PostgreSQL の新しいバッファ戦略

2004/05/12

谷田　豊盛 (ゆたか)

tanida@sra.co.jp

バッファリングはなぜ必要なのか

- ディスクアクセスはメモリアクセスよりずっと遅い。
- 同じデータに対してアクセスが重なると、キャッシュしておけば高速にアクセス可能である。
- ブロックレベルのデータ管理

対象になるアクセスの種類 (1)

- ランダムアクセス – 特定のブロックに対する散発的なアクセス。

- シーケンシャルリード - 特定の連続したブロック群に対する連続した読み込み
  - 全件検索とか

対象になるアクセスの種類 (2)

- 局所的なアクセス - あるプロセスで短時間に同じブロックに大量のアクセス
  - Nested Loop の内側ループとか
- 頻繁な参照 - 様々なプロセスから同じブロックに頻繁なアクセス
  - -> システムテーブルへのアクセスとか

単純なアルゴリズム

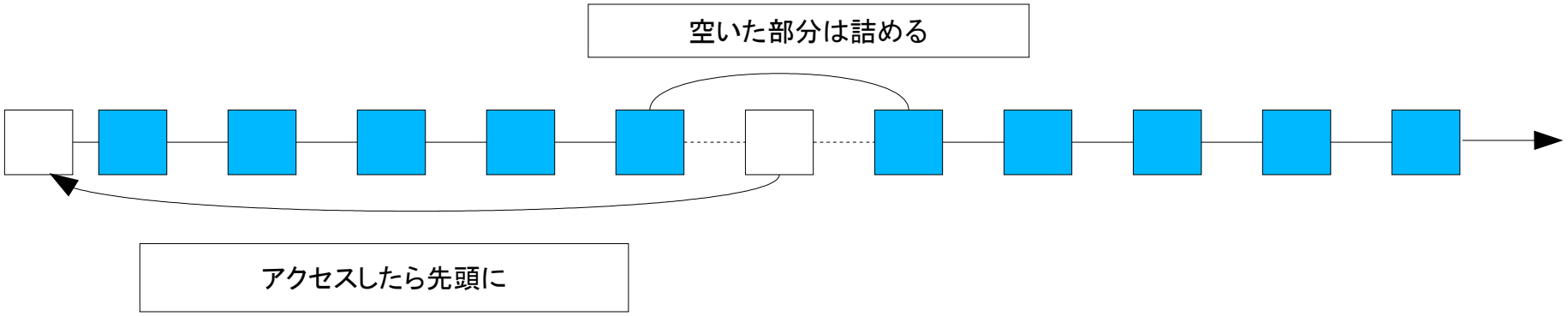
- 発想
  - キューを作って、アクセス順に並べておこう。
- 実装
  - キュー / リングバッファ。キャッシュ量一定
  - lookup の高速化のために、hashtable と組み合わせる

「１度ある事は2度ある」アルゴリズム(1)

- キャッシュに一度ヒットしたものは、きっと次もヒットするに違いないと考えるアルゴリズム。最初の一歩として妥当であると考えられる。
- 以下のような有名な物が存在する
  - LRU（PostgreSQL はこれ！）
  - CLOCK

LRU(Least Recently Used)

- 一度アクセスしたバッファは、キューの先頭に移動される。
- たった一カ所の改善で、偏りのあるランダムアクセスに対するヒット率は劇的に向上する。



PostgreSQL における LRU の実装 (1)

- もともと LRU はシングルプロセス用、競合を考慮していない
  - FreeList 操作時にはがっちりロック (BufMgrLock)
  - BufferDescriptor->cntxLock にてロック
  - BufferDescriptor->RefCount によるリファレンスカウント方式 GC
  - RefCount==0 になったら FreeList に返す

PostgreSQL における LRU の実装 (2)

- もともと LRU は同時実行は全く考慮されていない。
  - PinBuffer/UnPinBuffer でアクセス中のバッファは FreeList 上から追い出される。追い出さないと、アクセス中のバッファを FreeList から拾う事になる。
  - もちろん、キューをたどってアクセスしていない物を拾う事も可能だがコストもかかる。
  - 他のアルゴリズムに変換するときにこの点はかなりやっかい

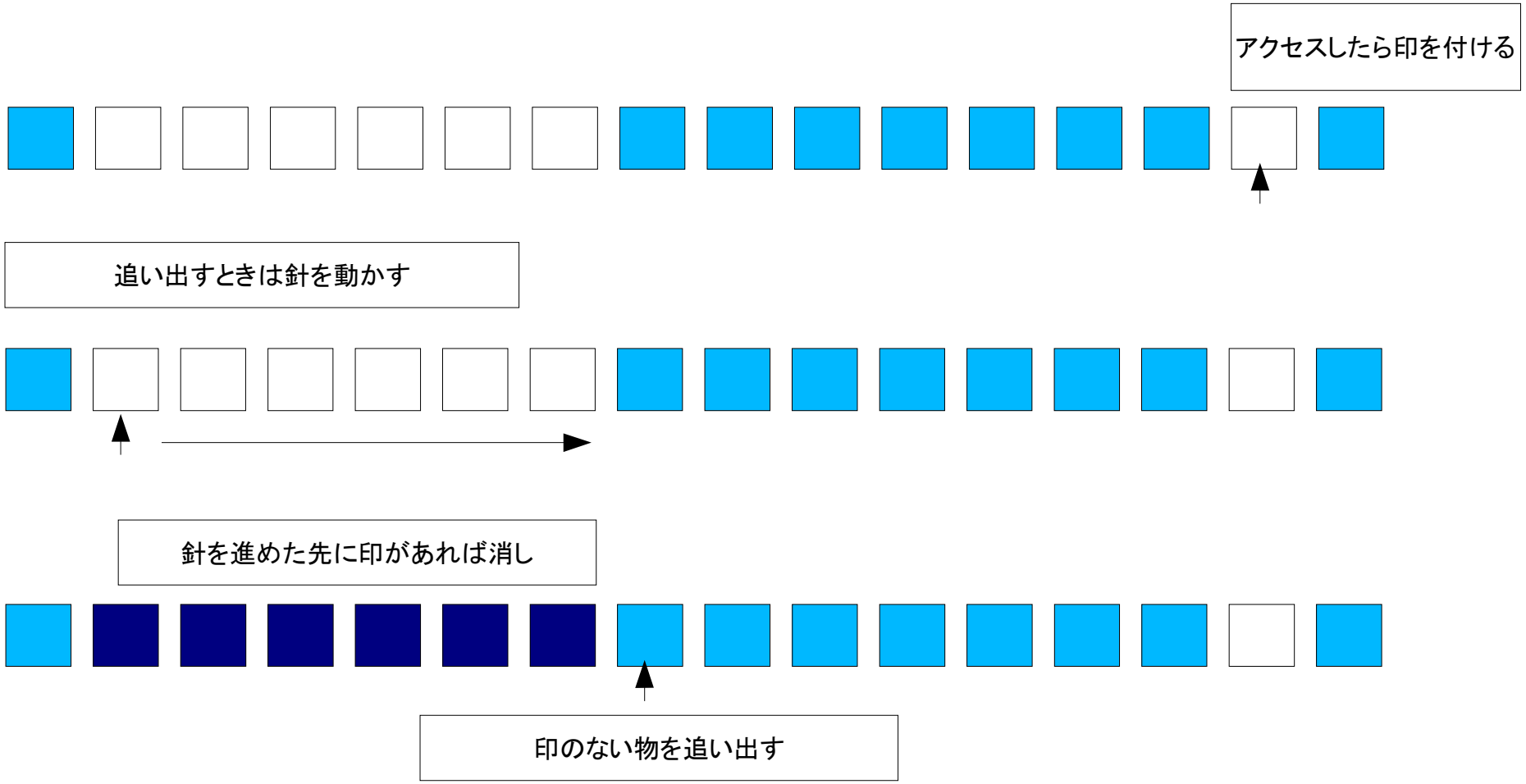
PostgreSQL における LRU の実装 (3)

- もともと LRU は読み時間を考慮していない。
  - BM_IO_IN_PROGRESS フラグや io_in_progress_lock により、読み込み中ロックするなどの管理を行っている。
- もともと LRU は書き込みを考慮していない。
  - cntxDirty フラグを立てておき、CheckPoint や追い出されるときに書き込む。

CLOCK

- 「時計の針」アルゴリズム。リングバッファとアクセスフラグの組み合わせ
  - 配列アクセスだけなので、基本的に軽い
- アクセスフラグ増減を見直すことで、アルゴリズムの性質を変える事が出来る。
  - GCLOCK/DGCLOCK などと呼ばれる。
  - が、時計の針はぐるぐる回るので、なかなか思った通りのコントロールが難しい。
  - 結局、チューニングは複雑

CLOCK(2)

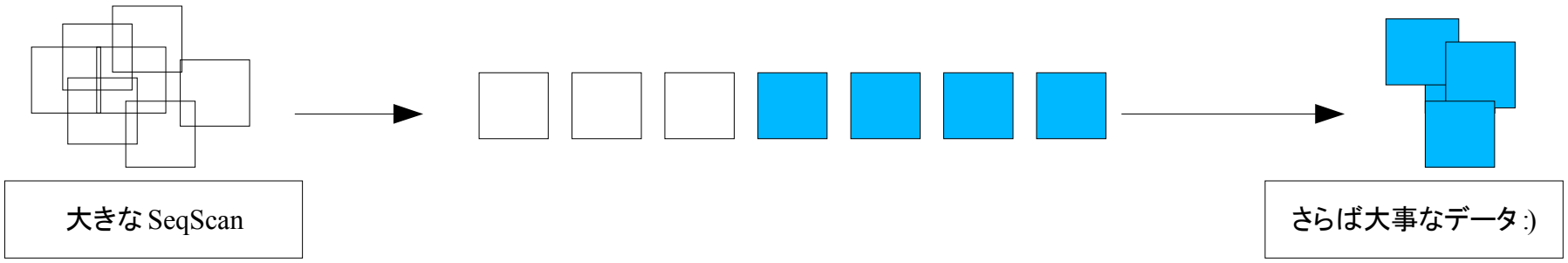


CLOCK(終)

- 下記の URL に、PostgreSQL に関する実習として、MRU とCLOCK を使った実装をせよ、と言う問題がある。
  - http://www-2.cs.cmu.edu/~olston/15-415/S04/HW/assign1.pdf
  - ここではリファレンスフラグを新設する方法について説明している。
  - 残念な事に、入れ替えるとどの程度良くなるかという事についての話は無い。

「１度ある事は2度ある」アルゴリズム(2)

- ・Sequential Scan と相性が悪い
  - 「一度ある事は二度ある」では、（最初に読み込んだデータとヒットしたデータを特別扱いしないために）巨大なシーケンシャルスキャンがすべてのデータを追い出してしまう。
  - 本来、巨大なシーケンシャルスキャンはどうせ小さなバッファ納められるはずもないので、バッファ管理的には無視出来るならしてしまうのが得策

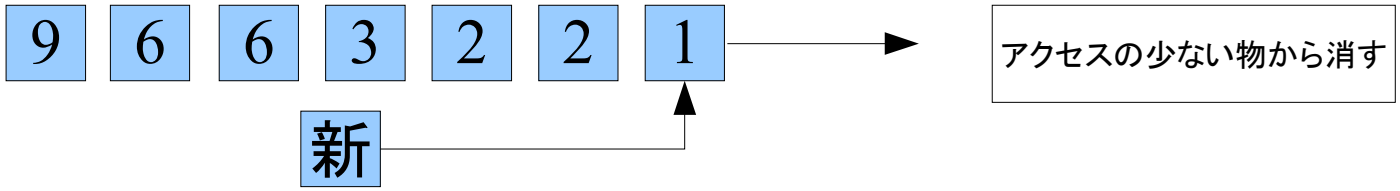


Sequential Scan に強いアルゴリズム

- 複数回アクセスのあったブロックを特別扱いするアルゴリズム
  - LFU/LRFU のような回数ベース
  - LRU-k
- シーケンシャルリードの優先度が下がるようなアルゴリズム
  - 2Q のような複数のキューの組み合わせ
  - ARC のような、アクセス状況を調べながら優先度を変える物

LFU/LRFU(1)

- LFU(Least Frequently Used)
  - 「一番たくさんアクセスがあった物はもっとアクセスされるはず」
- LRFU(Least Recency / Frequency Used)
  - LFU に加え時間とともに、古いアクセスカウントの価値を減じさせる仕組みを持つ LRFU
    - FreeBSD2.1 のディスクキャッシュで実装されているらしい。



LFU/LRFU(2)

- 弱点
  - 実装に Priority Queue が必要（遅い）
  - 局所的アクセスに弱い
    - 一度アクセスカウントが多くなってしまうと、もうアクセスしていないとしてもそのブロックはなかなか追い出されない。
    - LRFU はこれを若干解決している。

LRU-k(1)

- 「k 度あることは k 度ある」アルゴリズムの先駆けになったらしい。
- k（通例 k=2）回前のアクセスが早かった順に並べる。履歴がない（新しい）場合は一番前。
- 最悪のケースでも LRU 並のキャッシュヒット率を維持出来る。
- このアルゴリズムから見ると、前述の LRU は LRU-1

LRU-k(2)

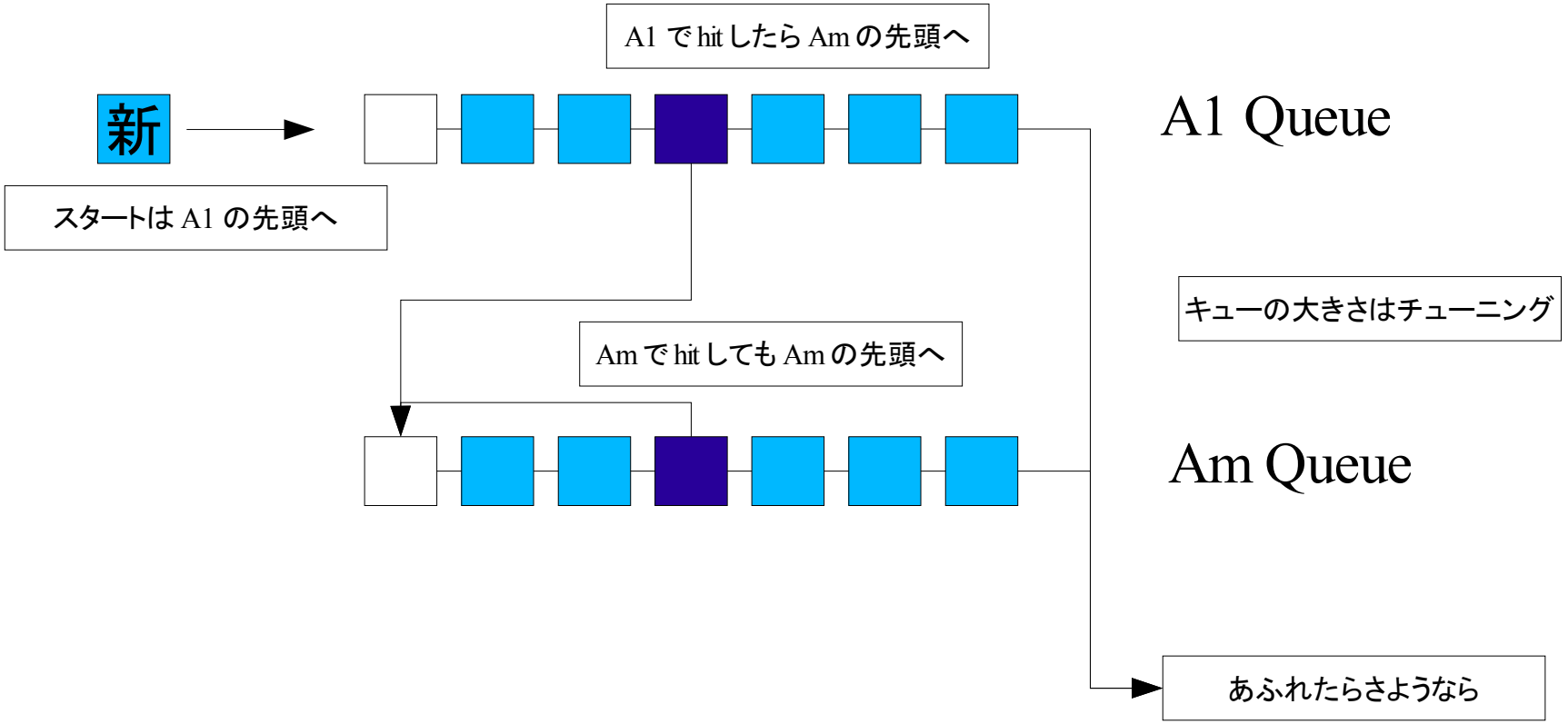
- 弱点
  - Priority Queue が必要で、ややこしい。
  - k 回分のアクセス履歴が必要。
  - バッファ読み込みのためのタイムスタンプが必要。
    - PostgreSQL の場合、おそらくXID が利用可能。
- 昔、Tom Lane が LRU-2 を実装したが、LRU より遅くなったので破棄したという話（伝聞）

2Q(1)

- 「キューの形状を工夫することでそれらしくする」アルゴリズム
- 複数のキューを組み合わせる事で、LRU-k 並のヒット率改善を実現したアルゴリズム
- アルゴリズムが簡単で LRU クラスのオーバーヘッド

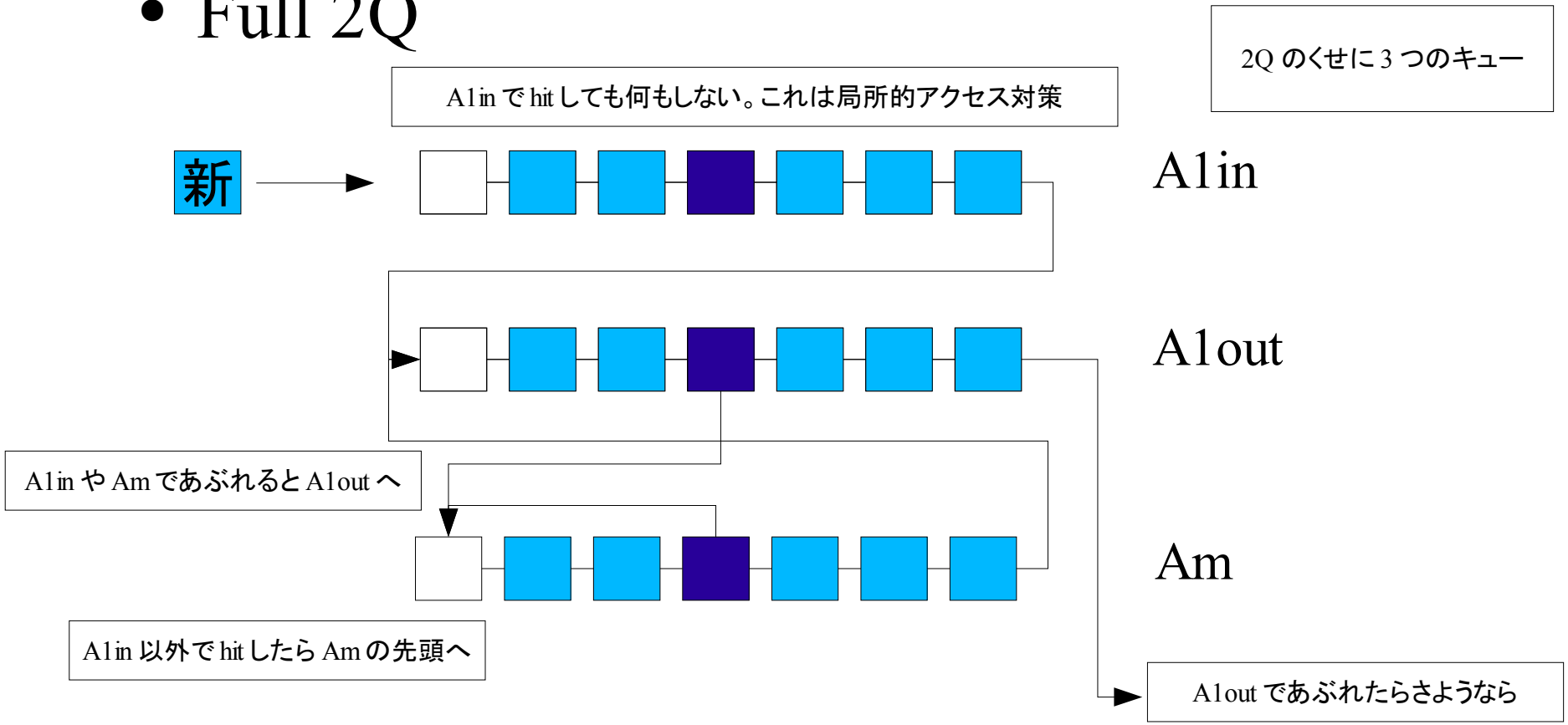
2Q(2)

- Simplified 2Q



2Q(3)

- Full 2Q


2Q のくせに 3 つのキュー
A1in で hit しても何もしない。これは局所的アクセス対策
新
A1in
A1out
Am
A1in や Am であぶれると A1out へ
A1in 以外で hit したら Am の先頭へ
A1out であぶれたらさようなら

2Q(4)

- PostgreSQL に 2Q を実装したとき
  - Full をチョイスしたが、Pin/UnPin があるので A1in は実装しなかった。
    - どちらかというと、Full と Simplified のあいのこ
  - FreeList を 2 つ作り、それを A1out,Am にした
  - BufferDesc に int のフラグを追加、キューのサイズ及びどのキューにいるかという状態監視に使い回す。(ここまでほとんど freelist.c/buf_internal.h)
  - 初期化コード (buf_init.c)

2Q(4)

- 弱点
  - キューの大きさについて、細かいチューニングが必要とされる。
    - アクセス状況によって、どの程度がよいかというのはかなり異なるようである。
  - 二度アクセスがないと、上部のキューが無駄になる
    - だから、select だけの pgbench の場合 LRU より遅くなる結果が出る事が後で判明した。

ARC(1)

- 2Q の構造を踏襲しながら、弱点をほとんど克服した優秀なアルゴリズム。
- データを保持している二つのキューと、保持していない二つのキューを組み合わせたアルゴリズム。

ARC(2)

- 登場人物 (1)
  - BufferDesc　バッファと、管理構造体
    - char *page;
    - int ARC_where;
    - ページ特定用情報 (bufferTag);
    - BufferDesc prev,next;

ARC(3)

- 登場人物 (2)
  - 各種キュー
    - (0)T1,T1Length キュー。最初に読み込んだときの格納先。データあり。
    - (2)T2,T2Length キュー。一度ヒットしないとここには入らない。データあり。
    - (1)B1,B1Length T1 からヒットせずそのまま落ちた物が入る場所。データなし。
    - (3)B2,B2Length T2 に一度でも入ったあと落ちた物が入る場所。データ無し。

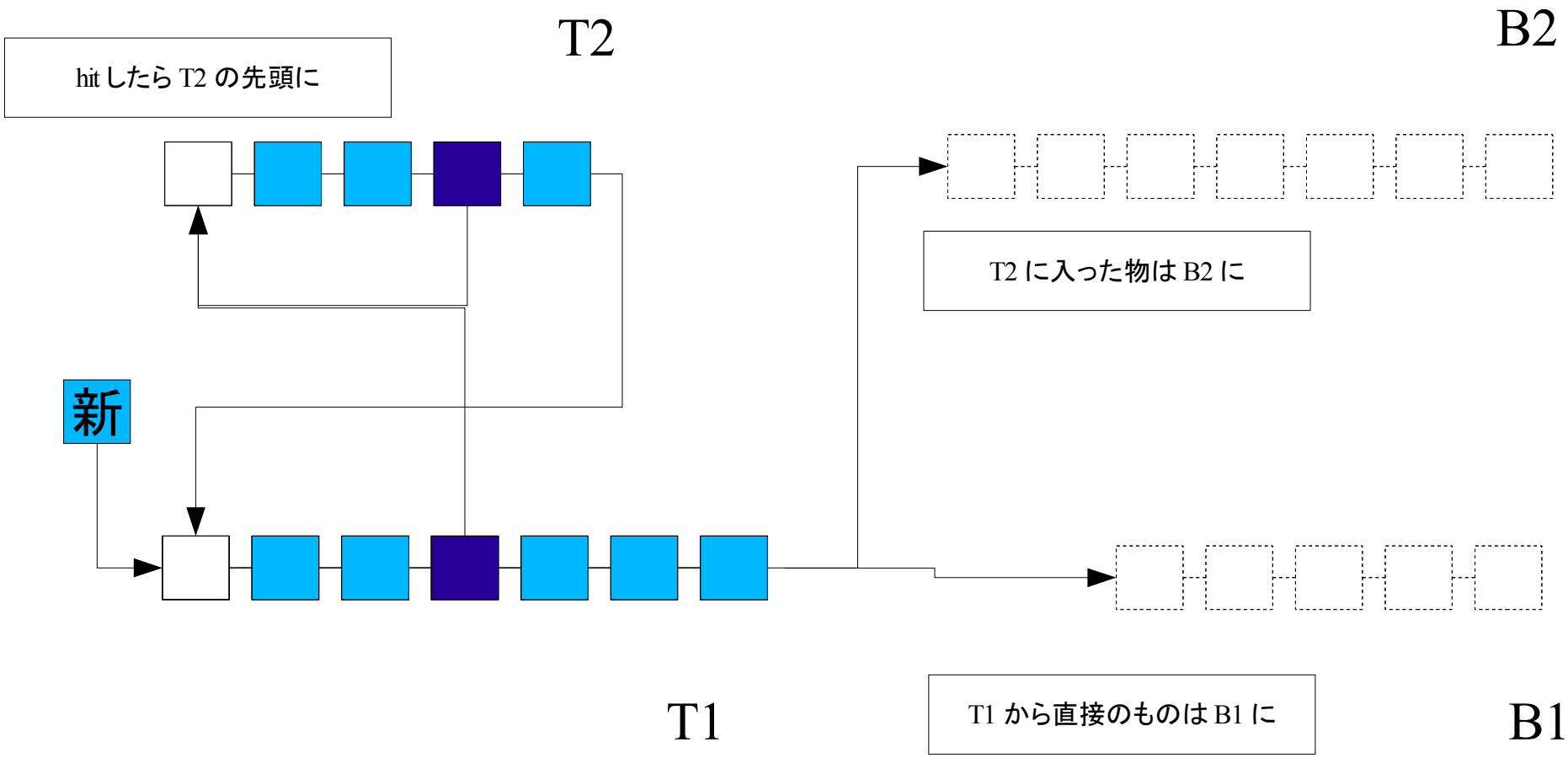
ARC(4)

- ルール
  - T1Length+T2Length = バッファサイズ
  - T1Length+T2Length+B1Length+B2Length = バッファサイズの倍
  - T1Length+B1Length = バッファサイズ

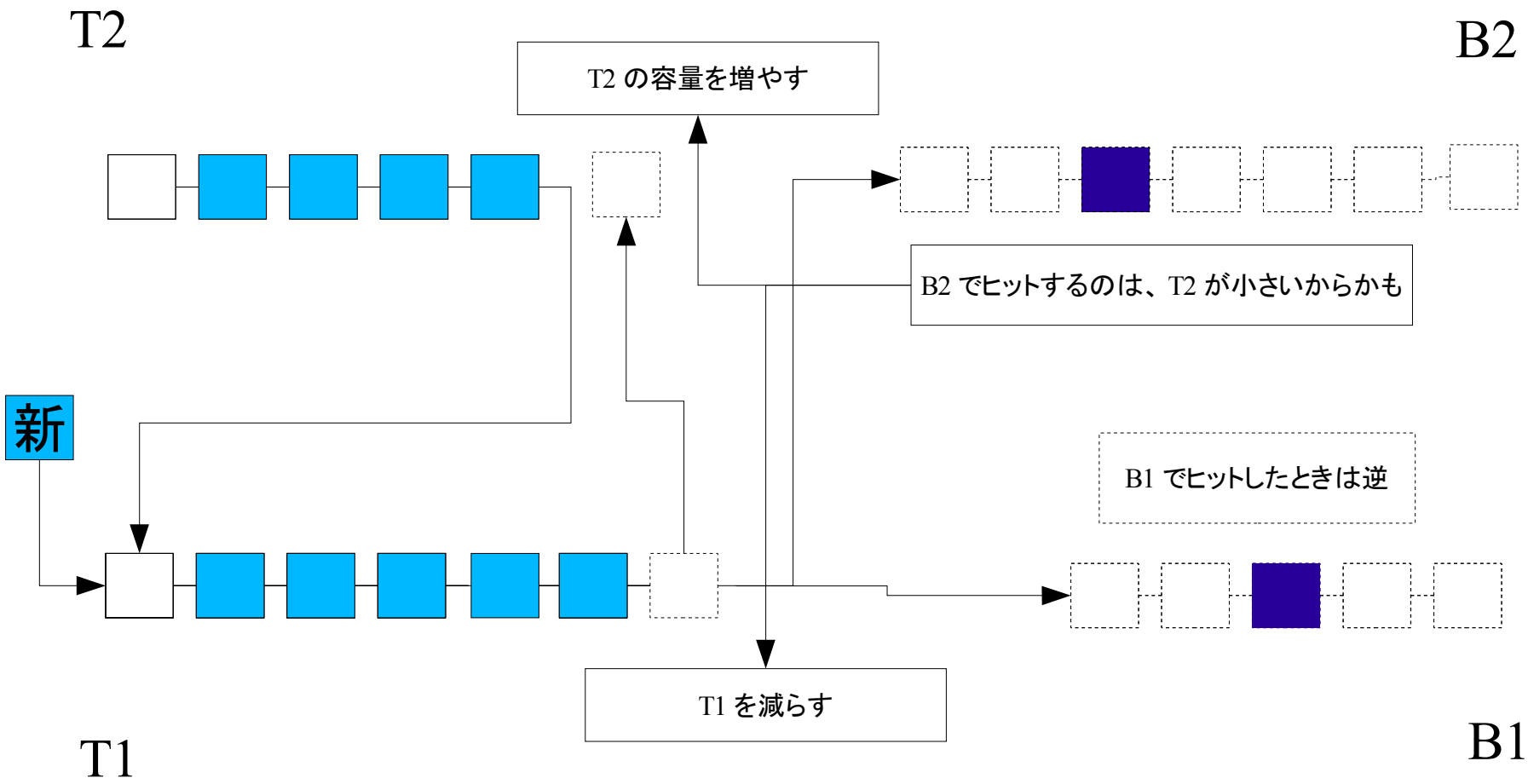
ARC(5)

- ・データ保持している部分だけを見ると、待機部分のない 2Q と同じ。
- ・データ保持していない部分からヒットした場合、T1/T2 の割合を変化させる。
  - 一回も再ヒットしなかったものが後でヒットしたなら、待つ時間が短すぎる可能性がある。
  - 数回ヒットした物が後でヒットしたなら、長期記憶に割り当てる量が少なすぎる可能性がある。
  - B1 や B2 にすらヒットしないものは、そもそもこのアルゴリズムではどうしようもない。

ARC(6)



ARC(7)


T2
B2
T2 の容量を増やす
B2 でヒットするのは、T2 が小さいからかも
新
B1 でヒットしたときは逆
T1 を減らす
T1
B1

ARC(8)

- 特徴
  - LRU からの移行が簡単
  - 設定項目の必要がない。
  - ヒント無しにシーケンシャルスキャンを自動検知し、る。
    - B1 でのヒットが多ければランダムアクセスが多いとして、T1 を大きくしヒットするまでの待ち時間を増やす。
    - B2 でのヒットが多ければヒットしないデータアクセスが多いとして、T2 を大きくしてヒットできるデータを保護

PostgreSQL/ARC(1)

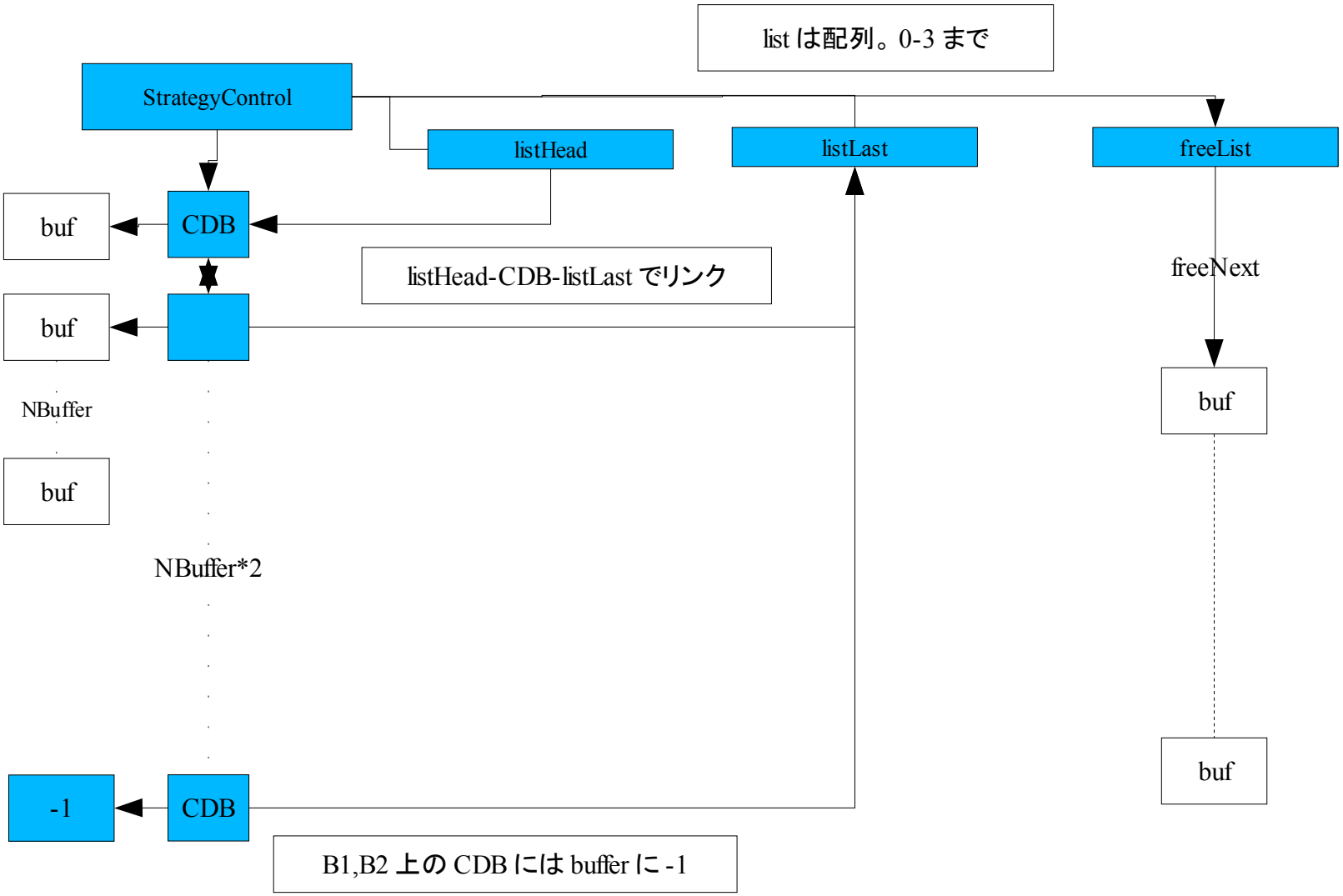
- ・バッファ戦略のためのデータ全部入り
  - typedef struct
    {
    int target_T1_size; /* What T1 size are we aiming for */
    int listUnusedCDB; /* All unused StrategyCDB */
    int listHead[STRAT_NUM_LISTS]; /* ARC lists B1, T1, T2 and B2 */
    int listTail[STRAT_NUM_LISTS];
    int listSize[STRAT_NUM_LISTS];
    Buffer listFreeBuffers; /* List of unused buffers */

    long num_lookup; /* Some hit statistics */
    long num_hit[STRAT_NUM_LISTS];
    time_t stat_report;
    /* Array of CDB's starts here */
    BufferStrategyCDB cdb[1]; /* VARIABLE SIZE ARRAY */
    } BufferStrategyControl;

PostgreSQL/ARC(2)

- キャッシュのための情報を格納している部分、CDB
  - typedef struct
    {
    int prev; /* list links */
    int next;
    short list; /* ID of list it is currently in */
    bool t1_vacuum; /* t => present only because of VACUUM */
    TransactionId t1_xid; /* the xid this entry went onto T1 */
    BufferTag buf_tag; /* page identifier */
    int buf_id; /* currently assigned data buffer, or -1 */
    } BufferStrategyCDB;
    - BufferTag は BufferDesc と同じになるが、相互参照はない

PostgreSQL/ARC(3)



PostgreSQL/ARC(4)

- それ以外の変更のポイント
  - もはや PinBuffer -> UnpinBuffer で、リストからの切り離しは行っていない。
  - freeList は完全なリストではない。もし freeList が空になら、代わりに T1->T2 の順にスキャンして、未使用の物から利用している。
  - このキャッシュの直近のアクセス履歴 xid を持っている。同じ xid からのアクセスでバッファ位置の LRU 移動は無い。
    - 局所的なアクセスが多発した場合に、キュー操作のオーバーヘッドを防いでいる

バッファにおけるヒント

- Sequential Scan を避けると言うが、executor は Seq Scan がいつ起こるか一番よく知っている。
- Sequential Scan 以外の場合でもバッファの適切な使用量というものがある。
- ヒントを出す事で、より重要なバッファを突き止める事が容易になる。

ヒントの種類

- 手動
  - 精通していないと上手く行えい。
- Relation の種類
  - システムテーブル、インデックス、・・・
  - 常にアクセスが多いとは限らない。
  - システムテーブルは別枠 (src/backend/util/cache/ 以下) がすでに存在している。。
- アクセスの種類
  - 様々な分類法が研究されている。

PostgreSQL とヒント

- StrategyHintVacuum(bool)(freelist.c)
  - PostgreSQL における最初の buffer hint 実装
  - vacuum は、後で使う見込みの低い Seq Scan しか行わないのが分かり切っている。
  - そこで、vacuum に利用するバッファは、FreeList の最前列！（最後尾ではない）に置くことでバッファ管理を最適化する。また、キューの移動もしない
  - これによる vacuum 中の性能向上は 20% と言われている。

bgwriter

- 7.5 で新設される新しいプロセス
- 適当なタイミングで freeList に入っている dirty なバッファをディスクに書き込む。これによって CheckPoint 時の負荷を軽減出来る。
- やっている事は、待つ・適当にバッファを flush する・開いているファイルを閉じるなど。
- postmaster.c/bufmgr.c#BufferBackgroundWriter を参照の事。

参考文献等

- Wenguang Wang ,”Storage Management in RDBMS” , 2001
- WOLFGANG EFFELSBERG and THEO HAERDER,“Principles of Database Buffer Management” , 1984
- Donghee Lee,Jongmoo Choi,Honggi Choe,Sam H. Noh,Sang Lyul Min and Yookun Cho , “ Implementation and Performance Evaluation of the LRFU Replacement Policy”, 1997
- Elizabeth J. O’Neil and Patrick E. O’Neil, “The LRU–K Page Replacement Algorithm For Database Disk Buffering” , 1993
- Theodore Johnson and Dennis Shasha , “2Q:A Low Overhead High Performance Buffer Management Replacement Algorithm” .1994
- Nimrod Megiddo and Dharmendra S. Modha , “A Simple Adaptive Cache Algorithm Outperforms LRU” , 2003
- Nimrod Megiddo and Dharmendra S. Modha ,” One Up on LRU “. 2003