数理解析研究所講究録
第525巻 1984年 100-123

大容量主記憶のもとでのファイルの構成法

京都大学工学部　小島　功　(Isao KOJIMA)
京都大学工学部　上林　弥彦　(Yahiko KAMBAYASHI)

1．まえがき

データベースシステムなどの大量のデータを操作，格納するような応用には，効率のよいファイル構成が必要である。

ファイル構成はディスク等の二次記憶を用いて検索／挿入／削除といった基本操作を実現するものであり，これまでにB－tree，ISAM，ハッシュなど種々のファイル構成が提案されている。ファイル構成の評価の基準としては，ディスクアクセス回数や記憶効率などが考えられている。

ところで，近年のハードウェアの発達により，大容量の主記憶が容易に実現可能になってきつつあり，これを用いてファイル構成の性能を向上することが考えられる。また，主記憶上にデータベースを実現することも可能となると思われる。主記憶の利用に関しては，索引木のコピーやディレクトリの常駐などの方法が知られているが，このような方法は主記憶を効率的に利用しているとは言えない。さらに，主記憶とディスクとはその性質が大きく異なるので，従来のフ

ァイル構成をそのまま用いるのは有利とは言えない。
そこで本論文では大容量の主記憶に適した新しいファイル構成を提案する。これは，動的ハッシュに基づくものであり，従来のファイル構成に比べて次のような特徴を持つ。
（１）ディレクトリを小さく実現してより多くのレコードを主記憶上に格納する。
（２）ディレクトリが自然に再構成されるので，レコード量が増加してもディレクトリ領域や，主記憶アクセス回数が増えることが少ない。
（３）種々のオーバーフロー処理を支援することで高い記憶効率を実現している。

本論文では，２章において主記憶をファイルに用いる時の問題点を示し，３章でファイルの構成とその操作方法を示す。４章ではこの応用として，部分マッチ検索に適用した場合を考える。

２．大容量の主記憶を用いたファイルの実現

従来提案されているファイル構成のうち，特に重要なものは，レコードの挿入／削除に伴い，ファイルが自動的に再構成される，動的ファイルである。　動的ファイル構成には次のようなものがある。

（１）Ｂ－ｔｒｅｅ

Ｂ－ｔｒｅｅはディスク上に木構造の索引を構成するもの

であり，次のような特徴を持っている。

（ａ）ディスクページの利用効率が最低でも５０％以上常に保たれる。

（ｂ）索引木を構成すると，その木は常にバランス木となる。すなわち木の端節にレコードを置く場合，各レコードまでのアクセス回数が等しくなる。

（ｃ）ディスクページを利用して，多分木の索引構成を実現している。これは，木の高さを低くし，レコードまでのディスクアクセス回数を減少させる。Ｂ－ｔｒｅｅにおける，レコード挿入とそれに伴うページ分割を図２．１に示す。

（２）動的ハッシュ

ハッシュに基づくファイル構成は，レコード集合の量にかかわらず高速のアクセスを支援するが，同じハッシュ値をもつレコードが増えると格納できなくなる。動的ハッシュは，このような状態をハッシュ関数の拡張によって解決している。

動的ハッシュはディレクトリを主記憶内に構成して，それによりハッシュ関数の変化を扱うから，ディレクトリの種類によって，次のような種類がある。

（ａ）主記憶にテーブル状のディレクトリを構成するもの。

（Ｅｘｔｅｎｄｉｂｌｅ　Ｈａｓｈ）

（ｂ）主記憶に木構造のディレクトリを構成するもの

（Ｔｒｉｅ　Ｈａｓｈ，Ｄｙｎａｍｉｃ　Ｈａｓｈ）

（ｃ）オーバーフローチェインを用い，主記憶にディレクト

リを置かないもの。

（Ｌｉｎｅａｒ　Ｈａｓｈ，Ｖｉｒｔｕａｌ　Ｈａｓｈ）

これらの（ａ）～（ｃ）について動的ハッシュの例を図２．２に示す。

以上のようなファイル構成における主記憶の利用法としては，次のような方法が知られている。

（１）索引木の一部を主記憶にコピーする。

（２）動的ハッシュのディレクトリを主記憶に常駐させる。

（３）ディレクトリ以外の主記憶領域はキャッシュとして用いられる。

次に，主記憶向きのファイル構成の基本となる，主記憶の性質特にディスク装置との性質の差をまとめる。

（ａ）アクセスが高速：ディスクアクセスは５０～５００ｍｓのアクセス時間を要するが，主記憶では１００ｎｓ程度のアクセスが行える。

（ｂ）アクセスの単位が小さい：ディスクと異なり，ページ単位のアクセスを必要としないので，任意長のレコードが実現できる。

（ｃ）揮発性：電源の切断によって主記憶の内容が失われる。

（ｄ）ディスク装置と異なり，主記憶は電源を投入すると内容のすべてがアクセス可能な状態に置かれるので，エラーなどの影響が大きい。

（ｅ）ディスクヘッドのような可動部分がないので，次に読

むレコードのアクセスコストが現在読んでいるレコードによって変化することがない。すなわち順アクセスと直接アクセスのアクセスコストが変わらない。

（ｆ）ディスク装置に比べると高価。

以上のような性質から，ファイル構成に主記憶を利用する場合，次のような問題を扱う必要がある．

（１）２分木実現：主記憶では，多分木におけるノード内の探索はキーの順番に行われるので，２分探索の方が有利となる。（図２．３参照）

（２）不平衡木実現：ディスクアクセスコストと主記憶アクセスコストとの差が大きいので，（ａ）から主記憶内に構成されたディレクトリは不平衡木実現であっても問題がない。（図２．４参照）

（３）初期化：主記憶は電源投入時には空であるから，効率のよい初期化操作を実現する必要がある。

（４）バックアップ：（ｃ），（ｄ）より，主記憶上のデータを二次記憶上に記録する，バックアップの手続きを考える必要がある。

（５）ディスク上のデータ構造は（１），（２）より不平衡２分木実現が適しているが，（４）で用いるバックアップは二次記憶上で実現されるので，平衡多分木の方が有利となる。

従って，異なるデータ構造の間のバックアップをとる必要がある。

（６）（ａ）（ｆ）より，処理を複雑にしても高い記憶効率を実現する必要がある。

本論文ではこれらのうち（１）（２）（４）（６）の問題を主に扱っている。　次章に示すファイル構成は動的ハッシュに基づくものであるが，ハッシュに基づくファイル構成を考える理由は次のようなものである。

（１）ハッシュ関数に基づくアクセスは，２分探索と同様なアクセス方法を支援することができる。

（２）キーの比較を行う索引木は，主記憶内のアクセスが増加すると共に，キーを格納するために多くの主記憶領域を必要とする。

３．ファイルの構成

３．１　ファイル構成のための基本方針

ここでは，データベースのかなりの部分を格納できる大きさの主記憶を用いる場合を想定する。　ここで示すファイル構成は，２章で示した動的ハッシュに基ずくものであるが，主記憶を利用する環境から，次のような性質を持つものが重要と考えられる。

（１）主記憶上には，ディレクトリとレコードが置かれ，オーバーフロー時にはレコードがディスクに書き出されるが，ディレクトリ領域をできるだけ小さく実現することで，より多くのレコードを主記憶上に置き，ディスクアクセス回数の減少を図る。

（２）ディスクアクセス回数をできるだけ減らすために，オーバーフローチェインは用いない。　そのため，Ｌｉｎｅａｒ　Ｈａｓｈのような方法はディレクトリ領域が不要であるという特徴を持つが使用できない。

（３）主記憶領域が大きいので，大きなディレクトリが実現できる。　これは，主記憶内のアクセス回数を増加させる。

また，効率の悪いハッシュ関数を用いた場合も図３．１のようにディレクトリが不必要に大きくなる場合があり，データ量が増加しても再構成によりディレクトリを小さくすることが望ましい。

（４）主記憶上でレコードとディレクトリが共存しており，データの増加に伴いレコードはディスクに書き出されるが，その変化がゆるやかである。　Ｅｘｔｅｎｄｉｂｌｅ　Ｈａｓｈのようにテーブル状のディレクトリを持つものは，図３．２のように一時的に大量のディスクアクセスを必要とする。例えばハッシュ長が１６－＞１７ｂｉｔになるときは６４ｋバイト，２０－＞２１のときは１Ｍバイト（エントリを１バイトとする）を一度に書く必要がある。

（５）更新に伴う影響が少ない。（４）も含めて，一回の更新に伴う操作は少ないほうが望ましい。例えば，Ｔｒｉｅ　Ｈａｓｈはディレクトリが線形に拡大するが，この方法は図３．３のようにオーバーフローを起こしたレコードとは無関係にディレクトリに隣合ったレコードをディスクに書く必要がある。

（６）このような環境では，ファイルのバックアップにかなりの時間がかかると考えられる。　その間ファイルをロックするとシステムの可用性が低下する。　従ってファイルの処理中にバックアップが行える構成を考える必要があり，図３．４のようにバックアップを取り始めた時点の状態を保てる構造が必要となる。

以上のような点を考慮して，ファイル構成を考える。

３．２　ファイルの構成

先に示したように，動的ハッシュに基ずくファイル構成を考える場合，次の２点をとくに考慮する必要がある。

（１）主記憶上のディレクトリ構造

（２）ディスクにおけるオーバーフローの処理

本論文で示す構成は，次の点が従来のハッシュファイルと異なっている。

（１）一般に用いられるディレクトリ構造（木，テーブル等）は主記憶の占有領域と主記憶内アクセス回数の点で主記憶に用いるのは適当でない。

従ってここでは容量の小さいディレクトリ構造を考える。

（２）ディスク領域の管理はページ分割やオーバーフローページの共有といった方法が提案されている。　本論文でもこれらの方法に従うが，記憶効率の向上のためにディスクアクセス回数を増やす方法は避け，種々の状態に応じていくつかの処理方法を支援することで記憶効率の向上を図っている。

（Ａ）　データ構造（図３．５参照）

まずディレクトリの構成を示す。基本的にはメモリ空間上に木構造を実現するが，そのまま実現するとポインタ等の領域が増加するのでフラグによる制御をおこなっている。　つまり，オーバーフローの時に作るディレクトリの場所を規則的に決めることでポインタを省いている。

各エントリに対して次のような構造を設ける。

（ａ）Ｐｏｎｉｔｅｒ　Ｆｉｅｌｄ

ハッシュ値に対応するレコード領域を指すのに用いられるが，これには次のようなデータが含まれる場合がある。

（１）ディスク領域を指すポインタ。

（２）主記憶領域を指すポインタ。

（３）ハッシュ値による分離キーの格納。

（ｂ）Ｒｅｈａｓｈｉｎｇ　Ｆｌａｇ

ここで提案するファイルは主記憶上で再ハッシュを行う必要がある。このフラグがセットされているときは再ハッシュを行う。それ以外のときは，上の（１）～（３）　を区別するのに用いられる。

次にディレクトリ全体にたいして下のデータ構造を与える。

（ｃ）Ｄｅｐｔｈ　ｏｆ　Ｈａｓｈ

現在用いられているハッシュ関数の有効レベル，　つまり初期ハッシュの長さである。

（ｄ）Ｄｅｐｔｈｃｏｕｎｔ（Ｎ）

ディレクトリの各エントリには種々のものがあるので，それぞれのレベルにおいてその個数を記憶する。　これは初期ハッシュの変更に用いられる。

ディスク上のページには主にレコードのみが置かれるが記憶効率の向上のために次のようなデータが記憶される。

（ｅ）Ｄｉｓｋｓｔａｔｅ

ディスク領域が複数のエントリによって共有されているかど

うかを示す。

（ｆ）Ｓｐｌｉｔ　Ｋｅｙ

（ｅ）で共有している場合，このページがオーバーフローするとページを分割する。このときに，ハッシュ値を用いる。

（Ｂ）　ファイルの操作

次に，ファイルに対する検索，挿入といった基本操作の実現方法を示す。

（１）初期化

ある一定数のレコード集合を用いて初期化を行う。　これは，レコードの分布を調べて，適当なハッシュ関数と，その有効長を決定するものである。　これにより定まったハッシュ関数とその深さに対して，

（ａ）主記憶領域を分割し，各領域の先頭にエントリを置く。

（ｂ）初期化に用いたレコードをハッシュする。

（ｃ）もし，すべてのレコードが主記憶領域に格納できない場合は，（３）の挿入により処理する。　初期化の様子を図３．６に示す。

（２）検索

検索のアルゴリズムの基本的な考えは次のようなものである。

（ａ）定められた長さの初期ハッシュを行う。

（ｂ）ディレクトリの再ハッシュ用のフラグがセットされていれば，再ハッシュを行い，ハッシュ長を１ｂｉｔ拡張する。

これをレコード領域が見つかるまで続ける。

（ｃ）そうでなければ，ｐｏｉｎｔｅｒによりレコード領域をアクセスする。

ここで示すディレクトリ構成は，木構造のうち内部ノードにあたる部分にはポインタを用いないので，各エントリは，ある規則に従って配置されている。　検索の例を図３．７に示す。この場合は一回の再ハッシュが行われる。

（３）挿入

挿入において重要な点は，オーバーフローの処理方法であるが，ここではディレクトリの構成から，次のようなオーバーフロー処理法を支援している。（図３．８（ａ）～（ｃ）参照）

（ａ）オーバーフローレコードが少ない場合は，ハッシュ関数の拡張を行う必要はない。　そのため，複数のエントリで一つのオーバーフローページを共有する。　このページがあふれた場合，ページ分割を行う。　これは，ページ内のハッシュ値を分離値として用いる。　この操作が一エントリに対して一個のオーバーフローページが割当てられるまで行われる，。

（ｂ）　（ａ）において一エントリに割当られたオーバーフローページがあふれた場合，ハッシュ関数の拡張を行う。これは，ディレクトリの後ろに続く主記憶領域を半分に割り，

それぞれの先頭にエントリを付ける。　ディスクページは２つに分け，それぞれのエントリから指されるようにする。古いエントリは再ハッシュのフラグをセットし，ｐｏｉｎｔｅｒ領域を解放してレコードやディレクトリ領域に用いる。

（ｃ）（ａ，ｂ）において，レコードが均等に分散されない場合は，ハッシュ値による分割を行う。　これは，（ｂ）において解放したｐｏｉｎｔｅｒ　ｆｉｅｌｄに分離キーとしてのハッシュ値を格納し，その値との比較によってレコードを分割するものである。

以上のようなオーバーフロー処理の特徴は，ファイルの拡大に伴うディレクトリの拡大が（ｂ）の操作によって再構成されることである。　さらに，ハッシュ関数が均等に拡大されると，初期ハッシュの長さを増やせるので，フラグ領域も解放できると共に再ハッシュの回数も減少する。　ディレクトリの再構成の様子を図３．９に示す。

（４）削除

削除のアルゴリズムは基本的には挿入の逆と考えることができる。　但し，次の２点が異なっている。

（ａ）レコード削除に伴い，主記憶領域に空白が生じる。

（ｂ）２つのエントリが併合され，新たなエントリを設ける時に，他の主記憶領域に影響を与えることがある。

（５）順アクセス

順アクセスはハッシュ値のインクリメントにより行うから，

順序保存ハッシュ関数を用いる必要がある。　ここで示した方法は検索を１回のディスクアクセスで処理できるが，ディスク領域の物理的な近接性は扱っていない。

３．３　ファイルの評価

以上のような操作によって実現されるファイルは次のような特徴をもつ。

（１）ディレクトリ領域が小さい。

木構造を実現するのにポインタを用いないので，ディレクトリ領域が小さい。

（２）ディレクトリが木構造とテーブル構造との中間的な構成であるから，ハッシュ関数の状態によって，レコード集合に対するディレクトリ領域の割合が大きくならない。　例えば，テーブル構造のディレクトリの場合，ハッシュ関数によるレコードの分布に偏りがある場合は不利となる。　一方，木構造のディレクトリの場合には，レコードの分布が均等でも木の内部ノードの領域が大きくなる。ここで示すファイル構成は以上のようなハッシュファイルを特別な場合として含むものと考えられる。　（図３．１０参照）

（３）ディスクアクセス回数が少ない。オーバーフローチェインを用いないので，検索操作は１回のディスクアクセスでよい。　その他の場合は次表に示すが，Ｂ－ｔｒｅｅに比べて更新の伝搬がないので，ディスクアクセス回数は少ない。

（４）レコード集合の拡大に伴うディレクトリ領域の拡大は１回のオーバーフローに対して１エントリであるからデータの移動が少なく，主記憶領域はゆるやかにディレクトリによって置きかわっていく。　（図３．１１参照）

（５）また，ディレクトリ領域とレコード領域がある規則に従って配置されているので，更新に伴う影響は局所化されており，不要なディスクアクセスを行わない。　（４），（５）に関しては，図３．１２のように，Ｅｘｔｅｎｉｄｉｂｌｅ　ＨａｓｈやＴｒｉｅ　Ｈａｓｈと比べて，レコード集合の拡大に伴うディスクアクセスの合計は同じであるが，それらが一時的に大量に行われない点に特徴がある。

（６）ディレクトリの構成がハッシュ関数の拡張に伴って再構成される。これは，ハッシュ関数が均等に拡大すれば，古いエントリは消去されるので，次のような利点を持つ。

（ａ）初期ハッシュの長さが変わるので，主記憶内のアクセス回数が減る。

（ｂ）エントリ領域が解放されるので，主記憶領域を有効に利用できる。　１個のエントリは小さく，レコードの格納にはあまり有効でないが，レコード領域をシフトすることで新しいエントリの生成時に用いることができる。

（７）記憶域の利用効率が高い。　ディスクの利用効率はページ分割に基づくので，Ｂ－ｔｒｅｅと同じ最低でも５０％以上を保つ。　挿入操作の（ａ），（ｃ）の場合は分離キー

を用いれるのでＢ＋ｔｒｅｅと同じとなる。（最低６７％）

主記憶の利用効率は（１），（２）で示したディレクトリの領域の他は１００％近く利用されている。　但し，主記憶上のレコードに削除を行った場合，空領域を詰めるには余分のディスクアクセスが必要となるが，そのあとに挿入があると再びオーバーフローを生じるので，利用方法によりどちらかを選択しなければならない。

（８）種々のオーバーフローの方法の支援によって，さまざまな形のディレクトリを実現できる。　これにより，ある時点でのディレクトリの形を保ちながらファイルの操作を行える。　これは，ファイル処理中のバックアップを可能にしている。

領域利用効率の向上

このファイル構成に対し，ディスクの利用効率を改良する方法を示す。

このファイルはＢ－ｔｒｅｅと同様の記憶効率を保つが，

（ａ）Ｌｉｎｅａｒ　Ｈａｓｈのようにオーバーフローチェインを用いるもの。

（ｂ）Ｄｅｎｓｅ－Ｔｒｅｅのようにレコードのノード間移動を行うもの。

はいずれも１００％近いディスク記憶効率を実現している。

これに対しては，フリーリスト（図３．１３参照）を用い

る方法を示す。これは，各ハッシュエントリに対して，ディスクの空き領域の大きさを示すフィールドを新たに設け，ポインタでつないだものである。　これを用いると空き領域の大きさがすべて管理されているので１００％近い記憶効率を実現できる。　この方法の得失を次に示す。

（１）３０％近くページ数が減少するので順アクセスが高速に行える。

（２）フリーリストの分だけレコードがディスクへ追い出される。

（３）更新のたびにフリーリストの管理が必要になる。また，ほとんどの挿入はオーバーフローを引き起すことになる。

４．あとがき

本論文で示したファイル構成は動的ハッシュに基ずくが，ディレクトリの構造をできるだけ小さくしてレコードを数多く主記憶に置きファイルの性能を向上させている点と，ディレクトリが再構成されてレコードの増加に伴うディレクトリ領域の増加や主記憶内アクセス回数の増加を抑えている点に特徴がある。　ファイル操作に関しては，順アクセスの処理について問題が残っている。　これは，一般にはハッシュ関数は順序保存でないが，主記憶領域を用いることで，そのような検索が行れる属性の値を主記憶に置いて処理できるからである。　また，本論文では扱わなかったファイルの初期化

およびバックアップについても考察する必要がある。

謝辞　　日頃熱心に御討論頂く矢島脩三教授をはじめとする研究室の皆様に深謝いたします。

References

[BURK 83] Burkhard,W.A. "Interpolation - Based Index Maintenance", 3rd PODS, pp76-85,1983.

[COME 7906] Comer,D. "The Ubiquitous B-trees", ACM Comput. Surv. Vol.11, NO.3, pp 121-137, June. 1979.

[FAGIN7909] Fagin,R. Nievergelt,J. Pippenger,N. and Strong H.R. "Extendible Hashing - A Fast Access Method for Dynamic Hashing", ACM TODS, Vol.4, No.3, pp315-344, September, 1979

[LARS 78] Larson,P.A. "Dynamic Hashing" BIT, Vol.18, pp184-201, 1978.

[LARS 80] Larson,P.A. "Linear Hashing with partial Expantions", 6th VLDB, pp224-228, 1980

[LARS 8112] Larson,P.A. "Analysis of Index-Sequential Files With Overflow Chaining", ACM TODS. Vol.6, No.,4 pp761-681, December, 1981.

[LARS 8212] Larson,P.A. "Performance Analysis of Linear Hshing with Patial Expantions", ACM TODS, Vol. 7, No.4, pp566-587, December. 1982.

[LITW 7809] Litwin,W. "Virtual Hashing: A Dynamically Changing Hashing", 4th VLDB, pp517-523, September, 1978.

[LITW 80] Litwin,W. "Linear Hashing : A New Tool For File and Table Addressing", 6th VLDB, pp212-223, 1980

[LITW 81] Litwin,W. "Trie Hashing", SIGMOD, pp19-29, 1981.

[LOME 8109] Lomet,D.B. "Digital - B-trees", 7th VLDB. pp333-344, September, 1981

[LOME 8303] Lomet,D.B. "Bounded Index Exponential Hashing", ACM TODS, Vol.8, No.1, pp136-165, March. 1983

[LOME 83] Lomet,D.B. "A High Performance Universal Key Associative Access Method". SIGMOD 83. pp120-133.

[MULL 8105] Mullin,J.K. "Tighly Contorolled Linear Hashing Without Separate Overflow Storage", BIT, Vol.21, pp390-400, May, 1981.

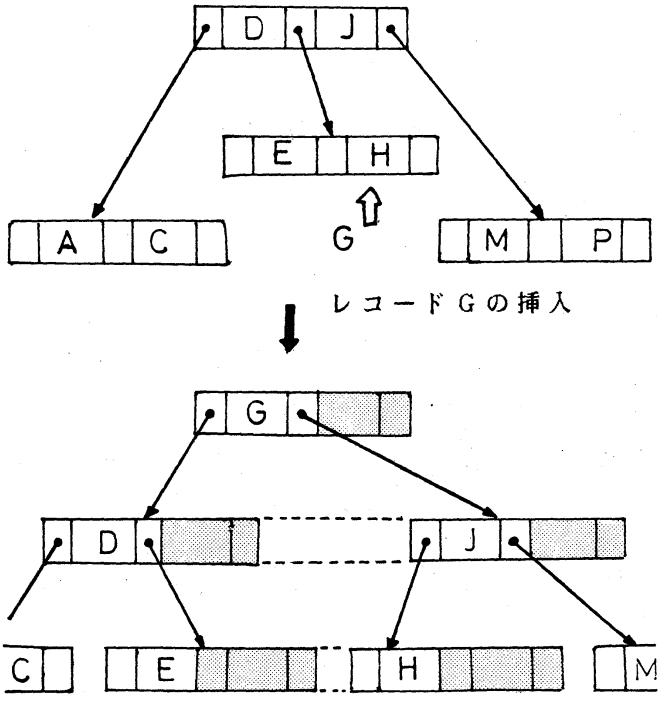
[OLLE 83] Orenstein,J.A. "A Dynamic Hash File for Random and Sequential Accessing", 9th VLDB ,pp132-141, 1983.

[SCHO 8103] Scholl,M. "New File Organizations Based on Dynamic Hashig", ACM TODS, Vol.6, No.1 pp194-211, March,1981

[STRO 7709] Strong,H.R. "Search Within a Page" IBM Research Rep. RJ2080, September, 1979

[TAMM 8112] Tamminen,M. "Order Preserving Extendible Hashing and Bucket Tries", BIT, Vol.21, pp 419-455. May, 1981.

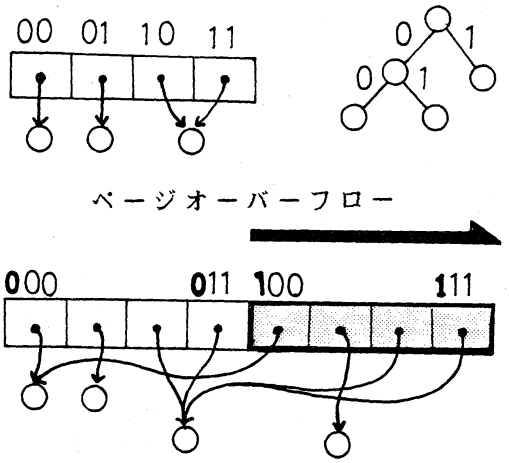
[TOREB83] Torenvliet,L. and Boas,v.E. "The Reconstruction and Optimization of Trie Hashing Functions", 9th VLDB, pp142-155, 1983.



（図２．１　Ｂ－ｔｒｅｅにおけるページ分割）

表形式のディレクトリを支援するもの。

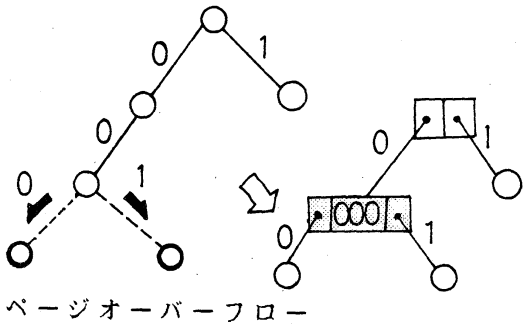
(1)（Ｅｘｔｅｎｄｉｂｌｅ　Ｈａｓｈ）



(2)

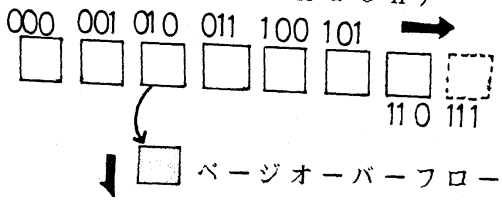
木構造のディレクトリを支援するもの。

（Ｔｒｉｅ　Ｈａｓｈ）

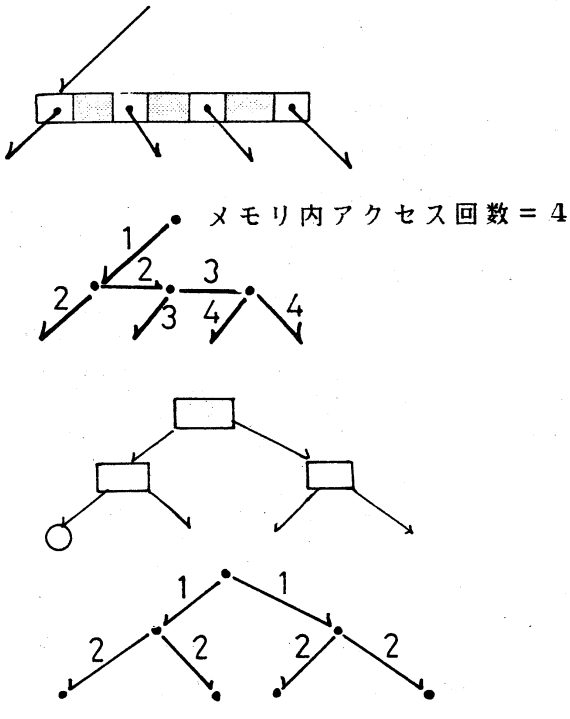


(3) ディレクトリ構造を持たないもの

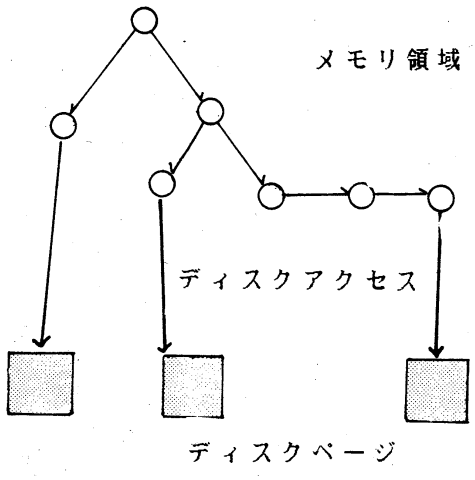
（Ｌｉｎｅａｒ　Ｈａｓｈ）



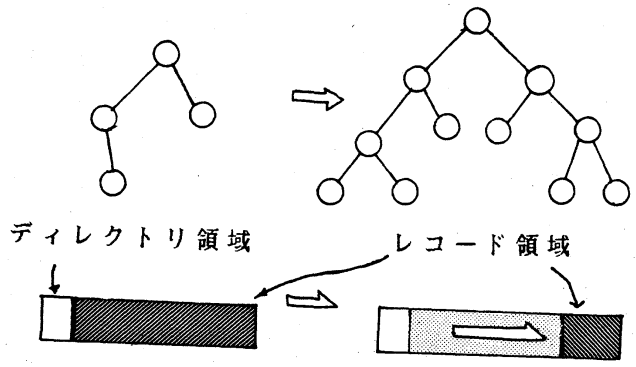
（図２．２　動的ハッシュの種類）



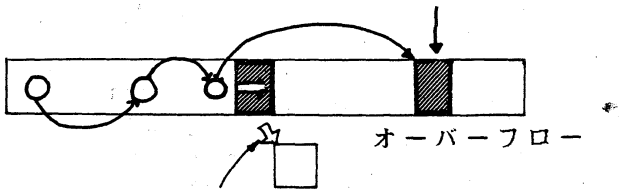
（図２．３　２分木実現と多分木実現）



（図２．４　不平衡木の実現）

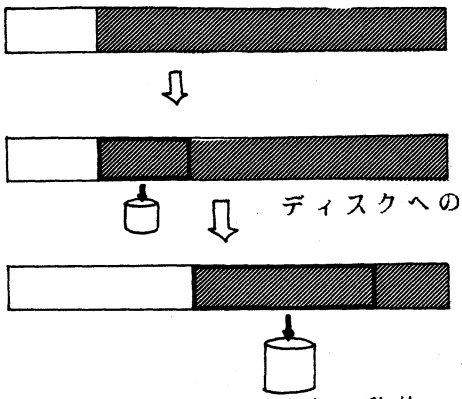


（図３．１　ディレクトリの拡大によるレコード領域の縮小）

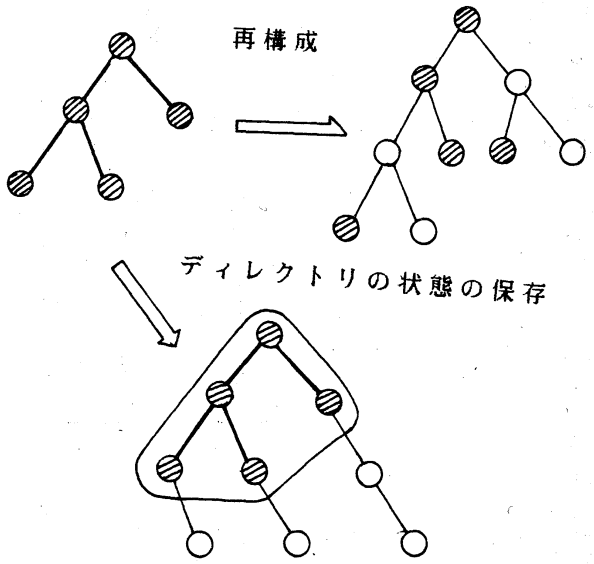


ディレクトリの拡大によりディスクに書かれるレコード

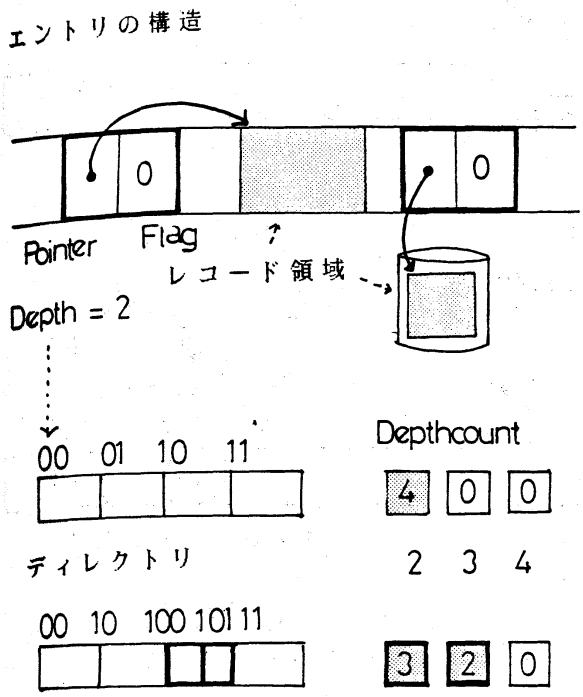
（図３．３　木構造の動的ハッシュにおけるディレクトリの拡大）



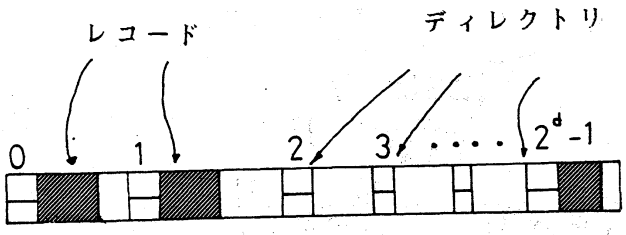
（図３．２　テーブル形式の動的ハッシュにおけるディレクトリの拡大）



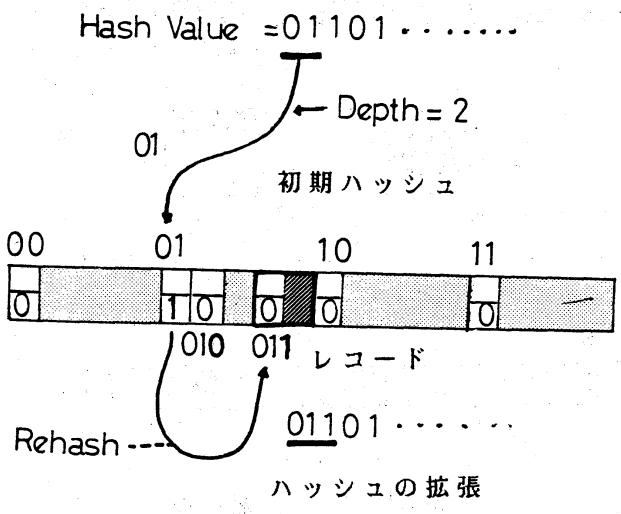
（図３．４　ファイル処理中のバックアップ）



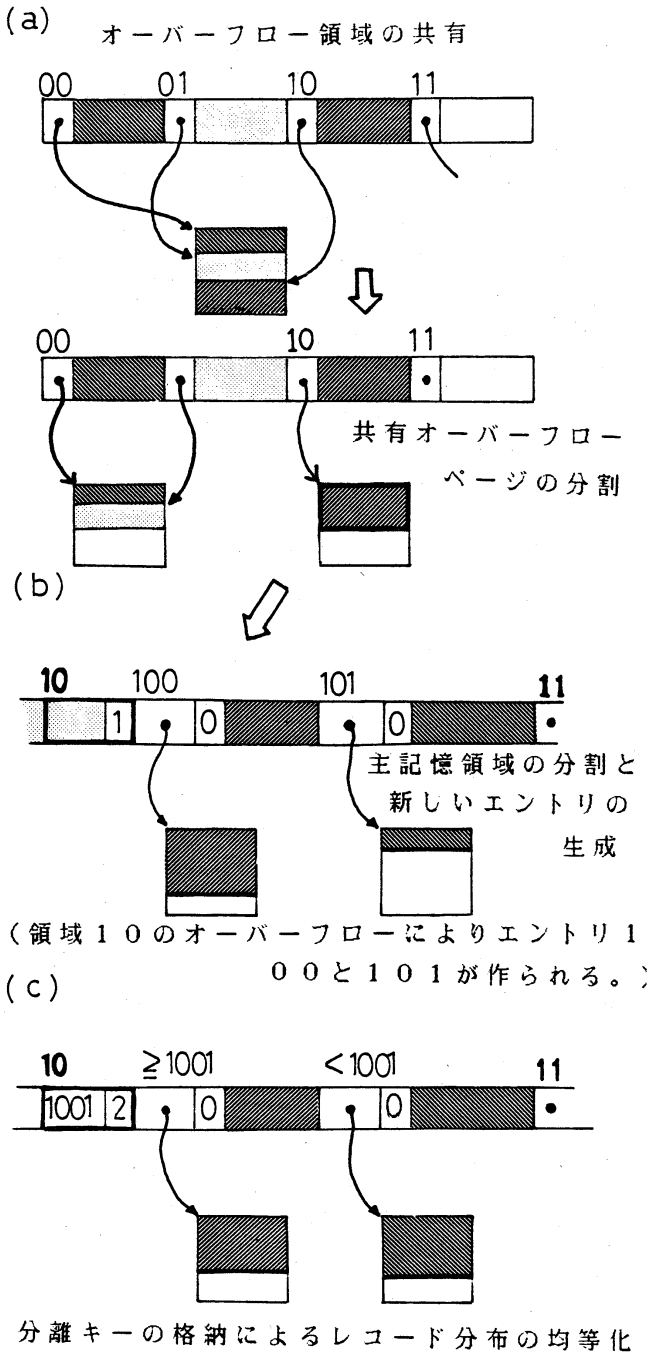
（図３．５　ファイルのデータ構造）



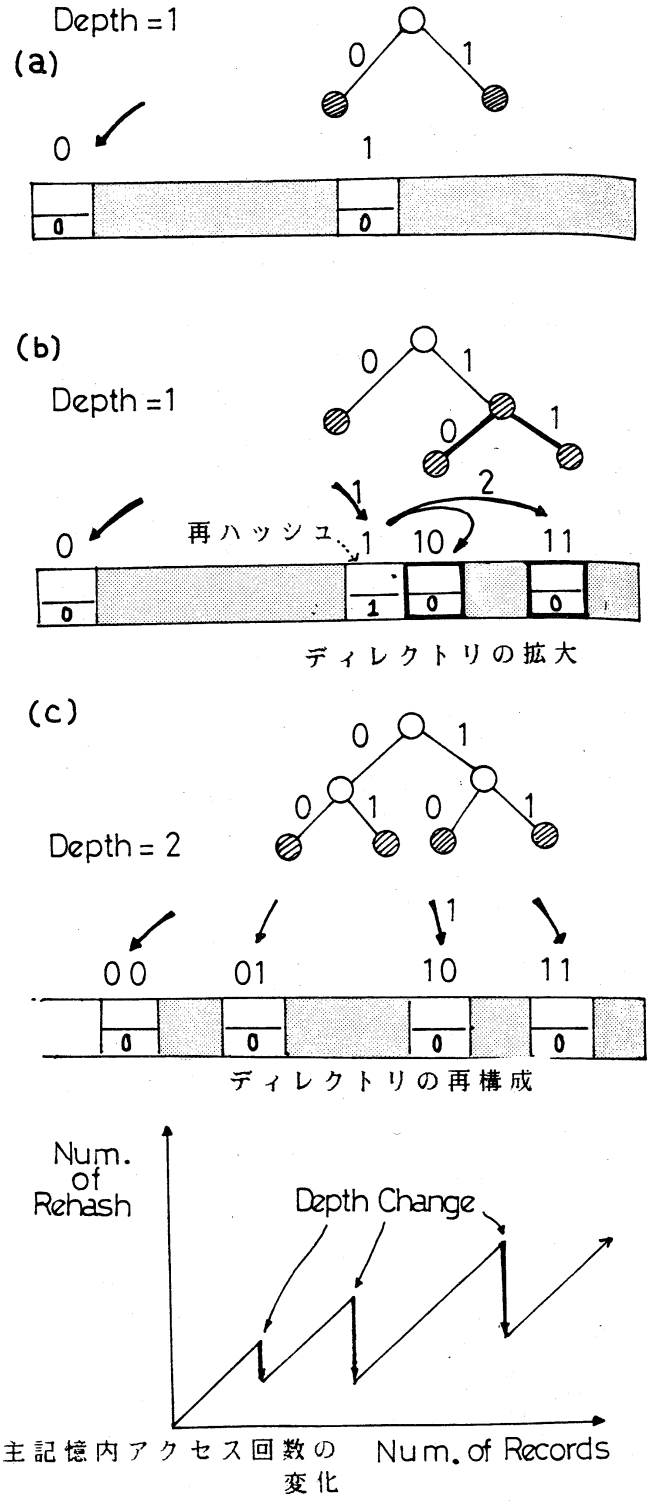
（図３．６　初期化）



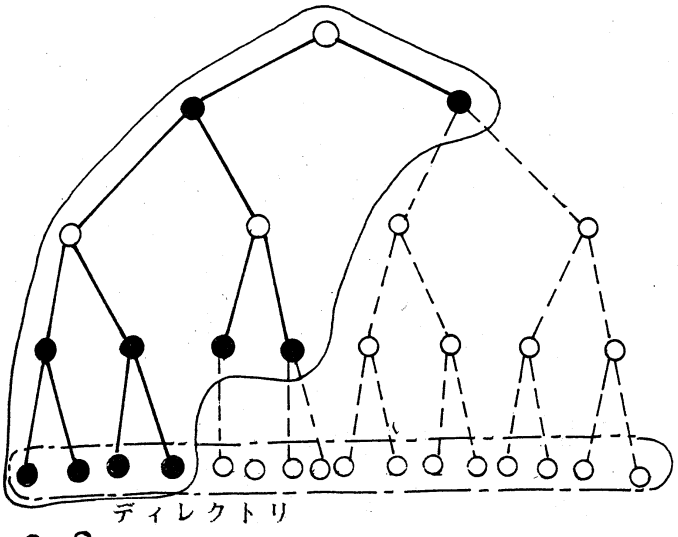
（図３．７　検索）



（図３．８　挿入）



（図３．９　ディレクトリの再構成）

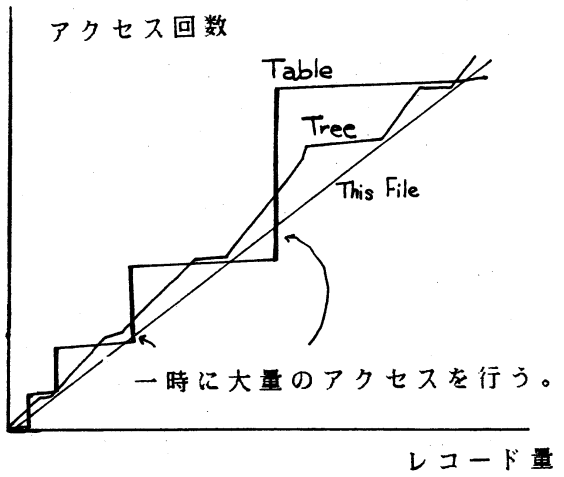


本論文で示すファイルにおいて作られるエントリ

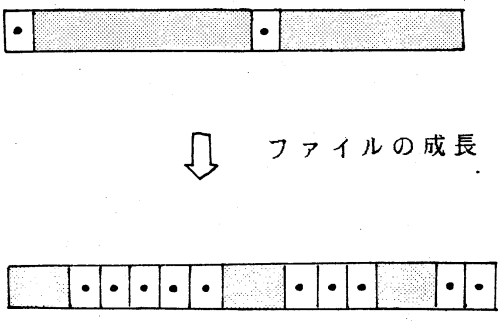
木構成のディレクトリにおいて必要なエントリ

テーブル構造のディレクトリにおいて必要なエントリ

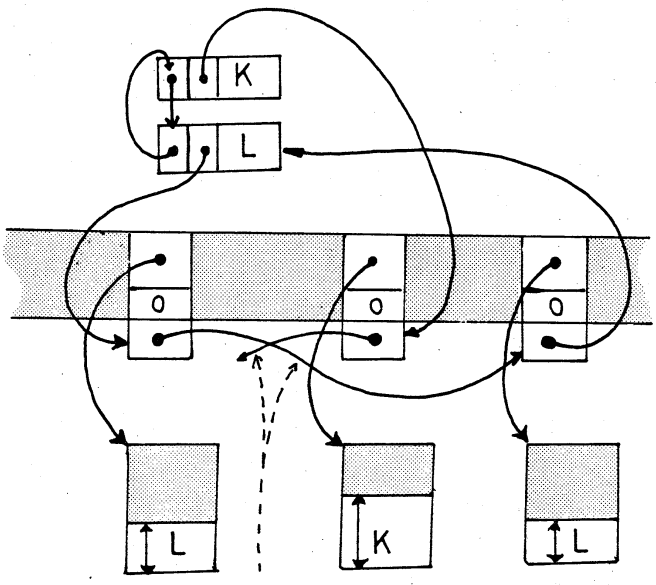
（図３．１０　ディレクトリ構成の特徴）



（図３．１２　ファイルの成長とディスクアクセス回数）



（図３．１１　ディレクトリ領域の拡大）



同じ領域の大きさのページを結んだチェイン

（図３．１３　フリーリストによる空領域の管理）