PostgreSQL Internal：ハッシュテーブル

小川 淳（a_ogawa@hi-ho.ne.jp）

本資料ではPostgreSQLの内部データ構造の1つである，ハッシュテーブルについて紹介する。
ハッシュテーブルはキーとデータを関連付けて保持し，キーを指定してデータを素早くとりだすことができる仕組みである。PostgreSQL内部では，ロックマネージャ，リレーションキャッシュ，共有メモリ管理など，多くのモジュールでハッシュテーブルを利用している。
PostgreSQLのハッシュテーブルは，リニアハッシュ[1]と呼ばれるアルゴリズムで実装されている。本資料ではリニアハッシュアルゴリズムの概要について説明した後，PostgreSQLの実装を見ていく。

1. リニアハッシュ

1.1. リニアハッシュ・アルゴリズム

リニアハッシュは，テーブルサイズを1つずつ拡張することができるハッシュテーブルであり，テーブル拡張時の計算量が少ないという特徴がある。
例として，テーブルサイズが4のハッシュテーブルを考える（図1）。

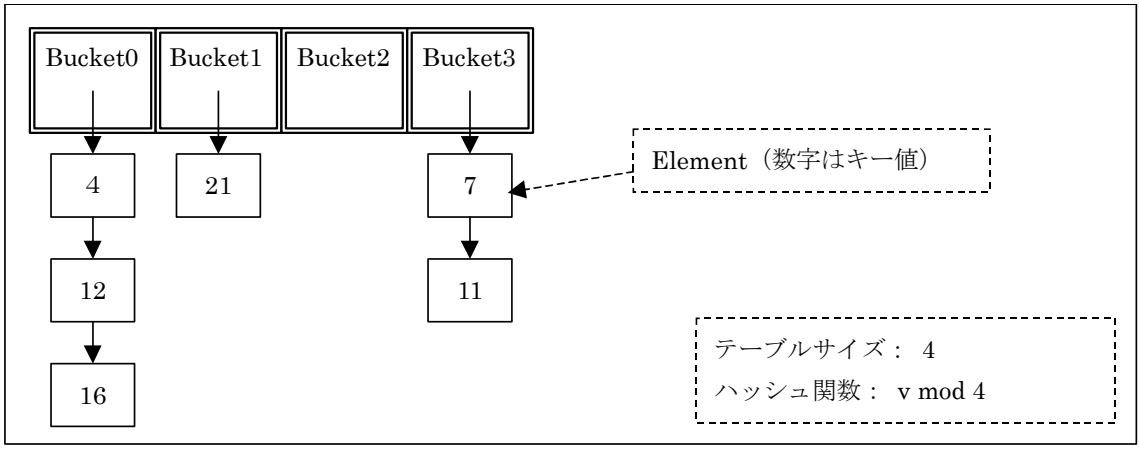

図1 ハッシュテーブル

テーブルの各要素をBucket呼ぶ。Bucketのインデックスは0から始まり，最大のインデックスはテーブルサイズ－1になる。それぞれのBucketをBucket0，Bucket1…と呼ぶこととする。
テーブルに登録されるデータをElementと呼ぶ。各Bucketでは，複数のElementを線形リストで保持する。

あるElementのキー値をk，テーブルサイズをNとすると，このElementを登録するBucketの番号を求める関数は，k mod Nで定義される。（この関数をハッシュ関数と呼ぶ）
ハッシュテーブルのテーブルサイズを拡張したときは，登録されているElementを新しいBucketへ再配置する必要があるが，テーブルを拡張するたびに，全てのElementの再配置を実行するのでは計算量が多くなってしまう。リニアハッシュではハッシュ関数を工夫することにより，テーブルサイズを1つ増やしたときに，再配置を実行するBucketを1つに限定できるようになっている。

リニアハッシュのテーブルを拡張するとき，拡張後のテーブルサイズが現在のハッシュ関数の除数より大きくなる場合は，ハッシュ関数の除数を2倍にする。ただし，計算結果がテーブルサイズを超えてしまうことがあるので，このときは除数を半分にして再度計算する。
つまり，キーをk，テーブルサイズをN，除数をDとした場合，以下のようなルールになる。

k mod Dを計算し，k mod D >= Nのときはk mod (D / 2)を計算する

また，テーブル拡張時にElementの再配置が必要になるBucket番号は，拡張前のテーブルサイズをN，ハッシュ関数の新しい除数をDとした場合，N mod (D / 2)で求めることができる。
テーブルサイズの初期値に8や32などの2の階乗を選択した場合は，これらの計算はbit maskの演算に置き換えることができるので，効率よく計算できる。

図1に示したハッシュテーブルのテーブルサイズを4から5に拡張する場合，拡張後のテーブルサイズがハッシュ関数の除数4より大きくなるので，ハッシュ関数の除数を2倍にしてk mod 8にする。ただし，計算結果がテーブルサイズを超える場合は，k mod 4を計算する。
このとき，Elementの再配置が必要なBucket番号は，4 mod (8 / 2) = 0となるので，Bucket0に登録されているElementの再配置を行う。Bucket1～3については，新旧のハッシュ関数の計算結果は不変であるため，何もしなくてよい。
図2に，テーブルサイズを4から5に拡張した場合のハッシュテーブルの変化を示す。

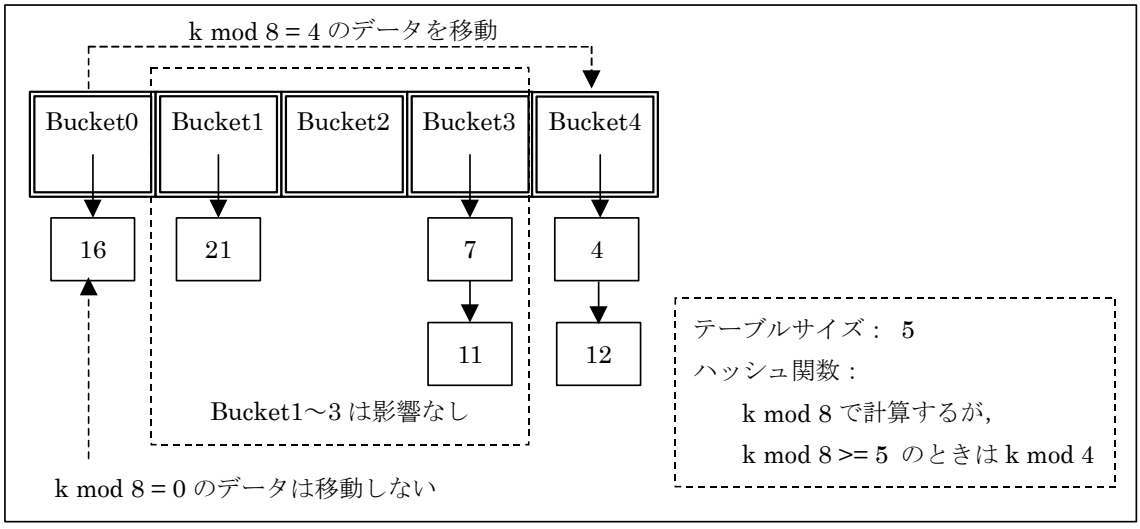

図2 テーブルサイズを4から5へ拡張

同様に，テーブルサイズを 5 から 6 に拡張する場合は，Bucket1 の Element の一部を，Bucket5 へ移動する処理を行う。

1.2. リニアハッシュのデータ構造

リニアハッシュでは，テーブルサイズを動的に拡張できるようにするため，図 3 のようなデータ構造が考えられている。

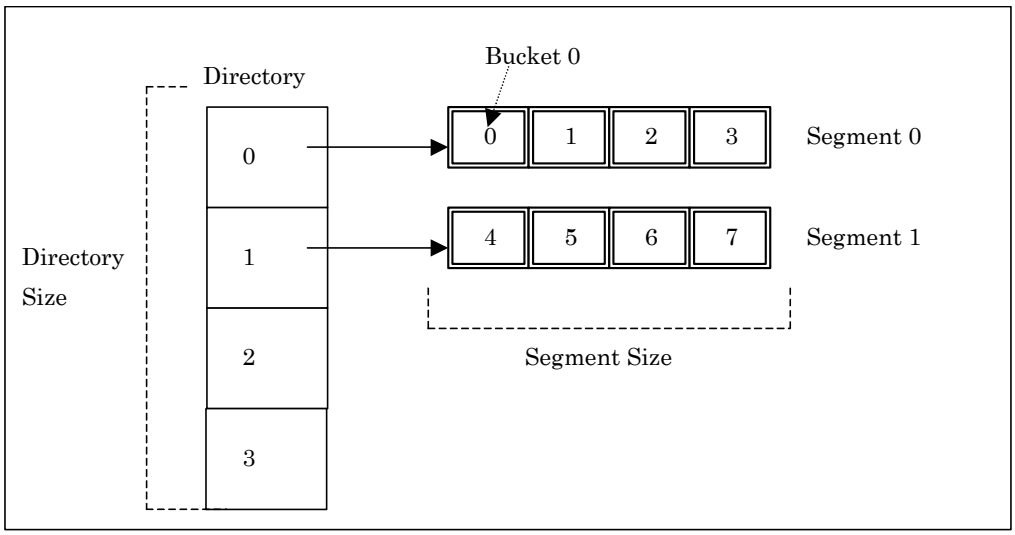

図３ リニアハッシュのデータ構造

リニアハッシュテーブルでは，Directory と Segment という 2 種類の配列で，Bucket を管理する。
1 つのハッシュテーブルには，1 つの Directory が割り当てられる。Directory は Segment の配列であり，複数の Segment を割り当てることができる。
Segment は Bucket の配列であり，複数の Bucket を持つ。1 つのハッシュテーブルには，同じサイズの Segmnet が割り当てられる。つまり，1 つのハッシュテーブルで持つことができる Bucket の数は，Directory Size * Segment Size になる。

Segment は必要に応じてメモリが確保されて，Directory に割り当てられる。最小のハッシュテーブルでは 1 つの Directory と，Directory の 0 番に割り当てられた 1 つの Segment を持つ。
例えば，図３のような segment size が 4 の Segment を 2 つ持つハッシュテーブルで，テーブルサイズを 8 から 9 に拡張する場合は新しい Segment が必要になるので，新しい Segment を作成し，Directory の 2 番へ割り当てる。また，Directory 一杯に Segment が登録されていて，さらにテーブルを拡張したいときは Directory のメモリを再確保(realloc)し，Directory Size を大きくしてから，Segment を作成することができる。

Bucket 番号を b とすると，Directory 内の Segment の位置（Segment 番号）は，b / segment size で求めることができる。また，Segment 内の Bucket 位置は b mod segment size で求めることができる。segment size に 256 などの 2 の階乗を選択した場合は，これらの計算は bit shift と bit mask の演算に置き換えることができるので，効率よく計算できる。

2. PostgreSQL の実装

2.1. 概要

PostgreSQL のハッシュテーブルの実装は，これまでに見てきたリニアハッシュのアルゴリズムを忠実に実装してある。リニアハッシュの仕組みを理解していれば，ソースコードを読むのはそれほど難しくない。
本資料では、PostgreSQL8.0.0beta2 のソースコードを対象にしている。

混乱がないようにハッシュ関数という言葉について説明しておく。1 章のリニアハッシュの説明では，ハッシュ関数は，Element のキーから Bucket 番号を計算するための計算式（k mod D）のことを意味していたが，PostgreSQL のソースコードでは，この計算式は calc_bucket という関数で定義されており，以降の説明でも calc_bucket と呼ぶことにする。
これとは別に，PostgreSQL のソースコードでは hash function という言葉が登場するが，これは Element のキーから 32bit の数値（hash value）を計算するための関数を指している。
PostgreSQL のハッシュテーブルの実装では，Element のキーとして文字列や構造体など，任意のデータ型を使うことが出来るようになっており，Element のキーから hash function で 32bit のハッシュ値を求め，ハッシュ値から calc_bucket で bucket 番号を計算している。以降の説明では，ハッシュ関数という言葉は，calc_bucket のことではなく，hash function のことを指す。

2.2. データ構造

ハッシュテーブルのデータ構造は，include/utils/hsearch.h で定義されている。主要なデータ構造について説明する。
ハッシュテーブルは HTAB と HASHHDR という 2 つの構造体で管理される。HTAB にはハッシュテーブルの構築時に定義され，プログラム実行中には不変のパラメータを持ち，HASHHDR にはデータの挿入・削除などで，プログラム実行中に変化するパラメータを持つ。

2.2.1. HTAB

HTAB 構造体の各メンバの説明を，以下に記す。

構造体のメンバ	説明
hctl	HASHHDR へのポインタ
dir	リニアハッシュの Directory（図 3 を参照）
hash	キーから 32bit のハッシュ値を計算するハッシュ関数のポインタ(*1)
match	２つのキーを比較する関数のポインタ(*2)
alloc	メモリ割り当ての関数。Segment や Element などのメモリ確保で使用。(*3)
hcxt	メモリコンテキスト
tabname	ハッシュテーブルの名前（エラーメッセージに出力される）
isshared	ハッシュテーブルが shared memory 上にあるときは，true

(*1)ハッシュ関数は以下のように定義されている。

```
typedef uint32 (*HashValueFunc) (const void *key, Size keysize);
```

(*2)キー比較関数は以下のように定義されている。memcmp()や strncmp()と同じインタフェース。

```
typedef int (*HashCompareFunc) (const void *key1, const void *key2, Size keysize);
```

(*3)メモリ割り当て関数は以下のように定義されている。malloc と同じインタフェース。

```
typedef void *(*HashAllocFunc) (Size request);
```

2.2.2. HASHHDR

HASHHDR 構造体の各メンバの説明を，以下に記す。

構造体のメンバ	説明
dsize	Directory Size（図３を参照）
ssize	Segment Size（図３を参照）。2 の乗数を指定する必要がある
sshift	log2(ssize)の値が入る。Segment 番号を計算するために使う。
max_bucket	使用可能な Bucket 番号の最大値（テーブルサイズ － 1 になる）
high_mask	calc_bucket で k mod D を計算するための mask 値
low_mask	k mod D > max_bucket のとき，k mod (D / 2)を計算するための mask 値
ffactor	ハッシュテーブルの密度。大きいほどテーブルの拡張回数が少ない
nentries	ハッシュテーブルに登録されている Element の数
nsegs	Directory に割り当て済みの Segment の数
keysize	Element のキーのサイズ
entrysize	Element 全体のサイズ
max_dsize	Directory Size の limit。Directory を拡張するときにチェックされる
freeList	Element 用のメモリの freeList

2.2.3. HASHELEMENT

ハッシュテーブルに登録される Element は，HASHELEMENT 構造体で管理される。HASHELEMENT は以下のように定義されている。

```
typedef struct HASHELEMENT
{
    struct HASHELEMENT *link;   /* link to next entry in same bucket */
    uint32  hashvalue;          /* hash function result for this entry */
} HASHELEMENT;
```

Element はリストで管理されるので，link に同一 Bucket 内の次の HASHELEMENT へのポインタを持つ。
hashvalue には HTAB 構造体の hash メンバで指定されたハッシュ関数で計算されたハッシュ値が入る。
また，メモリ上では各 HASHELEMENT の後に，データを保存する領域が確保される（図４）。
この領域に構造体を保存する場合は，キーとなるデータを，構造体の先頭のメンバに配置する必要がある。
（dynahash.c 内部でキーの比較を実行するときに，図４に示すような配置でデータが入っていることが前提になっている）

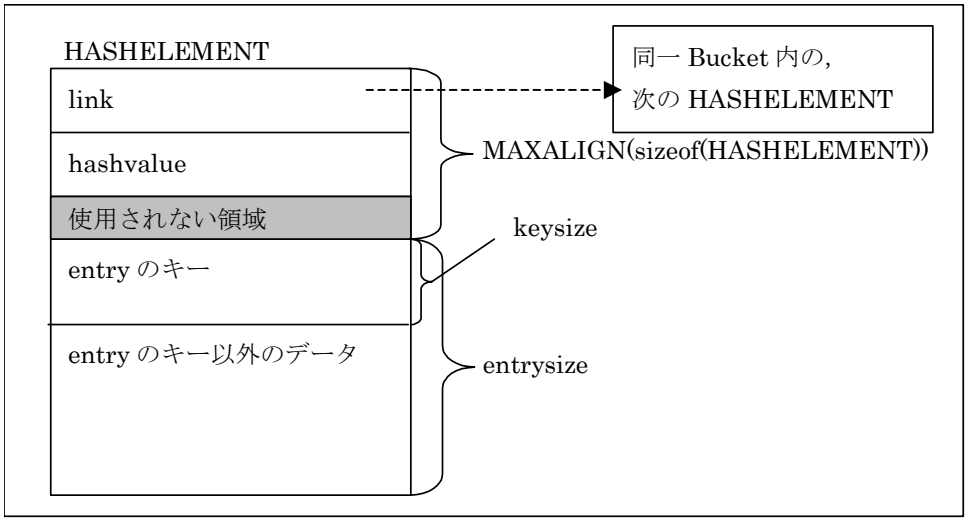

図４ HASHELEMENT のデータ構造

2.2.4. HASHCTL

ハッシュテーブルを作成するとき，各種パラメータにデフォルト値以外の値を指定する場合は，その値を HASHCTL 構造体に設定する。
HASHCTL 構造体の各メンバの説明を，以下に記す。

構造体のメンバ	説明	デフォルト値
ssize	Segment Size	DEF_SEGSIZE(256)
dsize	Directory Size	DEF_DIRSIZE(256)
max_dsize	Directory Size の最大値	NO_MAX_DSIZE(-1)
ffactor	ハッシュテーブルの密度	DEF_FFACTOR(1)
keysize	Element のキーのサイズ	sizeof(char *)
entrysize	Element 全体のサイズ	2 * sizeof(char *)
hash	ハッシュ関数のポインタ	string_hash
match	２つのキーを比較する関数のポインタ	hash が string_hash のときは strncmp それ以外のときは，memcmp
alloc	メモリ割り当ての関数	MEM_ALLOC

dir	Directory のポインタ	NULL(*1)
hctl	HASHHDR のポインタ	NULL(*1)
hcxt	メモリコンテキスト	DynaHashCxt

(*1)dir, hctl は, ハッシュテーブルを共有メモリ上に作成するときに設定する

2.3. インターフェース

ハッシュテーブルのインターフェースは, include/utils/hsearch.h で定義されている。

2.3.1. hash_create：ハッシュテーブルの作成

【定義】

```
HTAB *hash_create(const char *tabname, long nelem, HASHCTL *info, int flags);
```

【引数】

引数名	説明
tabname	ハッシュテーブルの名前
nelem	登録される Element の予想数（テーブルサイズの初期値を計算するために使う）
info	HASHCTL 構造体（ハッシュテーブルの各種パラメータを設定）
flags	info に指定した HASHCTL 構造体で, 有効なパラメータのフラグ include/utils/hsearch.h に定義されており, 例えば info->ssize と info->dsize を設定したときは, HASH_SEGMENT｜HASH_DIRSIZE などと指定する。

【戻り値】

ハッシュテーブルの構築に成功したときは, HTAB のポインタを返す。

失敗したときは NULL を返す。

【説明】

info で指定されたパラメータを使用して, ハッシュテーブルを構築する。

2.3.2. hash_destroy：ハッシュテーブルの削除

【定義】

```
void hash_destroy(HTAB *hashp);
```

【引数】

引数名	説明
hashp	ハッシュテーブルのポインタ

【戻り値】

なし。

【説明】

hashp で指定されたハッシュテーブルを削除する。

hash_create()でメモリコンテキストを指定して作成したハッシュテーブルのみ, hash_destroy()で削除できる。

2.3.3. hash_stats：ハッシュテーブルのステータスを表示

【定義】

```
void hash_stats(const char *where, HTAB *hashp);
```

【引数】

引数名	説明
where	hash_stats をコールするプログラムの場所を指定する（表示で使用）
hashp	ハッシュテーブルのポインタ

【戻り値】

なし。

【説明】

hashp で指定されたハッシュテーブルのステータスを表示する。

表示される情報は，アクセス数，コリジョン数，エレメント数など。

HASH_STATISTICS マクロを有効にして，ビルドする必要がある。

2.3.4. hash_search：ハッシュテーブルの検索，Element の登録，削除

【定義】

```
void *hash_search(HTAB *hashp, const void *keyPtr, HASHACTION action, bool *foundPtr);
```

【引数】

引数名	説明
hashp	ハッシュテーブルのポインタ
keyPtr	検索するキーのポインタ
action	実行する操作(詳細は下表を参照)
foundPtr	検索で見つかった場合，true が設定される。見つからないときは false。 このフラグが必要ないときは，NULL を指定する

action は以下の 5 種類がある（include/utils/hsearch.h に定義）

引数名	説明
HASH_FIND	keyPtr で指定されたキーの Element が登録されている場合，見つかったデータのポインタを返す。見つからない場合，NULL を返す。
HASH_ENTER	keyPtr で指定されたキーの Element が登録されている場合，見つかったデータのポインタを返す。見つからない場合，新しい Element を作成して，そのデータのポインタを返す。ポインタが検索で見つかったものか，見つからずに新しく作成されたものかは，foundPtr で判断する。 データが新しく作成された場合、キーデータ以外のデータは初期化されていないので、hash_search()の呼び出し側でデータを設定する必要がある。
HASH_REMOVE	keyPtr で指定されたキーの Element を削除する。
HASH_FIND_SAVE	keyPtr で指定されたキーの Element が登録されている場合，見つかったデータのポインタを返す。見つからない場合，NULL を返す。 このとき hash_search の static 変数に，Element の位置を保存する。
HASH_REMOVE_SAVED	hash_search の static 変数に保存されている Element を削除する。

【戻り値】

Element が見つかった場合，データのポインタを返す。見つからない場合は NULL を返す。

HASH_ENTER で NULL が返ったときはメモリ不足。

HASH_REMOVE や HASH_REMOVE_SAVED で返されたポインタはメモリから削除されているので，ハッシュテーブルに見つかったかどうかの判定をする以外には，操作してはいけない。（ダングリング・リファレンスの使用になってしまう）

【説明】

ハッシュテーブルから keyPtr で指定されたキーを持つ Element を検索する。Element が見つかった場合，action で指定した動作をする。

2.3.5. hash_seq_init：シーケンシャルサーチの初期化

【定義】

```
void hash_seq_init(HASH_SEQ_STATUS *status, HTAB *hashp);
```

【引数】

引数名	説明
status	HASH_SEQ_STATUS 構造体のポインタ
hashp	ハッシュテーブルのポインタ

【戻り値】

なし。

【説明】

ハッシュテーブルのシーケンシャルサーチを行うために，status を初期化する。

2.3.6. hash_seq_search：シーケンシャルサーチ

【定義】

```
void *hash_seq_search(HASH_SEQ_STATUS *status);
```

【引数】

引数名	説明
status	HASH_SEQ_STATUS 構造体のポインタ

【戻り値】

ハッシュテーブルに登録されているデータのポインタを返す。
テーブルを最後までサーチしたときは，NULL を返す。

【説明】

ハッシュテーブルに登録されているデータを，シーケンシャルサーチする。
hash_seq_init()を実行したあと，hash_seq_search()を実行するたびに，次のデータを返す。

2.3.7. hash_estimate_size：メモリサイズの見積り

【定義】

```
long hash_estimate_size(long num_entries, Size entrysize);
```

【引数】

引数名	説明
num_entries	Element の最大登録数
entrysize	Element のサイズ

【戻り値】

メモリサイズ。

【説明】

entrysize で指定した大きさで num_entries で指定した数の Element を，ハッシュテーブルに登録する場合に必要なメモリサイズを計算する。Postmaster 起動時に，共有メモリの大きさを計算するために使われる。メモリサイズには，HASHHDR，Directory，Segment，Element のサイズの合計になる。ただし，HTAB 構造体はローカルメモリ上に確保されるため，計算結果に含まれない。

2.3.8. hash_select_dirsize：ディレクトリサイズの見積り

【定義】

long hash_select_dirsize(long num_entries);

【引数】

引数名	説明
num_entries	Element の最大登録数

【戻り値】

ディレクトリサイズ。

【説明】

num_entries で指定した数の Element を保存するために，必要なディレクトリサイズを計算する。
共有メモリ上にハッシュテーブルを作成するときに，利用される。

2.3.9. string_hash：文字列用ハッシュ関数

【定義】

uint32 string_hash(const void *key, Size keysize);

【引数】

引数名	説明
key	キーのポインタ
keysize	キーサイズ

【戻り値】

32bit のハッシュ値。

【説明】

文字列(NULL で終端されたデータ)のハッシュ値を計算する。
内部的には，access/hash/hashfunc.c の hash_any()関数を実行する。keysize パラメータは使われず，strlen(key)で取得した文字列の長さを，hash_any()へ渡す。

2.3.10. tag_hash：固定長データ用ハッシュ関数

【定義】

uint32 tag_hash(const void *key, Size keysize);

【引数】

引数名	説明
key	キーのポインタ
keysize	キーサイズ

【戻り値】
32bit のハッシュ値。

【説明】
タグ（固定長データ）のハッシュ値を計算する。
タグは固定長のデータであれば，int などの数字や構造体など，なんでもよい。
string_hash と同様に，内部的には，access/hash/hashfunc.c の hash_any()関数を実行する。

2.4. 内部関数

ハッシュテーブルの実装(dynahash.c)で内部的に使われている関数やマクロの中から，重要と思われるものを紹介する。

2.4.1. ELEMENTKEY

Element のポインタから，キーの位置を計算する。
（図４ HASHELEMENT の構造を参照すると，理解しやすいと思う）

2.4.2. calc_bucket

32bit のハッシュ値から Bucket 番号を計算する。
(k mod D を計算して，テーブルサイズを超える場合は k mod (D / 2)を計算)
テーブルサイズは 2 の階乗になるように調節されるので，HASHHDR の highmask，lowmask を使って bitmask を計算する。

2.4.3. expand_table

テーブルサイズを拡張する。
テーブルサイズの拡張は，hash_search で新しい Element を追加するときに，以下の式が成立する場合に実行される。

Element 数 / (テーブルサイズ + 1) > ffactor

2.4.4. element_alloc

Element 用のメモリを確保する関数。HASHELEMENT_ALLOC_INCR で定義された個数分のメモリを確保し，htcl->freelist に登録する。(HASHELEMENT_ALLOC_INCR は 32 が定義されている)
hash_search で新しい Element を追加するときは，htcl->freelist からメモリをもらう。free_list が空のときは，element_alloc を実行した後，freelist を使う。

2.4.5. dir_realloc

Directory を使い切ったときに，Directory Size を 2 倍に拡張する。
(共有メモリ上のハッシュテーブルは，Directory を拡張できない)

3. 付録

3.1. ハッシュテーブルの使用箇所

PostgreSQL の Backend 内で，ハッシュテーブルを使っている箇所をリストアップした。

名前	ファイル	共有	用途
XLOG relcache	xlogutil.c		リカバリ処理中に Open したリレーションの情報をキャッシュする
Prepared Queries	prepare.c		PREPERE 文で解析した，SQL の実行 Plan を保存する(PREPERE 文で指定した名前がキーになる)
TupleHashTable	execGroup.c		group by を処理するために，group ごとの tuple を管理する
DupHashTable	nodeIndexscan.c		multiple index scan するとき，1 つの tuple を複数回返さないようにするために，1 度返した tuple の tid を覚えておく
Dead Backends	pgstat.c		stat プロセスが，Dead Backend のリストを管理
Databases hash	pgstat.c		stat プロセスが，Database 単位の統計情報を管理
Per-database table	pgstat.c		stat プロセスが，Table 単位の統計情報を管理
locallock hash	lock.c		自プロセスが取得したロックを管理
lock hash	lock.c	○	ロックの情報を管理
proclock hash	lock.c	○	プロセスごとのロック情報を管理
Pending Ops Table	md.c		stand alone で起動したときなどに，fsync 要求を管理する
smgr relation table	smgr.c		storage manager で relation object を管理する
RI query cache	ri_triggers.c		Foreign Key などの contraint check 用のトリガーの解析結果をキャッシュする
Operator class cache	relcache.c		opclass をキャッシュする(pg_amop の情報)

Relcache by name	relcache.c		名前をキーにして Relation をキャッシュする
Relcache by OID	relcache.c		OID をキーにして Relation をキャッシュする
Type information cache	typcache.c		Type をキャッシュする(pg_type の情報)
Record information cache	typcache.c		Record Type をキャッシュする
CFuncHash	fmgr.c		function manager で C function を管理する
Portal hash	portalmem.c		Portal メモリを管理する
ShmemIndex	shmem.c	○	共有メモリを管理する
Shared Buffer Lookup Table	buf_table.c	○	共有バッファを管理する
Free Space Map Hash	freespace.c	○	フリースペースを管理する

4. 参考文献：

[1] Per-Åke Larson. Dynamic Hash Tables
Communications of the ACM Volume 31, Issue 4(April 1988) Pages: 446 - 457
(動的拡張可能なハッシュテーブルの実装方法として、リニアハッシュとスパイラルストレージについて紹介している。本資料の図１～３はこの論文を参考にした。)