データベースシステム　99−31
（１９９４．７．２２）

フラッシュメモリ向きデータ構造におけるデータ管理方式とその性能評価

高倉弘喜　上林彌彦
京都大学工学部

主記憶データベースでは耐障害性を高めるため主記憶データを定期的に二次記憶にバックアップする必要がある。磁気記憶への保存ではCPUの主記憶アクセスに及ぼす影響が大きいため速度の速いフラッシュメモリの利用を検討する。フラッシュメモリはディスクと特性が異なる点を生かしたデータ構造としてＦ木を導入し、さらにフラッシュメモリをバックアップ媒体だけでなく通常の二次記憶として利用できる方式について提案する。フラッシュメモリの利用期間を決定する消去をできるだけ避けて上書きを行なうデータ管理方式について述べる。また本方式によりフラッシュメモリの利用期間が実用的であることをシミュレーションによって確かめた。

A Data Management Method for Flash Memory and Its Performance Evaluation

Hiroki TAKAKURA　Yahiko KAMBAYASHI
Faculty of Engineering, Kyoto University

In order to realize highly reliable main memory databases, main memory data should periodically be checkpointed to archive storages. Since flash memory can realize faster access speed than disks, utilization of flash memory as archive storages can realize effective checkpoint operations. Furthermore, a requirement to used flash memory as archive and secondary storages is arisen. In this paper a data management method to realize such storages is discussed. A method which utilizes special data structure, F-tree, and overwrite technique without erase operations is also discussed. Simulation results show that our method can practically utilize flash memory for secondary storages.

1 はじめに

主記憶データベースでは全データを主記憶に常駐させることにより極めて高速なデータ処理が行なえる。また、近年の半導体素子の大容量化と低価格化に伴って主記憶データベースを利用するシステムも実現されつつある。しかし、主記憶は一般に揮発性であるため電源障害やシステム異常などのハードウェア的な障害によってそのデータを失うおそれがある。また、ソフトウェアのバグなどによってデータが破壊されうる。このため、定期的に主記憶データを二次記憶にバックアップする必要がある。

二次記憶媒体としてはディスクが一般的であるがアクセス速度が主記憶に比べかなり遅い。このためページ単位に高速な乱アクセスができ、データを電源供給なしに保持できるフラッシュメモリがディスクに代わるものとして期待されている。しかし数ビットの書き換えであっても、その前に数kBのブロックを数ミリ秒かけて消去する必要があるため、読み出しに比べアクセス時間が長くなる。また消去回数に百万回程度の上限があり、メモリ全体の消去回数を一様にするデータ管理方式も必要である。

著者等は主記憶データのバックアップ媒体としてフラッシュメモリを利用する方式について提案した[3][4]。本方式は検査点間隔ごとに更新を受けた主記憶ページをフラッシュメモリにバックアップする。フラッシュメモリのデータ構造は主記憶のものと同じであり、データ更新の度にフラッシュメモリを書き換えると消去が多発するためバックアップ以外の用途には利用できない。またフラッシュメモリを二次記憶として利用するため、著者等はフラッシュメモリの構造に注目した上書きを行なうデータ構造(F木)について提案した[5][6]。本稿では、F木を主記憶データのバックアップと二次記憶の双方に使えるように改良した方式について述べる。また、F木を用いた場合のフラッシュメモリの利用状況をシミュレーション実験により解析し、1秒間に4kBの1ブロックをx回更新する場合、$35/x$年間フラッシュメモリが利用できることを示す。

2 基本的事項

[システム構成]

全データベースを主記憶に常駐させるのはコストや物理的な制約から考えて現実的でなく、良く使われるホットなデータのみを主記憶に常駐させる方が一般的である。このようなシステムでは二次記憶アクセスが必要であるが、ディスクでは主記憶に比べ遥かに遅いためCPUの主記憶アクセスが大きな影響を受ける。このため数μsの待ち時間の後はDRAM並のアクセス速度を持つフラッシュメモリの利用について検討する。

主記憶に常駐するデータが時間と共に変化する場合、フラッシュメモリをバックアップ記憶としてだけではなく二次記憶としても利用する必要がある。ホットでないデータはトランザクションの確定時あるいはグループ確定時に二次記憶に書き込まれる。ホットなデータも同様に書き込もうとすると、二次記憶への保存が終了するまで主記憶データがアクセスできないためシステムの性能が著しく低下するおそれがある。

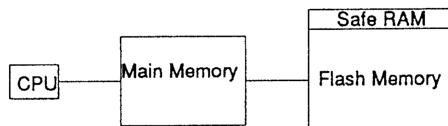


図 1: システム構成図

そこで、本稿のシステムは図1の様にCPU、主記憶、二次記憶(フラッシュメモリ)の他にバッテリバックアップされた不揮発主記憶[1]で構成される。ホットな更新データはトランザクションの確定時あるいはグループ確定時に不揮発主記憶にまず書き込まれ、その後二次記憶に保存されるものとする。不揮発主記憶は二次記憶の一部であると考えるとホットなデータとそうでないデータとの扱いに基本的な違いはなく、全データが主記憶に常駐していると一般化できる。そこで本稿ではホットなデータのみに注目し、簡単のためにデータベース全体が主記憶に常駐しているものとする。

また、F木はUNIXなどのように最新のものを主記憶に置き、定期的に二次記憶のものを更新する。F木の更新の前にシステム障害が発生するおそれがあるので、二次記憶上のF木からもフラッシュメモリ上の最新データの検索が可能でなければならない。

[フラッシュメモリ]

本稿のシステムでは、NAND型のフラッシュメモリを利用する。フラッシュメモリは図2のようにレジスタを介してメモリセルにアクセスする。このためアクセスはレジスタサイズ(1ページに相当)で行なわれる。以下に読み出しおよび書き込み操作の手順を示す。

[読み出し操作]

(1) 読み出し転送 : 1ページ分のデータをメモリセルからレジスタに転送($15\mu s$)

(2) 読み出し開始 : レジスタ中の任意の番地から連続読み出し(数十 ns/B)

[書き込み操作]

(1) 消去 : 希望ページを含むブロックまたはチップを消去(6〜10ms)

(2) 書き込み : データをレジスタの先頭番地から連続書

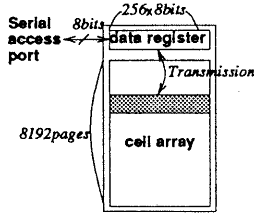


図 2: フラッシュメモリの構造

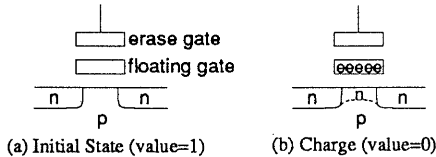


図 3: メモリセルの構造

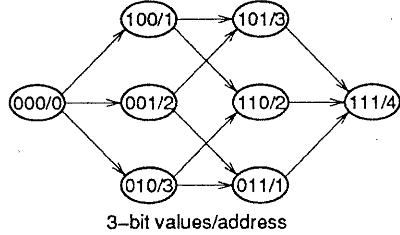


図 4: 消去の不要なアドレス割り当て (3bit の場合)

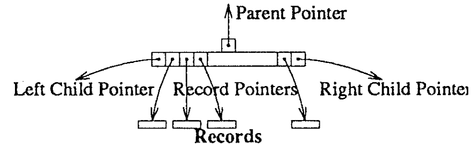


図 5: F 木の節点の構造

き込み (数十 ns/B)

(3) 書き込み転送 : レジスタデータをメモリセルに転送 (35μs)

(4) 自己検証 : フラッシュメモリは自動的にレジスタデータとメモリセルデータを比較 (10μs)

フラッシュメモリのメモリセルは図3に示すようにEEPROMとほぼ同じ構造を持つ。初期状態1のセルのフローティングゲートに電荷を蓄えると0を書き込んだことになる。1に戻すには消去ゲートに電圧を加え電荷を強制移動させる必要があり、この操作が消去である。消去によりフローティングゲート周辺の絶縁性が徐々に劣化するため消去回数には限界があり、最新のもので100万回程度である。一般に書き込み前の消去は必要であるが、EEPROMと同様にページ中の1であるビットを0にするだけなら消去せずに上書きできると考えられる。しかし0のセルに1を上書きしても0のままであり、必ず消去が必要となる。著者等はこの一方向の上書きを応用するデータ管理構造について提案した。

以下では議論を簡単にするため0と1を入れ換えて考える。(0010) の状態のセルを (0100) で書き換えると (0110) となる。これはビット列に対する論理演算のORを行なっていることを表す。ビット列のうち1ビットでも $1 \rightarrow 0$ の遷移を起こす場合は必ず消去が必要となるので、以下に述べる上書きが可能なビット列は (**1*) となる (*は0または1を表す)。

[上書き操作]

(1) 読み出し : フラッシュメモリからデータを主記憶に読み出す

(2)OR 演算 : 読み出したデータと更新データのOR演算を実行 (幾つかのビットが $0 \rightarrow 1$ に遷移する)

(3) 書き込み : 演算結果をフラッシュメモリに書き込む

フラッシュメモリは書き込み後に自己検証を行なうため、(1) 及び (2) の操作が必要となる。

上書きの応用として3ビットを符合化して5つのアドレスを指定する方式を図4に示す。初期状態 (000) はアドレス0を示し、1ビットずつ上書きすると、000/0→100/1→110/2→111/4 と遷移できる。この遷移ではアドレス3は指定できないが他の遷移で指定できる。

3 F-木の概要

F木の1節点の構造を図3に示す。

3.1 一般的な木の問題点

[木のバランス]

ディスク上の検索木では左右の部分木の深さが大きく異なると検索のオーバヘッドが大きくなるものがありディスクアクセスの効率が低下する。このためバランス木では深さの差が1節点より大きくならないように管理して検索時間のばらつきを小さくしている。これに対してフラッシュメモリのアクセスは極めて速いため、部分木の深さに多少の差があっても検索時間はそれほど大きくならない。但しキーが余りにも一方の部分木に偏ってしまうと検索時間が大きくなるため、深さの差はある程度以下にとどめる必要がありバランス操作が行なわれる。

[節点更新]

図 6: ポインタ部の構造

F木は基本的にはB木と同様な構造であり、節点からレコードへのポインタを設ける。但しB木と同様な操作を行なうと、ポインタはレコードの追加および削除の度に更新される。さらに後述するように、フラッシュメモリではレコードを更新後にバランス操作のため木の再構成の必要がある場合、節点間のポインタを変更しなければならない。これらの操作の度に節点ブロックを消去すれば短期間でフラッシュメモリが利用できなくなる。

[レコード更新]

レコードは更新の度にビット列が大幅に変更されるため、図4のような符合化による上書きはできない。また複数のビットで0、1を表しても更新頻度に偏りがある場合記憶効率が低下する。さらにレコードの追加、更新および削除の度にレコードブロックを消去すれば、数百Bの変更のために4kBのブロック全体を書き直す時間がかかり望ましくない。従ってF木では、レコードの更新は新たに未使用のページを探して書き込む必要がある。

[レコード更新と節点の関係]

節点からレコードへのポインタがレコードのアドレスを指す場合、データ更新の度にポインタを付け換える必要がある。これは節点ブロックの消去回数を増加させるため望ましくない。そこでポインタは単にレコードを含むブロックのアドレスを指すものとし、同一ブロック内でのアドレス移動に対応できるようにする。しかしホットなレコードを同一ブロック内に保存し続けることはできないし、他のブロックに移る度にポインタを付け換えることも望ましくない。

3.2 F木の基本的構造

[節点部の構造]

本稿のシステムでは1節点はフラッシュメモリの1ページを利用するものとする。各節点は以下の情報を持つ。

- 親節点へのポインタ
- 左および右の子節点へのポインタ
- 節点が管理するレコードのキーの値およびそのレコードへのポインタ

節点間のポインタ部の構成を図6に示す。ポインタ部は以下に示すポインタ領域とフラグからなる。

- $P_1, P_2, \cdots, P_i$: 親節点へのポインタ領域群
- F_p(i ビット): 親節点へのポインタの選択フラグ

  0···00: 親節点は存在しない (根である)

  0···01: P_1 が有効

  ······

  1···11: P_i が有効
- $L_1, L_2, \cdots, L_j$: 左の子節点へのポインタ領域群
- F_l(j ビット): 左の子節点へのポインタ選択フラグ (F_p と同様)
- $R_1, R_2, \cdots, R_k$: 右の子節点へのポインタ領域群
- F_r(k ビット、通常 $j = k$): 右の子節点へのポインタ選択フラグ (F_p と同様)

このように $P_1 \rightarrow P_2 \rightarrow \cdots$ とポインタを未使用領域に移し対応するフラグのビットを1つずつ立てていくことにより、消去無しに上書きすることができる。

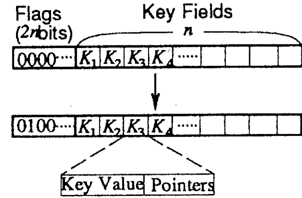


図 7: キー部の構造

また図7にキー部の構成を示す。キー部では1キー領域に対して2ビットのフラグを割り当てる。フラグは上書きにより以下のようにキー領域の状態を示す。

00: 初期状態 (キー領域は未使用)

01: キー領域は利用中

11: キー領域は無効 (利用不可)

例えば、図7のフラグ (0000···) は K_1 と K_2 は初期状態にあることを示しており、K_1 と K_2 はキーの保存に利用できる。この節点にキーを追加すると、次のフラグの状態 (0100···) のようになり、K_1 がキーの値とそのレコードへのポインタを保存するために利用中であることを示している。さらに新しいキーを保存する場合、K_2 に書き込み、かつ、フラグを 0101··· とすることにより消去無しに上書きができる。後述するようにキー領域からのポインタはレコードを含むブロックのアドレスを指す。ポインタの書き換えではビット列で $1 \rightarrow 0$ に変化するものがあるため単に上書きすることはできない。このためページサイズから考えて、1キー当たり2個のポインタ領域を割り当て、2回までのポインタ書き換えに対応する。

[レコード部の構造]

本稿ではレコードサイズはページサイズと同じものとして考える。レコードブロックの構造を図8に示す。節点のキー領域の場合と同じく、1レコードページに2ビットのフラグを割り当てる。フラグが示すページの状態はキー領域の場合と同じである。

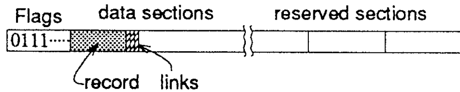


図 8: レコードブロックの構造

節点からレコードへのポインタ変更をできるだけ避けるため図8に示すように各レコード毎にリンクを設け、レコードが他のブロックに移る場合次のブロックを指すようにする。この方式ではアクセスが遅くなるため、ある段数以上にリンクを繰り返さないように制限を設ける。

物理的な構造としては、レコードブロックの先頭ページにフラグ部、キー値とそれに対応するリンクなどそのブロックの制御情報を置く。256B の1ページの内フラグやキー値が60B程度の部分を占める。このため残りの容量から考えて、1レコードブロックあたり3個のリンク領域を割り当て3回までのリンク書き換えに対応する。従ってリンクの繰り返しを軽減することができる。制御情報ページに続いてレコードページが存在する。本方式では、レコードページへのアクセスは幾つかのブロックの先頭ページと最新レコードページだけでよく、ディスクに比べると数百倍早い。

4　F木上の操作

ここではF木の操作アルゴリズムについて述べる。m を1節点中に存在可能である有効なキーの数とする。また、M_{node}、M_{record} をそれぞれ1ブロックに存在可能な無効節点ページおよび無効レコードページの数とする。

データ追加のアルゴリズムを図9に示す。

[データ追加]

F木を検索して最小値 < 追加キー値 < 最大値である節点を見つける

if 節点が見つかった

　節点にキーを追加

　if 節点中の有効なキーの数 > m (図9(b))

　　左の子孫で最大のキー値を持つ節点 V_L を見つける

　　if 見つからない (左の子孫が存在しない)

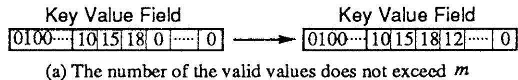
　　　新たに節点を作り、左の子孫とする

　　右の子孫で最小のキー値を持つ節点 V_R を見つける

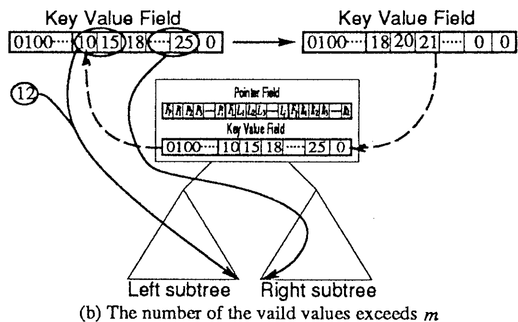
　　if 見つからない (右の子孫が存在しない)

　　　新たに節点を作り、右の子孫とする

　　キーをソート



(a) The number of the valid values does not exceed m



(b) The number of the vaild values exceeds m

図 9: 追加操作 (キー値 12 を追加)

　　小さい方から $\lfloor\frac{m}{3}\rfloor$ 個を V_L に追加

　　大きい方から $\lfloor\frac{m}{3}\rfloor$ 個を V_R に追加

else (節点が見つからない)

　最後に到達した節点に追加

節点の全てのキー領域が利用できなくなると別ブロックに節点を移動させなければならない。これに対して、節点に存在可能な有効キーの数 m を設けることによりキー領域に未使用領域が存在する確率が高くなり、更新操作の際に1つのキー領域が利用不能になっても同一ページの未使用領域に移動するだけで良い場合が多くなる。このため、更新により節点が別のブロックに移る確率を低くすることができる。また、図9(b)から分かるように節点 V_L は左の部分木の最も右側にある節点で、V_R は右の部分木の最も左側にある節点を表している。

データ更新のアルゴリズムを以下に示す。

[データ更新]

F木を検索してキーを含む節点を探す

if 同一レコードブロックで未使用ページ存在

　未使用ページに上書き

　旧ページのフラグを「無効」にする

else

　別レコードブロックの未使用ページに上書き

　旧ブロックから新ブロックへのリンク設定

　旧ページのフラグを「無効」にする

　if 旧ブロックの無効ページ数 > M_{record}

　　レコードブロック消去

1ブロックに存在できる無効レコードページ数に上限 M_{record} を設けることにより、レコードブロックのほとんどを無効ページが占有することが避けられ、二次記憶の利用効率が改善できる。

データ削除のアルゴリズムを以下に示す。

[データ削除]

F木を検索してキーを含む節点を探す
節点のキーに対応するフラグを「無効」にする
if 節点に有効なキーが存在しない
　if 節点に左右とも子孫が存在しない
　　旧節点を「無効」にする (削除する)
　　if 旧節点を含むブロックの無効節点数 $> M_{node}$
　　　節点ブロック消去
　if 節点に左右の1方だけ子孫が存在
　　親節点からのポインタを子につなぐ
　　旧節点を「無効」にする (削除する)
　　if 旧節点を含むブロックので無効節点数 $> M_{node}$
　　　節点ブロック消去
　if 節点の両方に子孫が存在
　　左の子孫で最大のキー値を見つける
　　右の子孫で最小のキー値を見つける
　　これらのキーを節点に追加

レコードページと同じく1ブロックに存在できる無効節点数に上限 M_{node} を設けることにより、二次記憶の利用効率が改善できる。

データ更新および削除で使つかわれているブロック消去のアルゴリズムを以下に示す。

[レコードブロック消去]

while 旧ブロックでページを利用中のレコードが存在
　利用中のレコードを他のブロックの未使用ページに上書き
　レコードのフラグを「無効」にする
　if 節点からのポインタが上書き可能
　　レコードポインタを上書き
　else
　　if 節点のキー領域に未使用部分が存在
　　　キーを未使用領域に移動
　　　元のキー領域を「無効」にする
　　else
　　　新しい節点を生成
　　　旧節点から有効なキーを新節点へコピー
　　　旧節点を「無効」にする
　　　if 旧節点を含むブロックの無効節点数 $> M_{node}$
　　　　節点ブロック消去
旧レコードブロック消去

[節点ブロック消去]

while 利用中の節点が存在
　新しい節点生成
　旧節点から必要なものをコピー
　旧節点を「無効」にする
旧節点ブロック消去

これらの操作で節点間のポインタを付け換えている間は、その節点以降の子孫は検索できない。ページ数の関係で詳細は述べないが、バランス操作は木の葉の部分から根へ向かって各節点から見た部分木の深さの差が1を超えないように順次バランスをとる。再構成中はF木を利用できないし、かなりの時間がかかる。このためフラッシュメモリ上のF木が利用できない間は主記憶にある検索木を利用する必要がある。付け換えや再構成中に障害が発生した場合、そのF木は利用できなくなるおそれがあるため何らかの対策が必要である。最悪の場合は、レコードブロックのキーを読みなおせばF木を再構成することができるが、これには極めて長い時間がかかる。

5 性能評価

ここでは計算機実験の結果について述べる。本研究では、Cによりシミュレーションプログラムを作成し実験を行なった。実験は初期状態で約1000個のキーをランダムに発生させ、これらのキーに対してランダムに更新を行なった。この実験をレコードのアクセスに偏りがある場合と、ない場合 (一様アクセス) とに分けて10回ずつ行なった。実験では、節点の各ポインタにそれぞれ2領域を、リンクに1領域を与え、レコード更新毎にF木も更新すると仮定した。このため、定期的にF木を更新する方式に比べ節点の消去確率はかなり高く実際の場合より利用期間は短く評価されることになる。

5.1 ブロック消去回数

節点ブロックの消去回数の総数を図10、11に示す。どちらの場合も消去回数に大きな違いはなく、更新回数が多くなると1000回のデータ更新毎に節点ブロックが7回消去されることが分かる。また、1000個のキーを保存するのに必要なブロック数は10個程度であり、これらの

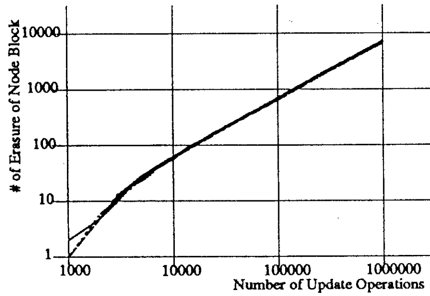


図 10: 節点ブロックの消去回数(偏りあり)

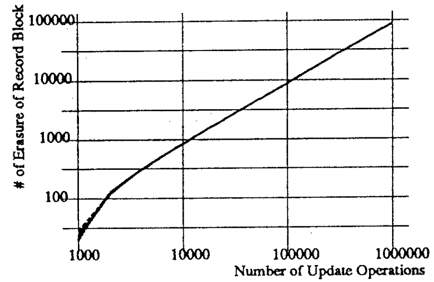


図 12: レコードブロックの消去回数(偏りあり)

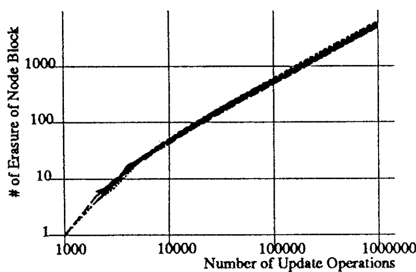


図 11: 節点ブロックの消去回数(一様アクセス)

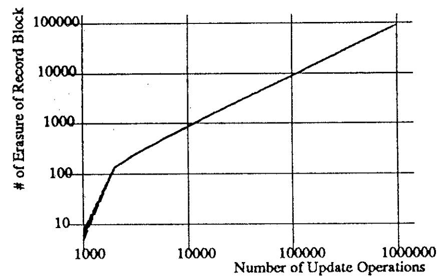


図 13: レコードブロックの消去回数(一様アクセス)

ブロックがほぼ均等に消去されるので1ブロック当たり10000回の更新毎に7回消去されることになる。

次にレコードブロックの消去回数の総数を図12、13に示す。この結果もどちらの場合でも消去回数に大きな違いは無いことを示している。図よりレコードブロックの場合は更新回数が多くなると100回のデータ更新毎にレコードブロックが9回消去されることが分かる。また、1000個のレコードを保存するのに必要なブロック数は100個程度であり、これらのブロックがほぼ均等に消去されるので1ブロック当たり10000回の更新毎に9回消去されることになる。

フラッシュメモリが利用できる期間を消去割合が若干高いレコードブロックについて考察する。フラッシュメモリの消去回数の上限を100万回とすると、$\frac{10^6}{9/10000}$ 回の更新が可能になる。1ブロック当たり1秒間に x 回の更新があるとすると、$\frac{10^6}{365\cdot 24\cdot 60\cdot 60\cdot x\cdot 9/10000}/\simeq 35/x$ 年間ブロックが利用可能である。本稿の方式では各ブロックがほぼ均等に更新されるため利用可能期間が著しくばらつくことは少ない。

5.2 ブロック利用効率

ここでは、利用中のブロックサイズに対するデータやキーの利用効率について述べる。節点ブロックに関してはポインタなどの多重化されている分や利用できないキー領域を考慮すると、最大でブロックの容量の約50%が利用される。レコードブロックに関しては16ページの内1ページが制御情報に使われ、また余裕領域があるためブロックの約75%がレコード保存のために利用されている。また、全体としては約1割がF木の、残9割がレコードの保存に利用されている。

節点ブロックの利用効率を図14、15に示す。最初の内はキーがまとめられているため利用率は64～72%でほぼ一定であるが、約1000回の更新後キーが急激に分散し、ブロック利用率が53～64%にまで下がることが分かる。さらに更新を繰り返すと、分散したキーが再び少数のブロックに集められ利用率は上昇する。更新回数の増加につれて下降と上昇を繰り返し利用率は53～72%の間を変動する。

レコードブロックの利用効率を図16、17に示す。初期状態ではランダムにレコードが生成されるため利用率は

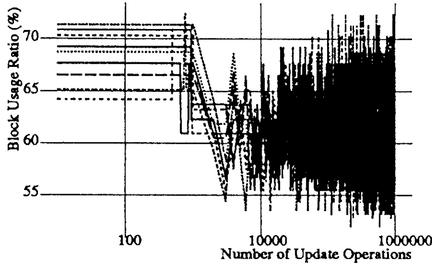


図 14: 節点ブロックの利用効率 (偏りあり)

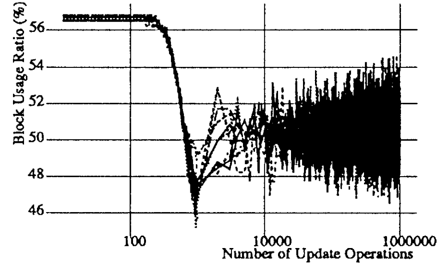


図 16: レコードブロックの利用効率 (偏りあり)

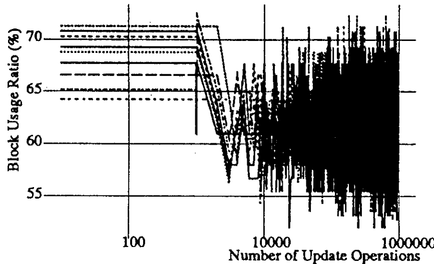


図 15: 節点ブロックの利用効率 (一様アクセス)

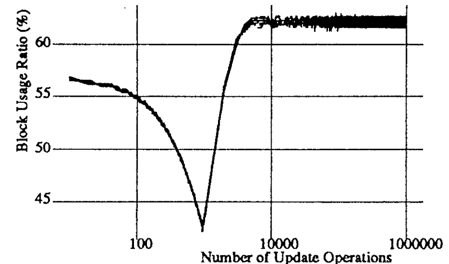


図 17: レコードブロックの利用効率 (一様アクセス)

57%程度である。その後、利用率は約1000回の更新で最低となり、以後上昇する。このように節点ブロックと似た変化をするが、一様アクセスの場合、安定後は61 ~ 62%の間を変動するようになる。これは更新回数が乱数の周期に比べかなり大きいため、更新順序に関連性が現れ、関連の高いものを同一ブロックに保存するようになるためであり、また、更新を受けて分散するブロックは高々数個であり、全ブロック数の100個に比べるとその影響は小さいためである。これに対して偏りがある場合、ほとんどアクセスされないレコードが存在し、これが分散の原因となるため記憶容量は46 ~ 55%の間を変動する。

以上の結果より本稿の方式は約1000更新毎に自動的なガベージコレクションを行なっていると考えられる。

6 まとめ

本稿では、フラッシュメモリを主記憶データベースのバックアップ媒体としても通常の二次記憶としても利用できるようなデータ構造(F木)について、構成の概要と操作アルゴリズムについて述べた。また、F木性能評価を計算機実験によって行ない十分利用できることを示した。尚、本研究は文部省科学研究費補助金(特別研究員奨励費)によるものである。

参考文献

[1] G.Copeland, T.Keller, R.Krishnamurthy, M.Smith, "The Case For Safe RAM," Proc. of the 15th International Conf. on VLDB, 1989, pp.327-335.

[2] T.J. Lehman, M.J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," Proc. of the 12th Int. Conf. on VLDB, 1986, pp.294-303.

[3] H. Takakura, Y. Kambahashi, "Continuous Backup Systems Utilizing Flash Memory Proc. of 9th Int. Conf. on Data Engineering," 1993, pp.439-446.

[4] 高倉弘喜, 上林彌彦 "フラッシュメモリバックアップ方式の設計と性能評価", 電子情報通信学会論文誌, D-I, Vol.J76-D-I, No.10, 1993, pp.514-521.

[5] 高倉弘喜, 上林彌彦, "主記憶データベース向きのデータ構造の検討", 情報処理学会第47回全国大会, 1993.

[6] 高倉弘喜, 上林彌彦, "フラッシュメモリ向きデータ構造F木の性能評価", 情報処理学会第48回全国大会, 1994.