ハッシュインデックスのしくみ

2004/11/30

NTTサイバースペース研究所

寺本　純司

PostgreSQL

PostgreSQLのインデックス

- PostgreSQLで利用できるインデックス
  - B-Tree
    - PostgreSQLで一般的に用いられる二分木インデックス。
    - 最適化が最も進んでいる。
  - R-Tree
    - 二次元空間データ問い合わせに適したインデックス。
  - GiST
    - ユーザ定義により拡張可能なインデックス。
  - Hash
    - Exact Match用の単純なインデックス。
      - 但し現状B-Treeよりも性能が悪く使用は推薦されていない
    - 仕組みも単純なため、インデックスのしくみを学ぶのにはよい。
      （と、当時は思っていた）

インデックスに関するシステムカタログ

- pg_am
  - インデックスアクセスメソッド (AM)の登録に必要な情報のカタログ
    - AMがサポートしている機能情報
      - amname　　AMの名前
      - amowner　　所有者ID（現在未使用）
      - amstrategies　　このAM用の演算子ストラテジの数（ハッシュなら"="）
      - amsupport　　このAM用のサポートルーチンの数（ハッシュ値計算）
      - amorderstrategy　　このAMがソート順を提供する場合、その演算子番号
      - amcanunique　　このAMが一意性インデックスをサポートするか
      - amcanmulticol　　このAMが複数列インデックスをサポートするか
      - amindexnulls　　このAMがインデックスエントリにNULLを許容するか
      - amconcurrent　　このAMが同時更新をサポートするか
    - AMが実装している関数情報
      - amgettuple, aminsert, ambeginscan, amrescan, amendscan, ammarkpos, amrestrpos, ambuild, ambulkdelete, amcostestimate, amvacuumcleanup（7.4で新設、なくても良い）
  - つまり、「演算子」「サポートルーチン」「10+1種のAM関数」を作成し、pg_amに登録すれば、新しいインデックスを実装できる。

PostgreSQLのハッシュ

- アルゴリズムはWitold Litwinの線形ハッシュ

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

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

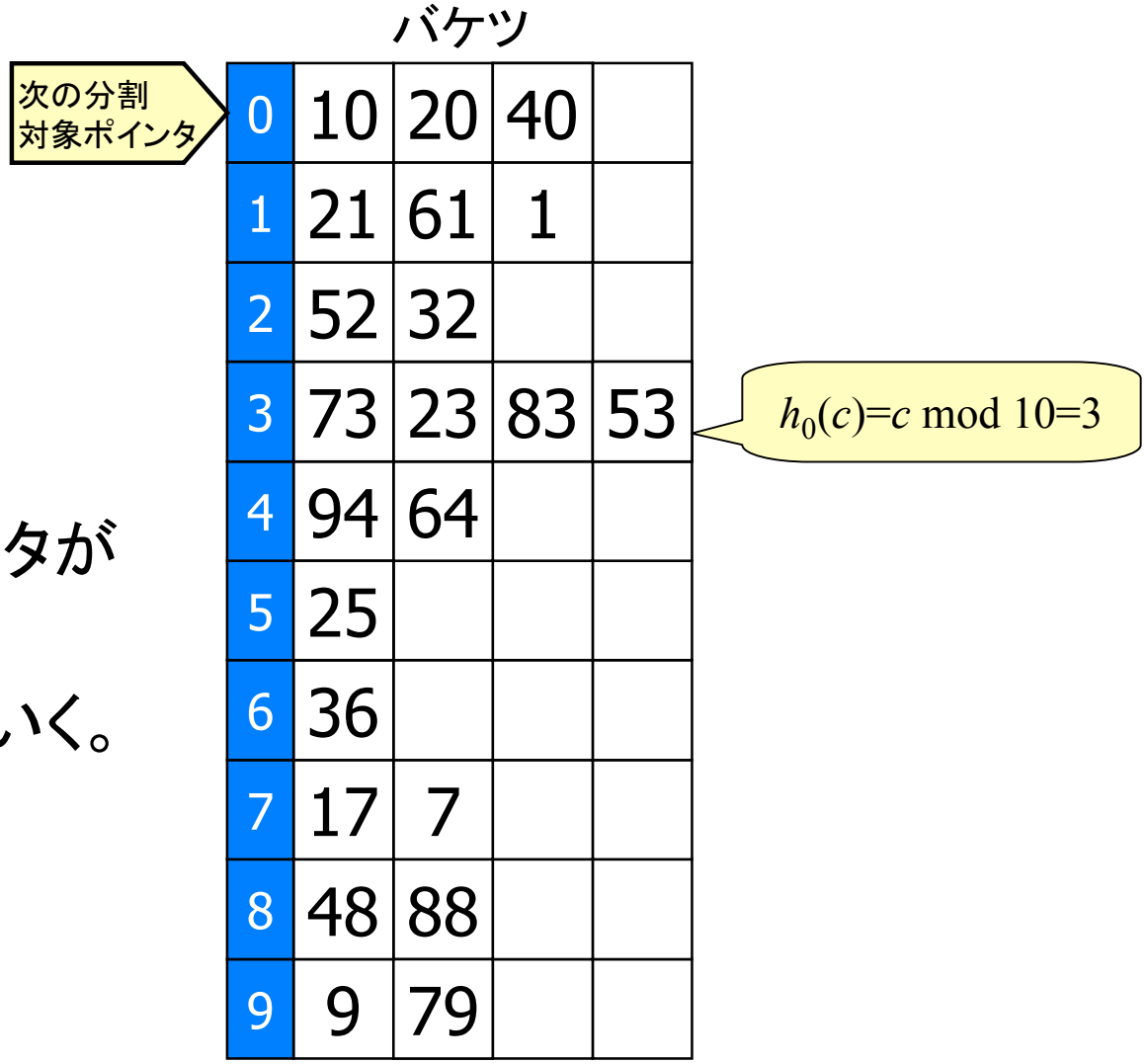
とする。
さらに、h_0のスプリット関数として$h_1,h_2,\ldots,h_i$を定義する。
スプリット関数は以下の条件を満たす。

1. $h_i: C \rightarrow \{0,1,\ldots,2^iN\text{-}1\}$
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. $h_i(c)=h_{i-1}(c)+2^{i-1}N$

- 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（例: $h_i(c)=c \bmod 2^iN$）

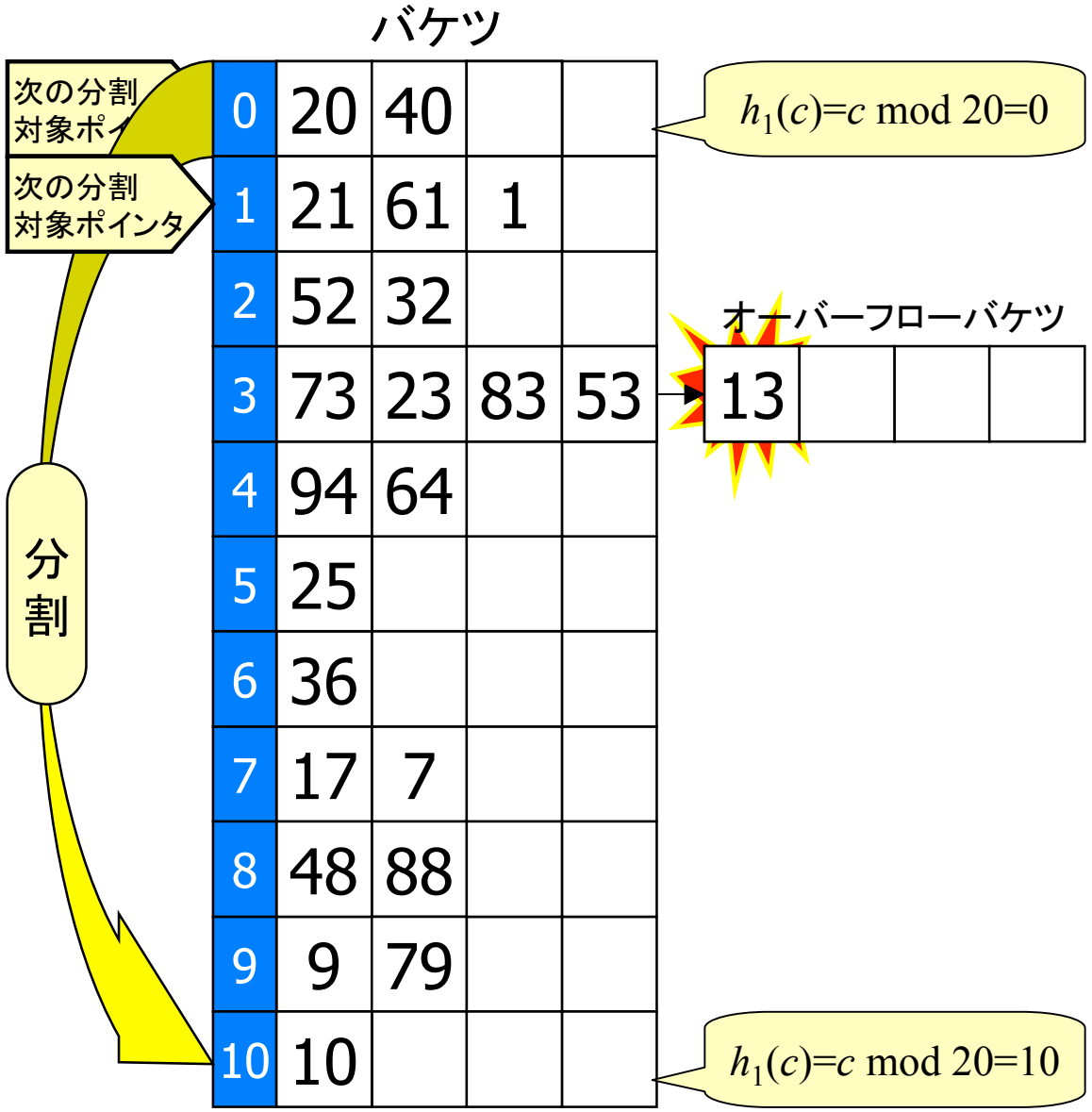
線形ハッシュ①

- 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。
- バケツの中には4つデータが入る。
- h_0でデータを振り分けていく。



線形ハッシュ②

- 13を挿入しようとしたらバケツがオーバーフロー
- バケツを分割する
  - オーバーフローしたバケツを分割せず、0番バケツから順番に分割
  - 0番バケツの中身を、スプリット関数
    $h_1(c)=c \bmod 20$
    で0番と10番バケツに分配
  - 以降、$h_0(c)=0$の時はハッシュ関数は$h_1(c)$を用いる
  - 分割対象ポインタを進める
  - （PGでは2の累乗で分割）
- オーバーフローしたデータは？
  - オーバーフローバケツをつなげて格納
  - そのうち分割の順番が来るので、オーバーフローバケツはそれほど必要としない

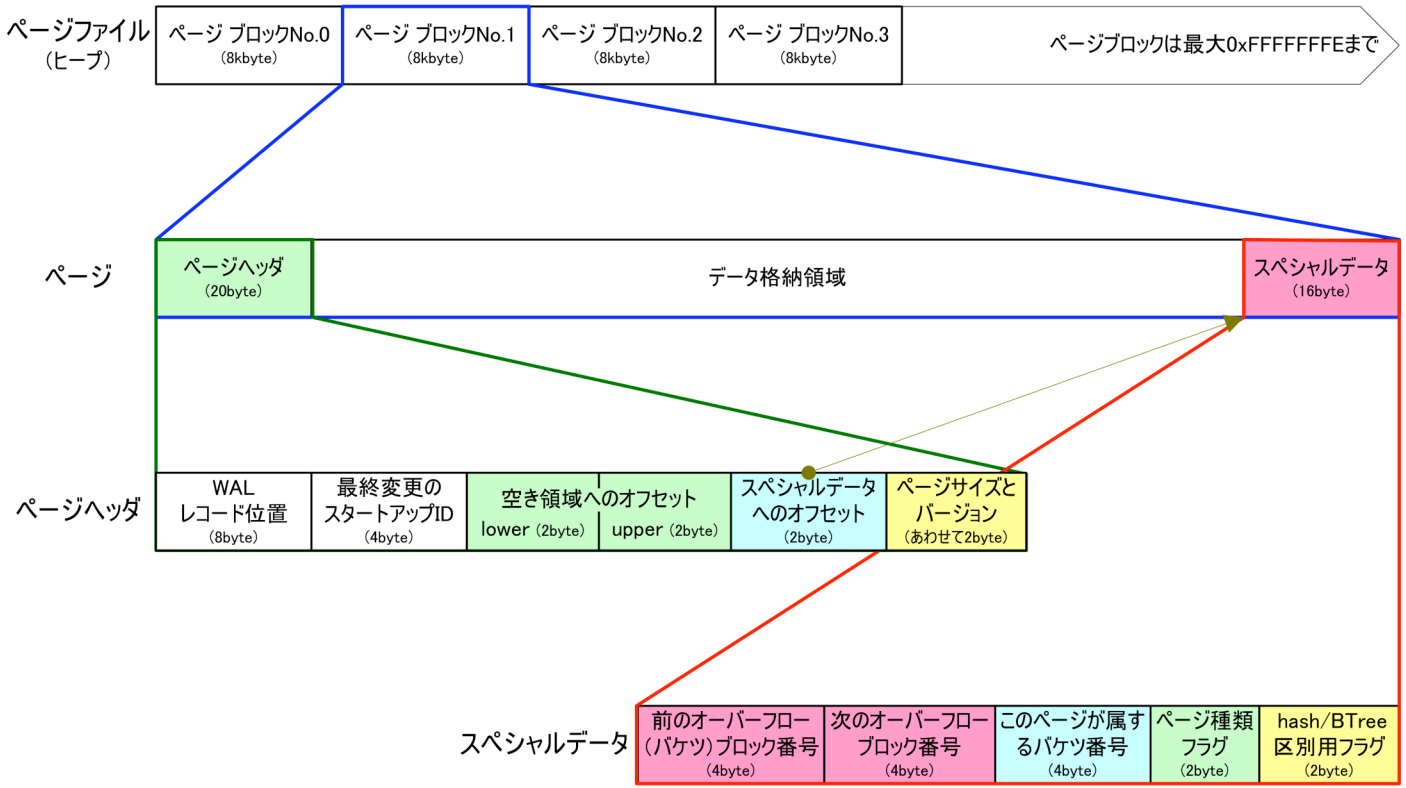


線形ハッシュ③

- 最初のバケツが分割され、総バケツ数が2倍になったら
  - iがインクリメントされ、基本ハッシュ関数としてh_0の替わりにh_1、スプリット関数としてh_2を用いる
  - 分割対象ポインタを0番に戻す
  - 以下、バケツ数が倍になる度にiを増やしポインタを戻す
- バケツの分割は、オーバーフローバケツを生成した時だけでなく、ある適当なタイミングで行っても良い
  - バケツの数に対するタプルの数の割合が一定値を超えた時、等（PostgreSQLもこの方法を採用している）
  - 7.4系ではオーバーフローバケツを生成しても分割は発生せず、上記条件が成立した時のみ分割している
- データが均一でないと、オーバーフローバケツが大量生産される欠点もある

ハッシュインデックスのデータ構造

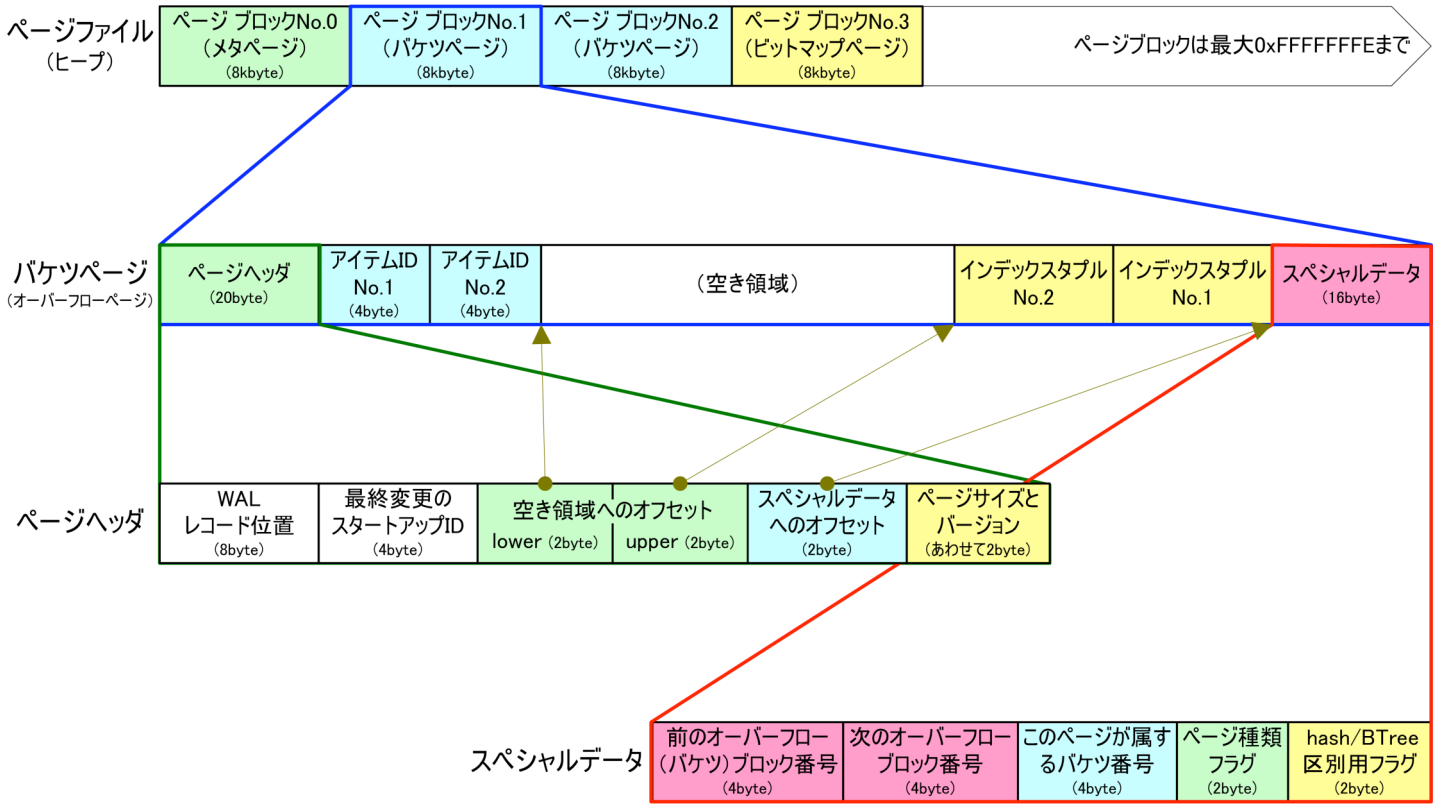
- データはページファイルに格納される
  - ページヘッダ領域　/　データ格納領域
  - スペシャルデータ領域
    - アクセスメソッド固有の情報



- バケツチェーンのためのポインタ
- このページが属するバケツ番号
- ページ種類フラグ
- haah/BTree区別用フラグ

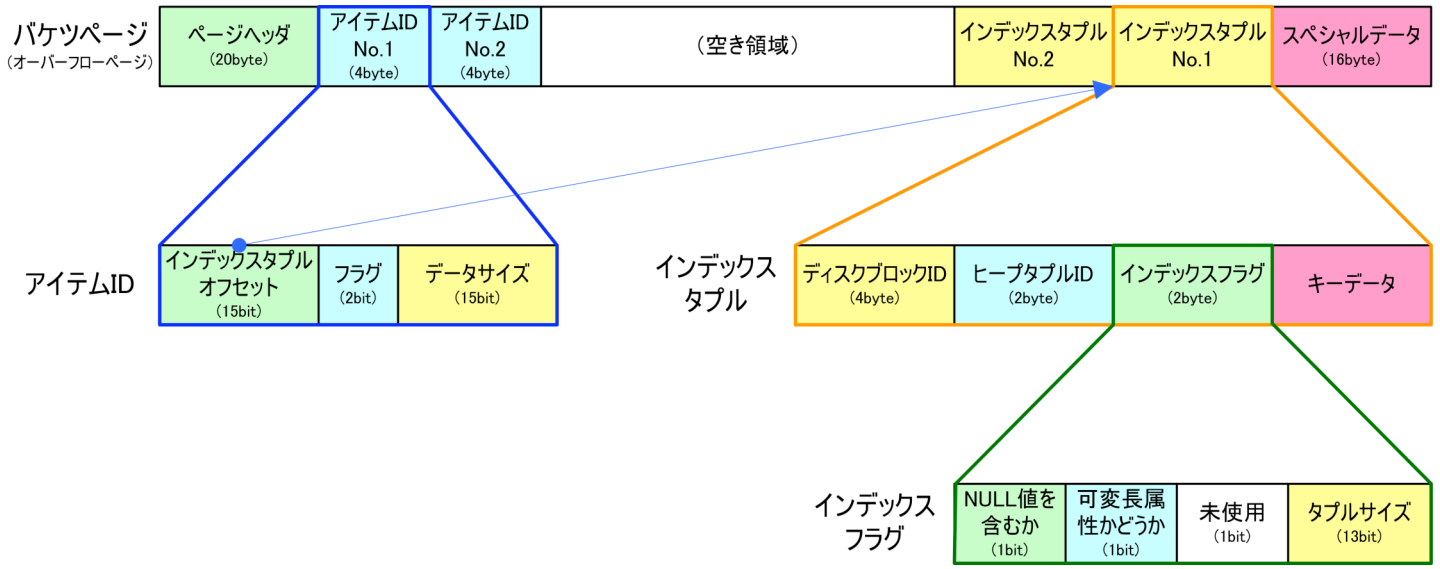
バケツページとオーバーフローページ

- ページ種類フラグが違うが構造は同じ
  - インデックスタプルはアイテムIDと対をなす
    - IDはデータ格納領域の先頭から、インデックスタプルは最後から割り当てられる



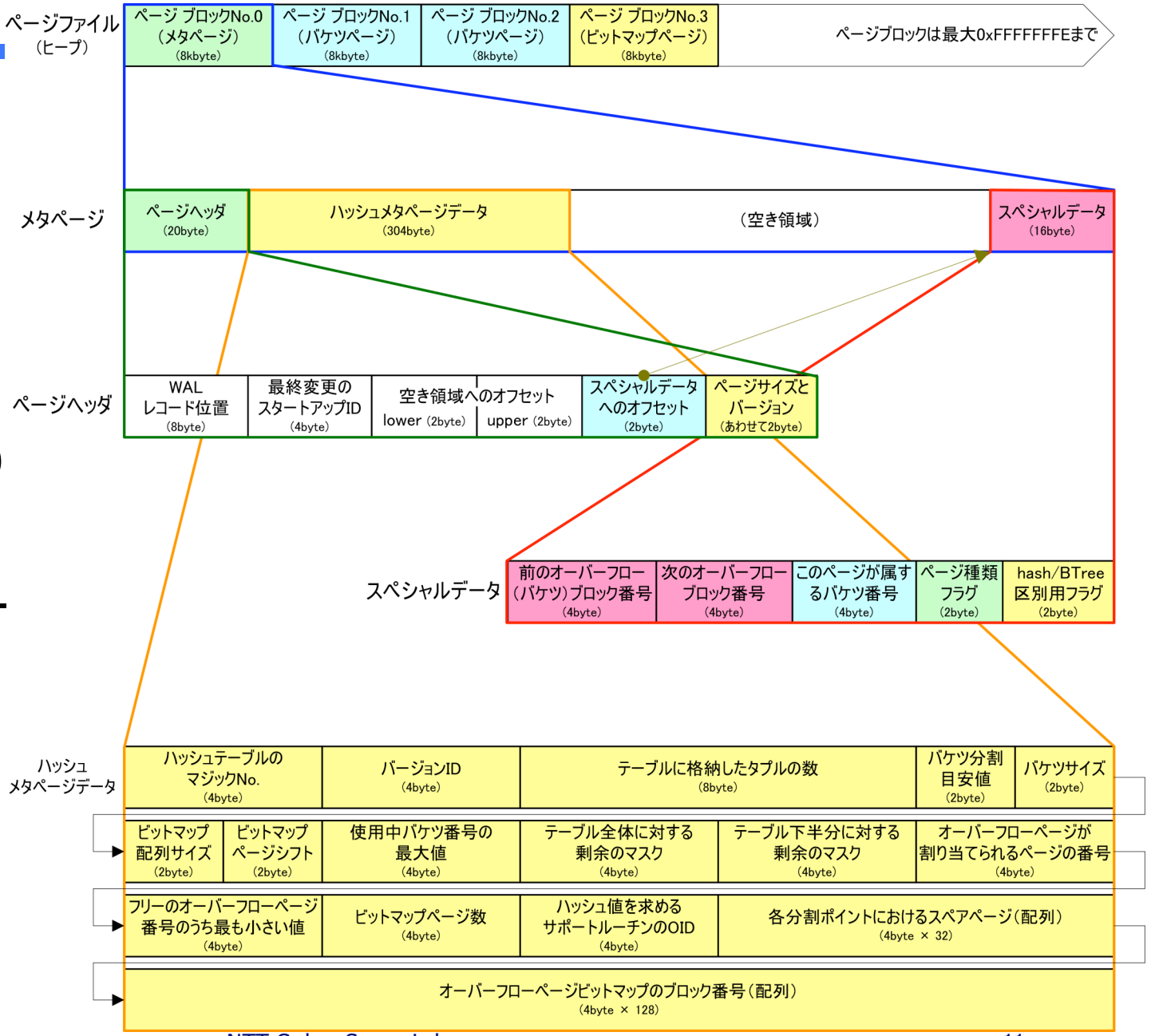
アイテムIDとインデックスタプル

- アイテムID
  - インデックスタプルオフセット（位置）
  - タプルの状態フラグ（使用中／削除済）
  - インデックスタプルの大きさ
- インデックスタプル
  - インデックスしたヒープタプルへのポインタ
  - インデックスタプルの情報
  - 実際のキーデータ



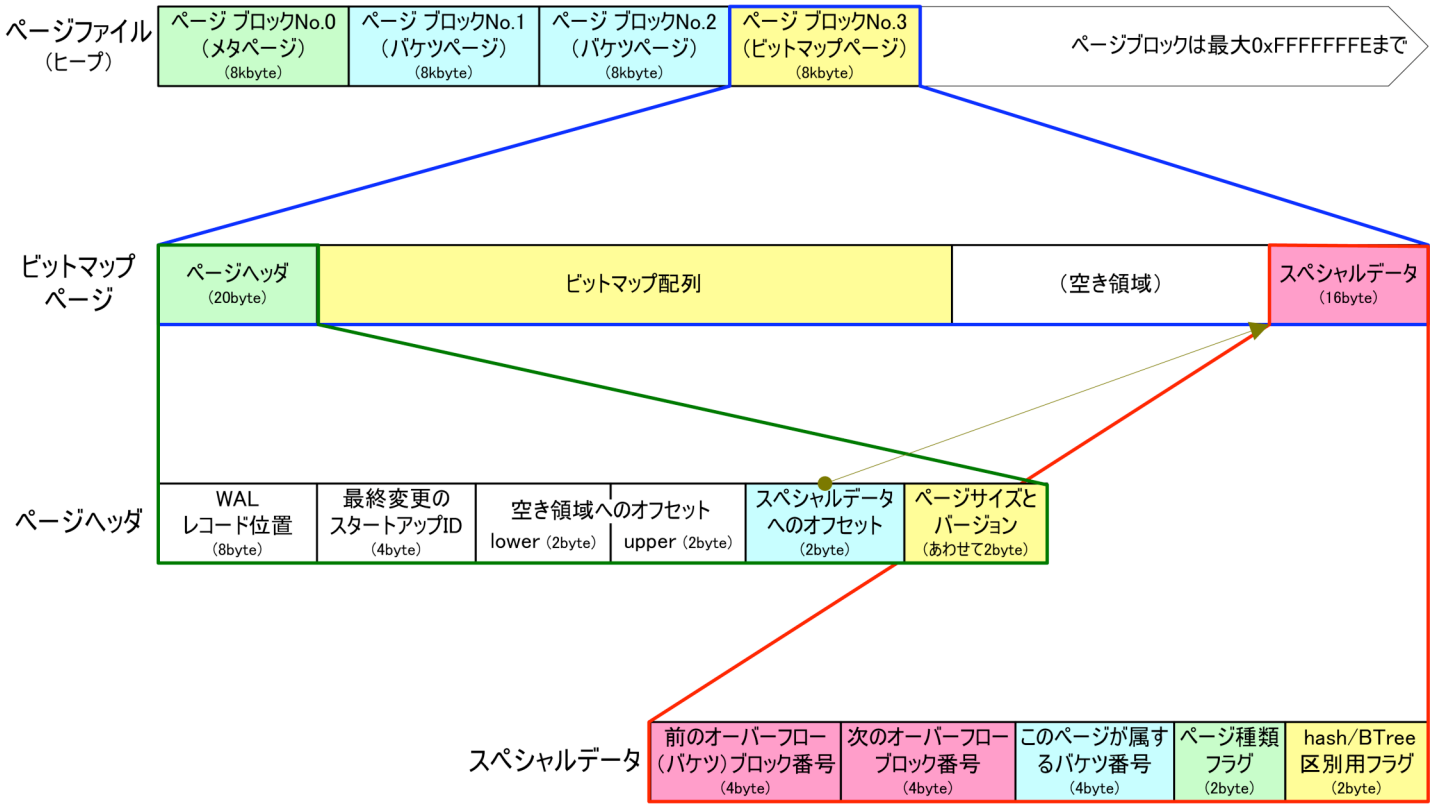
メタページ

- ファイル先頭の特殊なページ
- 固定長のメタページデータが1つ格納されている（304バイト）
  - 7.3から変更多
- HashMetaPage-Data構造体が格納されている



ビットマップページ

- 固定長のビットマップ配列の入った特殊なページ
  - 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配列(ページの再利用目的)



ハッシュインデックス関数群

- インデックスAMが実装する関数
  - amgettuple　次の有効なタプルを得る関数
  - aminsert　タプルを挿入する関数
  - ambeginscan　新規スキャンを開始する関数
  - amrescan　スキャンを再開する関数
  - amendscan　スキャンを終了する関数
  - ammarkpos　現在のスキャン位置を記録する関数
  - amrestrpos　記録したスキャン位置を復元する関数
  - ambuild　新規インデックスを構築する関数
  - ambulkdelete　バルク削除関数
  - amcostestimate　インデックススキャンのコストを推定する関数
  - amvacuumcleanup　VACUUM後クリーンアップ関数
（7.4で新設、hashにはない）
- 詳しくは別資料で...

スキャンアルゴリズム

`_hash_first()`

1. メタページを共有ロック
   - バケツが分割されないようにするlmgr locks
     （7.3以前には無かった。デッドロック検出のための措置。）
2. メタページを読みそのバッファを共有ロック
   - こちらはLWLock
3. ハッシュキーに相当するバケツ番号の計算
4. メタページバッファのロック解放
5. 対象バケツページの共有ロック
   - このバケツが分割されたり縮退されたりしないように
6. メタページのロック解放
7. リード要求毎に
   1. バケツページのバッファを共有ロック
   2. 必要に応じて次のページに移る
   3. タプル取得
   4. バケツページのバッファのロック解放
8. スキャンの最後に対象バケツページのロック解放

挿入アルゴリズム

`_hash_doinsert()`

1. メタページを共有ロック
2. メタページを読みそのバッファを共有ロック
3. ハッシュキーに相当するバケツ番号の計算
4. メタページバッファのロック解放
5. 対象バケツページの共有ロック
6. メタページのロック解放（ここまではスキャンと一緒）
7. バケツページを読みそのバッファを排他ロック
8. もしそのバッファが満杯なら、ロックを解放し、バケツチェーンをたぐって次のページを読んでバッファを排他ロック
   - 必要に応じて繰り返す
   - もし空きスペースが見つからなければオーバーフローページ追加
9. タプル挿入
10. バケツページを書き出しそのバッファのロック解放
11. メタページを読みそのバッファを排他ロック
12. タプルカウンタを増やし、ページ分割の必要性があるかチェック
13. メタページを書き出しそのバッファのロック解放
14. もし必要ならページ分割

バケツページ分割アルゴリズム

`_hash_expandtable()`

1. メタページを排他ロック
   - 分割する権利の証
2. メタページを読みそのバッファを排他ロック
3. 分割が依然として必要かチェック
   - もし分割の必要性がなくなっていたらロックを解放し終了
4. どのバケツを分割するか決定
5. 古いバケツページに対して他のスキャンがないか確認
   - あったら、そのロック解放を待たずにメタページロックを解放し終了。デッドロック回避。
6. 新しいバケツページに対して排他ロック取得を試み、失敗しないことを確認
7. メタページを更新し、新しいバケツ数を反映する
8. メタページを書き出しそのバッファのロック解放
9. メタページのロック解放
   - これで他のバケツへのアクセスが実行可能になる
10. バケツの分割を実行し、必要に応じてタプルを移動する
11. 古いバケツページと新しいバケツページへのロック解放

オーバーフローページ取得アルゴリズム

`_hash_getobflpage()`

1. メタページを読みそのバッファを排他ロック
2. ビットマップページ番号を取得
3. メタページバッファへのロック解放
4. ビットマップページを読みそのバッファを排他ロック
5. 空きページを探す（ビットが0のもの）
6. もし見つかったら
    1. ビットを立てる
    2. ビットマップページを書き出しロックを解放
    3. メタページを読みそのバッファを排他ロック
    4. 空きページ番号を更新
    5. メタページを書き出しロック解放
    6. ページ番号を返却し終了
7. もし見つからなかったら
    1. ビットマップページへのロック解放
    2. 次のビットマップページを探し、4.に戻る
    3. 全てのビットマップページを探しても見つからなかったら、新しいオーバーフローページを追加する

オーバーフローページ解放アルゴリズム

`_hash_freeobflpage()`

1. バケツチェーンのリンクから解放するページを外す
2. メタページを読みそのバッファを排他ロック
3. 解放するページが該当するビットマップページ番号を取得
4. メタページバッファへのロック解放
5. ビットマップページを読みそのバッファを排他ロック
6. ビットを更新
7. ビットマップページを書き出しロックを解放
8. メタページデータのうち、空きページ番号の情報より今回解放したページの番号の方が小さかったら
    1. メタページを読みそのバッファを排他ロック
    2. 依然としてメタページの空きページ番号情報より今回解放したページの番号の方が小さかったら、空きページ番号を更新
    3. メタページを書き出しロック解放

おわりに

- PostgreSQLのハッシュインデックスが用いているアルゴリズムについて概説した
  - アルゴリズムは単純だが、それをDBMSに載せるとなると、同時実行制御に関する周辺の処理を熟慮して設計する必要がでてくる
    ⇒GiSTはその部分をWrapする、という触れ込み
- Executorと個々のAMとの間の部分の解析が不十分なので、今後も調査していきたい