす場合、アクセスメソッドによって後で設定される）
bool	got_tuple	index_getnext() が成功したら TRUE	FALSE
void	*opaque	アクセスメソッド固有の情報	NULL（但し後で設定される）
ItemPointerData	currentItemData	カレントインデックスポインタ	ItemPointerSetInvalid() マクロで invalid 値に
ItemPointerData	currentMarkData	記録したインデックスポインタ	ItemPointerSetInvalid() マクロで invalid 値に
HeapTupleData	xs_ctup	カレントヒープタプル	ItemPointerSetInvalid() マクロ等で invalid 値に（102〜105 行目）
Buffer	xs_cbuf	スキャン中のカレントヒープバッファ	InvalidBuffer
FmgrInfo	fn_getnext	アクセスメソッド固有の getnext() 関数のキャッシュ済みルックアップ情報	invalid 値（但し hashbeginscan() の呼出元である index_beginscan() で、システムカタログ pg_am における各アクセスメソッド毎の “次の有効なタプル” 関数が設定される）
int	unique_tuple_pos	（アクセスメソッドを呼ばずに済ますための *注）論理的場所	0
int	unique_tuple_mark	（アクセスメソッドを呼ばずに済ますための *注）記録した論理的場所	0
PgStat_Info	xs_pgstat_info	統計情報収集器のフック	pgstat_initstats 関数で初期化（後述）

*注: バッファリークが発生するため 7.3.3 以降では disabled されていたが、7.4 で対策が施された。

8.2.5 戻り値の出力

*l.*117

初期化したインデックススキャンデスクリプタ scan へのポインタを戻り値として返す。

8.3 index_rescan() 関数の動作

backend/access/index/indexam.c *l.*299 ~ 330

アクセスメソッド毎の “amrescan” 関数を呼び出す index_rescan() 関数について説明する。

amrescan 関数は、インデックスアクセスメソッドに関するシステムカタログ pg_am の中で指定されている関数で、文献 [1] によれば「“このスキャンを再開” 関数」と記されている。

実際には、amrescan 関数は、再度インデックスをスキャンする際（この関数が ExecIndexReScan() 関数からの呼び出しの場合)、スキャンキーやアクセスメソッド固有の情報を再設定する機能を担当しており、今回のように新規にインデックスをスキャンする際（RelationGetIndexScan() 関数からの呼び出しの場合）にも同じ目的で呼び出される。

index_rescan() 関数は、以下の引数を持つ。

scan（**IndexScanDesc 型**）インデックススキャンデスクリプタ

key（**ScanKey 型**）インデックススキャンを限定するスキャンキーの配列

この関数の戻り値はない。

この関数の動作は大きく次のように構成されている。

1. インデックススキャンデスクリプタのサニティチェック
2. インデックスアクセスメソッド毎の amrescan 関数 OID 取得
3. 各種初期化

4. amrescan 関数呼び出し
5. 統計情報のリセット

以下、その詳細について説明する。

8.3.1 インデックススキャンデスクリプタのサニティチェック

*l.*316

SCAN_CHECKS マクロを用いて変数 scan のサニティチェックをする。
具体的には、scan, scan->indexRelation, scan->indexRelation->rd_am が NULL でないことを確認する。

8.3.2 インデックスアクセスメソッド毎の amrescan 関数 OID 取得

*l.*317

GET_SCAN_PROCEDURE マクロを用いて、RegProcedure 型の変数 procedure に amrescan 関数の OID を代入する。
ハッシュインデックスの場合、hashrescan 関数の OID が登録される。

8.3.3 各種初期化

*l.*319 ~ 323

インデックススキャンデスクリプタ scan のうち、以下を初期化する。

- scan->kill_prior_tuple を FALSE に
- scan->keys_are_unique を FALSE に
- scan->got_tuple を FALSE に
- scan->unique_tuple_pos を 0 に
- scan->unique_tuple_mark を 0 に

8.3.4 amrescan 関数呼び出し

*l.*325 ~ 327

インデックスアクセスメソッド毎の amrescan 関数を、Postgres Function Manager 経由で呼び出す。具体的には、関数 OidFunctionCall2() に、amrescan 関数の OIDprocedure、インデックススキャンデスクリプタ変数 scan、スキャンキー配列 key を与えて関数を呼び出す。
ハッシュインデックスの場合、hashrescan 関数が呼び出される。この関数の詳しい動作については後述（9章）する。

8.3.5 統計情報のリセット

*l.*329

統計情報収集器が用いる変数 scan->xs_pgstat_info を pgstat_reset_index_scan() マクロでリセットする。このマクロでは、変数のうちインデックススキャンがカウントされたかどうかのフラグ scan->xs_pgstat_info->index_scan_counted を FALSE にする。

9 ハッシュインデックスのスキャン再開

ハッシュインデックスのスキャン再開の手続き（ハッシュインデックススキャン固有の情報の初期化、スキャンキーの再設定処理）は、hashrescan() 関数によって行われる。

9.1 hashrescan() 関数の動作

backend/
access/hash/
hash.c
*l.*302 ~ 346

hashrescan() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

scan（**IndexScanDesc 型**）インデックススキャンデスクリプタ

scankey（**ScanKey 型**）インデックススキャンを限定するスキャンキーの配列

この関数の戻り値はない[9]。
この関数の動作は大きく次のように構成されている。

1. 各種変数初期化
2. バッファの初期化とバケツへのロック解放
3. インデックスポインタの初期化
4. スキャンキーの再設定

以下、その詳細について説明する。

9.1.1 各種変数初期化

*l.*310 ~ 311

各種変数を初期化する。
どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque）、変数 so に格納する。
スキャン対象のインデックスリレーションを変数 scan から取り出し（ scan->indexRelation）、変数 rel に格納する。

9.1.2 バッファの初期化とバケツへのロック解放

*l.*313 ~ 329

まず変数 so が NULL かどうかを確認する。これが NULL の場合（8章で説明した hashbeginscan() 関数を経由した呼び出しの場合）は、以下の作業は冗長なので次の処理に移る。
そうでない場合（本当にスキャンの再開の場合）は、スキャン用バッファの初期化を行う。

- スキャン中バッファを記録している変数 so->hashso_curbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを_hash_dropbuf() 関数で抜き、変数 so->hashso_curbuf を invalid 値にする。
- スキャン位置を記録したバッファを記録している変数 so->hashso_mrkbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを_hash_dropbuf() 関数で抜き、変数 so->hashso_mrkbuf を invalid 値にする。

さらに、スキャン再開の場合は、スキャン中のバケツに通常ロックを ShareLock モードで取得している場合にそのブロック番号を記録する変数 so->hashso_bucket_blkno を確認し、何らかの値が入っていたら、_hash_droplock() 関数でそのロックを解放し、変数 so->hashso_bucket_blkno に 0 を代入する。

9.1.3 インデックスポインタの初期化

*l.*331 ~ 333

カレントインデックスポインタ scan->currentItemData とスキャン位置を記録したインデックスポインタ scan->currentMarkData を invalid 値にする。

9.1.4 スキャンキーの再設定

l.335 ~ 343

スキャンキーが存在し、かつスキャンキーの数が 1 以上の場合、memmove() 関数を用いて、変数 scan->keyData に変数 scankey の内容をコピーし、スキャンキーの再設定を行う。

また、変数 so が NULL でなければ、スキャン中のバッファが正当かどうかの値 so->hashso_bucket_valid を FALSE に設定する。

10 ハッシュインデックスのスキャン終了

ハッシュインデックスのスキャン終了のための手続きは、hashendscan() 関数によって行われる。

この関数の目的は、ハッシュインデックススキャン固有のリソースを開放することにある。RelationGetIndexScan() 関数（8.2節）で確保した各インデックススキャン共通のリソースは、この関数を（Postgres Function Manager 経由で）呼び出す、上位の index_endscan() 関数で開放される。

10.1 hashendscan() 関数の動作

backend/
access/hash/
hash.c
l.348 ~ 383

hashendscan() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

scan（**IndexScanDesc 型**）インデックススキャンデスクリプタ

この関数の戻り値はない[10]。

この関数の動作は大きく次のように構成されている。

1. 各種変数初期化
2. インデックススキャンの登録抹消
3. バッファの初期化とバケツへのロック解放
4. インデックスポインタの初期化
5. スキャンキーの再設定

以下、その詳細について説明する。

10.1.1 各種変数初期化

l.355 ~ 356

各種変数を初期化する。

どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque）、変数 so に格納する。

スキャン対象のインデックスリレーションを変数 scan から取り出し（scan->indexRelation）、変数 rel に格納する。

10.1.2 インデックススキャンの登録抹消

l.358 ~ 359

インデックススキャンの開始時に、グローバル変数 HashScans にスキャンを登録（8.1.3節）した。スキャン終了にあたり、ここで_hash_dropscan() 関数を用いてスキャン scan をグローバル変数 HashScans から抹消する。

[9]正確には、Postgres Function Manager 互換の関数作成用マクロによる VOID（Datum 型で 0）が返される。
[10]正確には、Postgres Function Manager 互換の関数作成用マクロによる VOID（Datum 型で 0）が返される。

10.1.3 バッファの初期化とバケツへのロック解放

l.361 ~ 373

まず、スキャン用バッファの初期化を行う。

- スキャン中バッファを記録している変数 so->hashso_curbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを _hash_dropbuf() 関数で抜き、変数 so->hashso_curbuf を invalid 値にする。
- スキャン位置を記録したバッファを記録している変数 so->hashso_mrkbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを _hash_dropbuf() 関数で抜き、変数 so->hashso_mrkbuf を invalid 値にする。

次に、スキャン中のバケツに通常ロックを ShareLock モードで取得している場合にそのブロック番号を記録する変数 so->hashso_bucket_blkno を確認し、何らかの値が入っていたら、_hash_droplock() 関数でそのロックを解放し、変数 so->hashso_bucket_blkno に 0 を代入する。

10.1.4 インデックスポインタの初期化

l.375 ~ 377

カレントインデックスポインタ scan->currentItemData とスキャン位置を記録したインデックスポインタ scan->currentMarkData を invalid 値にする。

10.1.5 ハッシュインデックススキャン固有の情報で使っていたリソースの開放

l.379 ~ 380

ハッシュインデックススキャン固有の情報で使っていたリソース（変数 so）を pfree() 関数で開放する。そして、そこを指していたポインタ（scan->opaque）に NULL 値を入れる。

11 現在のハッシュインデックススキャン位置の記録

現在のハッシュインデックススキャン位置記録の手続きは、hashmarkpos() 関数によって行われる。

この関数と、次の 12章で解説する、記録したインデックススキャン位置復元の手続きに用いる hashrestrpos() 関数は、リレーションのソートマージ結合でインデックスをスキャンする際に必要となる[11]。

11.1 hashmarkpos() 関数の動作

backend/
access/hash/
hash.c
l.385 ~ 411

hashmarkpos() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

scan（IndexScanDesc 型） インデックススキャンデスクリプタ

この関数の戻り値はない[12]。
この関数の動作は大きく次のように構成されている。

1. 各種変数初期化
2. 記録済データのピンを抜き初期化
3. 記録対象のデータにピンを刺し記録する

以下、その詳細について説明する。

[11] 文献 [5] の p.277 における「プレースホルダ」に相当する。但し、ハッシュインデックスはソートをサポートしていないので、ソートマージ結合に用いることができない。これらの関数を用いる上位のエクゼキュータで使用条件を判断していると考えられるので、今後調査する。位置記録／復元自体は他の目的でも利用できるが、現状ではソートマージ結合処理でしか用いられていないようである。

[12] 正確には、Postgres Function Manager 互換の関数作成用マクロによる VOID（Datum 型で 0）が返される。

11.1.1 各種変数初期化

l.392 ~ 393

各種変数を初期化する。

どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque）、変数 so に格納する。

スキャン対象のインデックスリレーションを変数 scan から取り出し（ scan->indexRelation）、変数 rel に格納する。

11.1.2 記録済データのピンを抜き初期化

l.395 ~ 399

スキャン位置を記録したバッファを記録している変数 so->hashso_mrkbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを _hash_dropbuf() 関数で抜き、変数 so->hashso_mrkbuf を invalid 値にする。

そして、スキャン位置を記録したインデックスポインタ scan->currentMarkData を invalid 値にする。

11.1.3 記録対象のデータにピンを刺し記録する

l.401 ~ 408

記録対象となるインデックスポインタ（scan->currentItemData）を確認し、何らかの値がすでに入っていたら、以下の処理を行う。

1. カレントバッファ（位置を記録する対象）を _hash_getbuf() 関数で取得し（ロックはかけない。ピンを刺すだけ）、その位置を変数 so->hashso_mrkbuf に代入し記録する。
2. 記録したインデックスポインタ（scan->currentMarkData）に記録対象となるカレントインデックスポインタ（ scan->currentItemData）を代入する。

12 記録したハッシュインデックススキャン位置の復元

記録したハッシュインデックススキャン位置復元の手続きは、 hashrestrpos() 関数によって行われる。

12.1 hashrestrpos() 関数の動作

backend/access/hash/hash.c
l.413 ~ 439

hashrestrpos() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

scan（IndexScanDesc 型） インデックススキャンデスクリプタ

この関数の戻り値はない[13]。

この関数の動作は大きく次のように構成されている。

1. 各種変数初期化
2. 現在のデータのピンを抜き初期化
3. 記録済データにピンを刺しカレントポインタを移動する

以下、その詳細について説明する。

12.1.1 各種変数初期化

l.420 ~ 421

各種変数を初期化する。

どのバッファがスキャンされているかという情報を変数 scan から取り出し（scan->opaque）、変数 so に格納する。

スキャン対象のインデックスリレーションを変数 scan から取り出し（ scan->indexRelation）、変数 rel に格納する。

12.1.2 現在のデータのピンを抜き初期化

l.423 ~ 427

現在のスキャン中バッファを記録している変数 so->hashso_curbuf を確認し、何らかの値が入っていたら、そのバッファへのピンを_hash_dropbuf() 関数で抜き、変数 so->hashso_curbuf を invalid 値にする。

そして、カレントインデックスポインタ scan->currentMarkData を invalid 値にする。

12.1.3 記録済データにロックをかけポインタを移動する

l.429 ~ 436

記録済インデックスポインタ（scan->currentMarkData）を確認し、何らかの値がすでに入っていたら、以下の処理を行う。

1. 位置記録済バッファを_hash_getbuf() 関数で取得し（ロックはかけない。ピンを刺すだけ）、その位置を変数 so->hashso_curbuf に代入しカレントインデックスポインタを移動する。
2. カレントインデックスポインタ（scan->currentItemData）に記録済インデックスポインタ（scan->currentMarkData）を代入する。

13 ハッシュインデックスのバルク削除

ハッシュインデックスのバルク削除のための手続きは、hashbulkdelete() 関数によって行われる。

この関数の目的は、ハッシュインデックスに記録されたタプルのうち、すでにヒープリレーションで削除済となっているものを探してきてまとめて削除することにある。

PostgreSQL の場合、タプルの削除には 2 段階ある。通常の削除ではタプルの状態を示すフラグ lp_flags に削除済という状態が記録されるだけで、タプルそのものはテーブルから物理的に削除されない。このような無駄な領域を回収するために、DB 利用者は VACUUM コマンドを用いる必要がある。バルク削除関数は、この VACUUM コマンドから呼び出されて用いられる。

13.1 hashbulkdelete() 関数の動作

backend/
access/hash/
hash.c
l.441 ~ 625

hashbulkdelete() 関数は、Postgres Function Manager 互換の関数作成用マクロを使って実装されており、Postgres Function Manager を経由して以下の情報を入力する。

rel（**Relation 型**）バルク削除処理対象のインデックスリレーション

callback（**IndexBulkDeleteCallback 型**）コールバックする関数を指定する関数名実引数

*callback_state（**void へのポインタ型**）コールバック関数からの結果を受け取るための変数へのポインタ

ここで、変数 callback で指定されるコールバック関数は、ヒープタプルの状態を確認し、削除済かどうかを判断する関数である。このコールバック関数は、hashbulkdelete() 関数を Postgres Function Manager を経由して

[13]正確には、Postgres Function Manager 互換の関数作成用マクロによる VOID（Datum 型で 0）が返される。

呼び出す index_bulk_delete() 関数の、さらにその呼び出し元（4 種類）によって指定される。4 種類の呼び出し元は、VACUUM の種類が FULL かそうでない（lazy）か、インデックスリレーションのバルク削除の前に行っているヒープリレーションのスキャンにおいて削除すべきタプルが存在したかどうかという条件をもとに選択される[14]。その関係について表5にまとめる。

表 5: コールバック関数の種類

VACUUM の種類	削除タプルの有無	呼び出し元	コールバック関数	機能
FULL	あり	vacuum_index()	tid_reaped()	アイテムポインタが指すタプルを削除していいなら TRUE を返す
FULL	なし	scan_index()	dummy_tid_reaped()	FALSE を返すだけ（スキャンのみで削除しないため）
lazy	あり	lazy_vacuum_index()	lazy_tid_reaped()	アイテムポインタが指すタプルを削除していいなら TRUE を返す
lazy	なし	lazy_scan_index()	dummy_tid_reaped()	FALSE を返すだけ（スキャンのみで削除しないため）

この関数は、Postgres Function Manager 互換の関数作成用マクロにより、VACUUM 結果の報告のための構造体（IndexBulkDeleteResult 型）へのポインタを呼び出し元に返す。

この関数の動作は大きく次のように構成されている。

1. インデックスメタページ情報の取得
2. バケツをスキャンして削除済みエントリを処理
3. インデックスメタページのチェック
4. インデックスメタページの更新
5. 戻り値の出力

以下、その詳細について説明する。

13.1.1 インデックスメタページ情報の取得

*l.*469 ~ 483

インデックスリレーションのメタページ用のバッファを _hash_getbuf() 関数で取得し、リードロックをかけて変数 metabuf に入れる。次に、metabuf に関連するメタページを BufferGetPage() 関数で取得し、変数 metap に入れる。取得したメタページは、_hash_checkpage() 関数でサニティチェックをかける。

ここで、メタページの中の情報（表2参照）で、使用中バケツ番号の最大値（metap->hashm_maxbucket）と、インデックステーブルに格納したタプルの数（metap->hashm_ntuples）をそれぞれ変数 orig_maxbucket,orig_ntuples に格納しておく。さらに、メタページ構造体そのものも memcpy() 関数を使って変数 local_metapage にコピーしておく。これは、メタページ情報の変更の確認や、バケツスキャン時に何度もメタページを読み直すのを抑止するためである。

コピーをとったので、最後に _hash_relbuf() 関数でメタページバッファは解放する。

13.1.2 バケツをスキャンして削除済みエントリを処理

*l.*485 ~ 573

全てのバケツをスキャンし、削除済みヒープタプルを指しているインデックスエントリを削除する。

この処理は 3 重ループで構成されている。一番上のループはバケツ毎の処理、その中のループは各バケツのバケツチェーン（ページ単位）をたぐる処理、さらにその中のループはページ内のタプルをスキャンする処理となる。

*l.*486 ~ 487

ループに入る前に、現在スキャン中のバケツ番号を示す変数 cur_bucket を 0 にし、スキャンが必要な最大のバケツ番号を示す変数 cur_maxbucket に先ほどメタページから得た変数 orig_maxbucket の値を代入する。

[14]詳細は vacuum() 関数から始まる一連の VACUUM 処理について追う必要があるがここでは割愛する。

準備ができたところで、ループを開始する。現在スキャン中のバケツ番号 cur_bucket が最大のバケツ番号 cur_maxbucket を超えない限り以下の while ループ（13.1.2.1節〜13.1.2.6節）を繰り返す。

*l.*496 ~ 500 **13.1.2.1 スキャン対象バケツの取得** スキャン中バケツ番号 cur_bucket に対応するブロック番号を BUCKET_TO_BLKNO() マクロで求め、変数 bucket_blkno に格納する。

そして、バケツを縮めることができるように、_hash_getlock() 関数でスキャン対象バケツブロックへの通常ロックを ExclusiveLock モードで取得する。

*l.*502 ~ 504 **13.1.2.2 このバケツへの他のスキャンがないか確認** 対象のバケツ cur_bucket に対する他のアクティブなスキャンがあるかどうか_hash_has_active_scan() 関数でチェックし、ある場合はエラーとする。

13.1.2.3 バケツ中のバケツチェーンをページ単位にスキャン まず、バケツチェーンをたぐるためのバケツブロック番号変数 blkno に、先頭バケツのブロック番号である bucket_blkno を代入する。

*l.*507

*l.*508 ~ 562 そして、この blkno が正当な値を示している間、以下の while ループ（13.1.2.3.1節〜13.1.2.3.3節）を繰り返す。

*l.*517 ~ 521 **13.1.2.3.1 スキャン対象バッファの取得** スキャン対象バケツのバッファを_hash_getbuf() 関数で取得し、変数 buf に入れる。この時このバッファにライトロック（ここで行うロックはバッファに対する軽量ロック (LWLocks)）がかかる。

次に、buf に関連するページを BufferGetPage() 関数で取得し、変数 page に入れる。取得したページは、_hash_checkpage() 関数でサニティチェックをかける。

ページの中に含まれるスペシャルデータ（アクセスメソッド特有のデータ。3.2.1.3節を参照のこと）へのポインタを PagegetSpecialPointer() マクロで取り出し変数 opaque に入れる。スペシャルデータのうち、このページが属するバケツ番号 opaque->hasho_bucket を変数 cur_bucket と比較し、正しいバケツ番号のページを取得したか確認する。

*l.*524 ~ 525 **13.1.2.3.2 ページ中のタプルをスキャン** まず、現在スキャン中のページのディスクオフセット番号を示す変数 offno を初期化（FirstOffsetNumber、OffsetNumber 型で 1）にし、スキャンが必要な最大のディスクオフセット番号を示す変数 maxoffno に、スキャンするページ page の最大のディスクオフセット番号を PageGetMaxOffsetNumber() マクロで得た値を代入する。

*l.*526 ~ 551 現在スキャン中のページのディスクオフセット番号 offno が最大のディスクオフセット番号 maxoffno を超えない限り以下の while ループを繰り返す。

1. スキャン対象のハッシュタプルを PageGetItem() マクロで取り出し、変数 hitem に格納する。
2. ハッシュタプルが参照しているヒープタプルへのポインタ（hitem->hash_itup.t_tid）を変数 htup に取り出す。
3. ヒープタプルへのポインタ htup をコールバック関数に与えて、このタプルが削除済タプルで、インデックスタプルを削除していいものなのか確認する。前述（表5参照）のとおり、コールバック関数は 4 種類あり、状況に応じて使い分けがされる。

   - もし「削除する」という結果だった場合、PageIndexTupleDelete() 関数を用いて、インデックスページ page 中のオフセット番号 offno にあるタプルを削除する。この関数は、タプルを削除し、その隙間を詰める処理まで行う。
     そしてバケツ／ページの書き換えを実行したことを示す bucket_dirty,page_dirty を TRUE にする。
     タプルの削除をしたので、ディスクオフセット番号 offno はインクリメントせず、最大ディスクオフセット番号 maxoffno を OffsecNumberPrev() マクロでデクリメントする。
     最後に、削除したタプル数を記録する変数 tuples_removed に 1 を加える。

- 「削除しない」という結果の場合、ディスクオフセット番号 offno を OffsecNumberNext() マクロでインクリメントし、インデックスタプル数を記録する変数 num_index_tuples に 1 を加える。

4. このループの先頭に戻る。

l.553 ~ 561 **13.1.2.3.3 必要に応じてページを書き出す** ページの中に含まれるスペシャルデータ変数 opaque から、バケツチェーンの次のブロック番号 opaque->hasho_nextblkno を変数 blkno に代入する。次のブロックが無い場合はここに Invalid 値が入っているので、この while ループを抜けることになる。

そして、変数 page_dirty を確認し、TRUE だった場合、つまりページの書き換えがあった場合は、バッファを_hash_wrtbuf() 関数で書き出し、ライトロックを解放する。変数 page_dirty が FALSE だった場合は、書き出しはせず、_hash_relbuf() 関数でライトロック解放のみする。

以上でこのループの先頭（13.1.2.3.1節）に戻る。

l.564 ~ 566 **13.1.2.4 バケツの縮退** バケツチェーン全体のスキャンが済んだところで、変数 bucket_dirty を確認し、TRUE だった場合、つまり何らかの削除が発生したことがわかった場合、_hash_squeezebucket() 関数を呼び出し、バケツの縮退を試してみる。この関数は、バケツチェーンの後ろからタプルを取得し、バケツチェーンの先頭からスキャンして見つけた空き領域に詰めていく。不必要なオーバーフローバケツは解放される。

l.568 ~ 569 **13.1.2.5 バケツへのロック解放** スキャン対象バケツブロックへの排他ロックを_hash_droplock() 関数で解放する。

l.571 ~ 572 **13.1.2.6 次のバケツに進む** スキャン中バケツ番号 cur_bucket に 1 を加え、このループの先頭（13.1.2.1節）に戻る。

13.1.3 インデックスメタページのチェック

l.575 ~ 587

インデックス中全てのバケツのスキャンが済んだところで、再度インデックスメタページを取得しチェックする。

インデックスリレーションのメタページ用のバッファを_hash_getbuf() 関数で取得し、リードロックをかけて変数 metabuf に入れる。次に、metabuf に関連するメタページを BufferGetPage() 関数で取得し、変数 metap に入れる。取得したメタページは、_hash_checkpage() 関数でサニティチェックをかける。

ここで、メタページの中の情報で、使用中バケツ番号の最大値 metap->hashm_maxbucket と、以前保存しておいた、スキャン前の値 orig_maxbucket を比較する。この値が違っている場合、バケツ分割が途中で発生していたことになるので、追加されたバケツについて追加処理をする必要がある。よって、変数 cur_maxbucket を metap->hashm_maxbucket の値にし、さらにメタページ構造体そのものも memcpy() 関数を使って変数 local_metapage にコピーして、_hash_relbuf() 関数でメタページバッファを解放したうえで、3 重ループの先頭（13.1.2.1節）に戻る。

13.1.4 インデックスメタページの更新

l.589 ~ 614

以上で全てのスキャンが完了したので、インデックスメタページの情報を更新する。

まず、メタページの中の情報で、使用中バケツ番号の最大値（metap->hashm_maxbucket）と、インデックステーブルに格納したタプルの数（metap->hashm_ntuples）を、それぞれ以前保存しておいたスキャン前の値（orig_maxbucket,orig_ntuples）と比較する。

値に差異がなければ、スキャンを開始してからバケツの分割や挿入が行われていないことがわかるので、3 重ループの中のカウンタを信用し、インデックステーブルに格納したタプル数変数 metap->hashm_ntuples に、ループ中に数えたインデックスタプル数の変数 num_index_tuples の値を代入する。

どちらかの値に差異があった場合は、分割されてしまったバケツのタプルをダブルカウントしているかもしれない。なので推測で処理を進めるしかない。具体的には、変数 metap->hashm_ntuples から、ループ中に数えた削除したタプル数の変数 tuples_removed の値を引く。但し値が 0 以下にならないようにする。

更新が完了したので、メタページバッファを_hash_wrtbuf() 関数で書き出し、ライトロックを解放する。

13.1.5 戻り値の出力

l.616 ~ 624

VACUUM 結果の報告のための IndexBulkDeleteResult 型構造体 result を設定する。

まず、インデックスリレーション rel 中の現在のページ数を RelationGetNumberOfBlocks() 関数で取得し、変数 num_pages に代入する。

次に、result 構造体を palloc() 関数で領域確保し、

- result->num_pages に rel 中の現在のページ数 num_pages を、
- result->tuples_removed に今回削除したタプルの数 tuples_removed を、
- result->num_index_tuples に今回削除されなかったタプルの数 num_index_tuples を、

それぞれ代入する。

最後に、result 構造体へのポインタを呼び出し元に返す。

参考文献

[1] 日本 PostgreSQL ユーザ会・文書・書籍関連分科会. PostgreSQL 7.4.3 日本語付属ドキュメント, 2004. http://www.postgresql.jp/document/pg743doc/.

[2] Margo I. Seltzer and Ozan Yigit. A new hashing package for UNIX. In *USENIX Winter*, pp. 173–184, 1991.

[3] Witold Litwin. Linear Hashing: a new tool for file and table addressing. In *The 6th Conference on Very Large Databases*, pp. 212–223, 1980.

[4] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Companies, Inc., third edition, 2003.

[5] 増永良文. リレーショナルデータベース入門. サイエンス社, 新訂版, 2003.