データベースシステム　88－1
（１９９2.　5.　8）

ベクトル計算機上でのデータベース演算の実現

目木信太郎　　上林弥彦
京都大学工学部

データベースの応用分野の拡大に伴って、高速なデータベースシステムに対する要求はますます大きくなりつつある。特別なハードウェを用いてデータベース演算を効率よく実行するために多くの努力がなされてきたが、市販されているLSIを使うことができないので、これらの試みは現実的ではないことが多く、これに代る方法が求められている。ベクトル計算機は強力なパイプライン演算器と大容量の主記憶を備えており、近年急速に普及が進んでいる。そこで本稿では、ベクトル計算機の特徴を十分に生かしたベクトル計算機向きのデータベース演算のアルゴリズムを示す。アルゴリズムをベクトル計算機上に実現した結果、10倍以上の加速率が得られることが明かになった。

Realization of Relational Database Operations on Vector Processors

Shintaro Meki and Yahiko Kambayashi
Faculty of Engineering, Kyoto University
Sakyo, Kyoto, 606-01 Japan

Due to the expansion of database application area, requirement for fast database operations has been getting larger. Although great efforts have been made to process database operations efficiently by means of special hardware, these attempts seem not to be economically feasible, and another approach is required. Vector processors have been getting popular these days, which have powerful pipeline function units and large amounts of main memory. In this paper vectorized algorithms for relational database operations are described, which sufficiently use the features of vector processord. These algorithms are implemented on a vector processor, and it is shown that acceleration ratios are over ten.

1．はじめに

近年のデータベースの応用分野の拡大のために、高速なデータベース管理システム(DBMS)に対する要求は年々大きくなっている。DBMSのための特別なLSIを開発するには非常に時間がかかるので、そのような試みはかなり高価なものになり、しかも最新のハードウェア技術を利用することができない。コストに見合ったDBMSを開発するための比較的うまくいっているアプローチとして二つの方法がある。主記憶データベースを実現するために大容量の主記憶を備えた計算機をもちいることと、並列データベースシステムを実現するために多くのマイクロプロセッサを使うことである。

近年の大規模集積回路技術の進歩に伴ってメモリデバイスの価格は下がってきており、計算機の主記憶容量はますます大きくなりつつある。したがって、データのすべてあるいは主要な部分を主記憶上に置くことは可能になりつつあり、そのことによって二次記憶装置に対するアクセスが減少して高速なDBMSが可能になる。また、ハードウェア技術の進歩によってマイクロプロセッサの能力が上がるとともにその価格が下がってきており、並列データベースマシンを経済的に実現することも近年可能になってきている。これらのアプローチは大量生産されたLSIチップを利用しているので、高速なデータベース演算のための特別なLSIを開発する必要はない。したがって、これらは高速なDBMSを実現するためのかなり経済的な方法であると考えられる。

本稿ではベクトル計算機つまりスーパコンピュータをもちいるという、別のアプローチをとることにする。ベクトル計算機は近年の大規模数値計算の要求に応えるべく開発され、さまざまな分野の数値計算にもちいられている。この計算機はアレイ構造のベクトルデータに対する演算をパイプライン演算器で高速に実行し、大規模数値計算において大量のデータを扱う必要からギガバイトオーダーの大容量の主記憶も備えている。したがって、主記憶データベースをベクトル計算機上に実現することは十分可能であり、その高い性能を十分に引き出してやれば、それは高速なデータベースシステムとなることが期待できる。実際、ベクトル計算機ではベクトルレジスタ上のデータに対する演算だけでなく、主記憶とベクトルレジスタの間のデータの転送もパイプライン演算器によってベクトル処理されるので、関係データベース演算のような、本来の数値計算とは異なる演算も高速に実行することが期待できる。以上のことを踏まえて、我々はベクトル計算機上に関係データベース演算を実現する方法を考案し、それらを実際に京都大学大型計算機センターのベクトル計算機上に実現した。パイプライン演算器はすでにチップ化されており、将来的にはより多くの計算機でもちいられると考えられるので、これらの方法はますます重要なものになっていくと考えられる。

本稿は以下のように構成されている。次章では、基本概念として関係データベース演算とベクトル計算機にふれておく。3章では、ベクトル計算機上での実行に適した関係データベース演算のアルゴリズムについて述べ、4章では、3章で導入したアルゴリズムをもちいてデータベースシステムを構築する際に注意すべきことがらについて述べる。5章では、試作したシステムのベンチマークを行なう際にもちいた関係と質問について簡単にふれた後、ベンチマークの結果について報告する。最後の章は結論である。

2．基本概念

2.1 関係データベース演算

関係Rの属性の集合をX、組tのXに含まれる属性の値をt[X]とする。また、AとBをそれぞれ関係R、Sの属性とし、u[B]を組uの属性Bの値とする。三つの基本的な関係データベース演算は次のように定義される。

(i)選択：関係RのAに関する選択をR[Aθc]と書くことにすると、それは次のように定義される。

$$R[A\theta c]=\{t \mid t[A]\theta c,\ t\in R\}$$

ここでθは=、≦、≧、<、>、≠のいずれかであり、cとt[A]は比較可能であるとする。

(ii)射影：関係RのXに関する射影をR[X]と書くことにすると、それは次のように定義される。

$$R[X]=\{t[X] \mid t\in R\}$$

射影の形式的な記述は上のように表現されるが、質問言語のSQLでは二つのタイプの射影質問が提供されている。一つは結果の関係の組はユニークであることを保証するものであり、DISTINCTという句で表現される。もう一つは結果の関係の組は必ずしもユニークではないものである。前者を非冗長射影、後者を単純射影と呼ぶことにする。

(iii)結合：関係RとSの属性AとBに関する結合をR[AθB]Sと書くことにすると、それは次のように定義される。

$$R[A\theta B]S=\{(t,u) \mid t[A]\theta u[B],\ t\in R,\ u\in S\}$$

ここでθは=、≦、≧、<、>、≠のいずれかであり、t[A]とu[B]は比較可能であるとする。もしθが=ならば、等結合と呼ばれ、等結合において重複する結合属性を除いたものを自然結合という。

2.2 ベクトル計算機の概要

ベクトル計算機はベクトルデータ(配列)に対する演算をパイプライン演算器で高速に実行する計算機のことである。現在のベクトル計算機は浮動小数点数だけでなく、整数、論理型データなどもベクトル処理の対象としており、単純な四則演算だけでなく、論理演算やシフト演算などもベクトル処理される。また、マスク演算機能を備えているので、IF文によって制御される演算もベクトル処理される(図1)。

```
      DO 10 I=1,N
         IF (A(I).EQ.0) THEN
            C(I)=B(I)
         ENDIF
   10 CONTINUE
```

図1：条件文を含むDOループ

パイプライン演算器では複数のベクトルデータに対する演算がオーバラップして実行されるので、各演算が並列に実行できるものでなければベクトル計算機上での実行には適さない。総和演算、最大値・最小値を求める演算、ベクトル収集操作などは並列性がないので本来はベクトル処理できないが、現在のベクトル計算機ではこれらのマクロ命令を実行する特別のハードウェアが用意されているのでこれらもベクトル処理される。

また、主記憶とのデータの転送速度が重要であることから、主記憶上のデータに対して連続アクセス、等間隔アクセス、リストベクトルアクセス(ベクトルデータを指標とする間接アクセス)の3種類のアクセス方法が提供されており、これらもロード/ストアパイプラインでベクトル処理される(図2)。最後のリストベクトルアクセスはテーブルルックアップのような操作をベクトル処理するのに有用である。

ベクトル計算機の最大性能は非常に高く、汎用計算機の10倍以上の速度でプログラムを実行することができるけれども、その性能を十分に引き出すためには次の二つの点に注意をしてアルゴリズムの設計及びプログラミングを行なう必要がある。

1) ベクトル化率

ベクトル命令で実行される演算の、プログラム全体の演算に対する割合をベクトル化率という。ベクトル計算機でプログラムを実行した場合でも、すべての処理がパイプラインでベクトル処理されて高速になるわけではなく、ベクトル処理できない演算はスカラ処理される。ベクトル計算機のスカラ

```
    DO 20 I=1,N
       A(I)=B(I)
 20 CONTINUE
```

(a) 連続アクセス

```
    DO 30 I=1,N,K
       A(I)=B(I)+C(I)
 30 CONTINUE
```

(b) 等間隔アクセス

```
    DO 40 I=1,N
       A(I)=B(L(I))
 40 CONTINUE
```

(c) リストベクトルアクセス

図2：3種類のベクトルアクセス

処理の性能は汎用計算機の性能と同じであるので、ベクトル化率が低いとベクトル計算機で実行してもあまり速くならない。例えば、ベクトル化率が50%ならばベクトル処理される部分が0秒で実行されても、全体では高々2倍の速さにしかならない。それでベクトル化率を高くすることが高速化を達成する上で重要になる。

2) ベクトル長

ベクトル命令のオペランドの長さをベクトル長という。パイプライン演算器では定常状態のときは1サイクルごとに演算結果が得られるが、パイプライン演算器にデータを転送する時間はパイプラインの1サイクルの時間に比べてかなり大きいので、演算を開始するときにはパイプラインの立上りのオーバヘッドが発生する。したがって、ベクトル長が短いとこのオーバヘッドを吸収できないのでベクトル処理してもあまり速くならない。大雑把に言って、ベクトル計算機の性能を十分に引き出すためにはベクトル長は数百以上必要である。

ここで、プログラムのできるだけ多くの部分をベクトル命令で実行することをベクトル実行と定義し、すべての部分をパイプライン演算器を使わずにスカラ命令で実行することをスカラ実行と定義する。また、あるプログラムがベクトル計算機での実行に適しているかを評価するために、ベクトル実行での計算速度の、スカラ実行での計算速度に対する割合を加速率と定義する。高い加速率は、ベクトル計算機の性能が十分に引き出されていることを意味し、低い加速率の場合はそうではないことを示す。

上で述べたように、ベクトル化率を高くすることと、ベクトル長を長くすることはベクトル計算機の高い性能を引き出す上で重要である。逆に、これらの二つの条件が満たされれば、ベクトル計算機の高い性能が得られることになる。

3．関係データベース演算のベクトル化アルゴリズム

3.1 選択

選択はデータベース上でのデータ処理において最も頻繁に使われる質問の一つであるので、これを高速に実行することはデータベースシステムのトータルな性能を改善する上で非常に重要である。ベクトル計算機はベクトル収集操作の機能を提供しており、このベクトル収集操作はある配列からある条件を満たす要素だけを選択し、それを別の配列に挿入する。図3はこの機能をFortranのプログラムで利用する方法を示している。このプログラムコードは、配列Aの要素でその値がゼロより大きいものを集めて配列Bに挿入する操作を表現している。配列Aを入力の関係、配列Bを結果の関係と見なすと、この操作は明かに選択操作そのものに対応しており、したがって選択質問はベクトル計算機のこの機能によって高速に実行される。ほとんどすべての処理がベクトル命令で実行され、そのベクトル長は入力の関係の組の数に等しくなるので、ベクトル計算機の高い性能が得られることになる。

```
    K=0
    DO 10 I=1,N
       IF (A(I).GT.0) THEN
          K=K+1
          B(K)=A(I)
       ENDIF
 10 CONTINUE
```

図3：ベクトル収集操作

3.2 ソーティング

ソーティングは重要なデータベース演算である結合、選択、一貫性制約などを実現するのに有用であるだけでなく、データの分散、最大値・最小値などを解析するのにも非常に有用である。石浦らは基底法[6]と彼らが最初に開発した番地計算基底法がベクトル計算機上での実行に適していることを明かにした[7]。それで我々は彼等の結果を利用することにする。基底法は最下位ビットから最上位ビットまで各桁の値によってグループ分けを順次行なっていくソートの方法である。計算機上ではデータは二進数で表現されるので、グループ分けはベクトル計算機のベクトル収集命令によって二つのグループに分けることによって行なわれる。このアルゴリズムの計算量はデータの数とキーの長さの最大値の積に比例するので、キーが広い範囲に分布していると計算量が大きくなる。番地計算基底法はそのような場合の基底法の非効率を避け、しかも浮動小数点数さえもソートできるようにするために開発された。このアルゴリズムは次の三つのフェーズからなっている。

番地計算フェーズ：元のキーの長さの最大値よりも小さくなるように、ソートされるデータのラベルが計算される。ラベルは元のキーの大小関係を保存しなければならないので、この計算は順序つきハッシュでなければならない。この計算はベクトル命令で実行される。

基底法フェーズ：データは元のキーの代わりにラベルによってソートされ、ラベルの長さの最大値は小さくなっているので元のキーをもちいる場合よりも効率的にソートが行なわれる。しかし、ラベルが同じになるデータはこのフェーズではまだソートされていない。

局所ソートフェーズ：データは奇遇置換法、つまり並列バブルソートによってソートされ、このソートは隣り合うデータの並列比較と交換によって実行される。前章で示したように、現在のベクトル計算機はFortranのDOループに含まれている条件文で表されるマスク演算の機能を備えているので、その比較と交換はベクトル命令によって実行される。

基底法と番地計算基底法の両方の場合において、ほとんどすべての過程はベクトル命令によって実行され、そのベクトル長は常にデータの数に等しくなる。したがって、ベクトル計算機の高い性能が得られることになる。

表1と表2はこれらのアルゴリズムをベクトル計算機FACOM VP-200上で実行した結果を示している[7]。表1ではスカラ実行とベクトル実行のCPU時間と加速率が示されており、10倍以上の高い加速率はこれらのアルゴリズムがベクトル計算機上での実行に適していることを示している。また、これらはキーの大きさが$2^{15}-1$の場合のCPU時間であるのに対して、表2はキーの大きさが$2^{31}-1$の場合のベクトル実行のCPU時間を示している。これらの結果から、キーの大きさの最大値

表1：各アルゴリズムの処理CPU時間[7]

	要素数	処理CPU時間(μsec)		加速率
		スカラ実行	ベクトル実行	
基底法	2^{10}	11300	1020	11.1
	2^{14}	214000	14300	15.0
番地計算法	2^{10}	14000	950	14.7
	2^{14}	347000	15900	21.8

表2：大きな要素数に対する処理CPU時間[7]

		処理CPU時間(m sec)	
		基底法	番地計算基底法
要素数	2^{15}	52.8	34.1
	2^{16}	105.6	71.9
	2^{17}	211.3	155.0

が大きい場合には、番地計算基底法は単なる基底法よりも高速であることが分かる。

3.3 射影

3.3.1 単純射影

２章で定義したように、単純射影質問は結果の関係のタプルがユニークであることを保証しない。この場合、通常のディスクベースのデータベースシステムに関しては自明なアルゴリズムしか存在しない。すべてのタプルは、幾つかの属性の値が消去されながら入力の関係から出力の関係へ単に複写されるだけである。ベクトル計算機に関しても、このような射影質問は複写操作のみによって実行される。２章で述べたように、ベクトル計算機はロード/ストアパイプラインによって高速なロード/ストア操作を実現している。まず、タプルは主記憶上の関係からベクトルレジスタへ複写され、次にそこから主記憶上の結果の関係へ複写される。しかも、ベクトル長はタプルの数に等しくなる。したがって、この場合にもベクトル計算機の高い性能が得られることになる。

3.3.2 非冗長射影

２章で定義したように、非冗長射影は結果の関係の組がユニークであることを保証するが、前節で示した単純射影の場合と同じように、幾つかの属性の値を消去することはロード/ストアパイプラインによって効率よく実行される。時間を消費し、問題となるのは重複するタプルを消去する部分である。これを実行するためにはどのタプルがユニークで、どのタプルが同じで、そしてこの同じタプルのどれを残すかを決めなければならない。ソーティングを行なうと重複するタプルが隣接する位置におかれるので、通常のディスクベースのデータベースシステムの場合、重複するタプルを見つけるためにソーティングが行なわれる。しかし、ソーティングの計算量は、タプルの数をｎとすると、O(nlog n)であり、十分効率的であるとは言えない。我々はベクトル計算機上の主記憶データベースを考えているので、重複するタプルを見つけるのにソーティングよりも良い方法であるハッシュをもちいることができる。重複するタプルは常に同じハッシュ値をとるので、すべてのタプルをハッシュテーブルに登録していくとそれ等は容易に見つけることができる。

ハッシュをもちいる場合には衝突、つまり二つのタプルが異なっているにもかかわらず同じ値をとる、という問題がある。したがって、ハッシュをもちいる場合、タプルを挿入する前にまずテーブルを引いて衝突が起こっているかどうかを調べる必要がある。もし衝突していれば、別のハッシュ値を計算してテーブルを引いて調べる操作を繰り返さなければならない。すぐにわかるように、この処理は各タプルごとにシーケンシャルに行なう必要がある。しかしながら２章で示したように、ベクトル計算機上で実行される計算は並列に実行可能なものでなければベクトル計算機の高い性能を達成することはできない。

越智らはBDD(Binary Decision Diagram、二分決定グラフ)をベクトル計算機上に実現する際にこの問題に直面し、全体の処理を次の三つのフェーズに分割して解決した[6]。

表の検索と挿入フェーズ：各タプルのハッシュ値が計算され、もし対応するテーブルのエントリが空ならそこへ挿入される。

検査フェーズ：各タプルが実際にテーブルに挿入されたかどうかを決定するためにテーブルが調べられる。

衝突処理フェーズ：対応するテーブルのエントリが空でなかったか、他のタプルの挿入によって上書きされることによって、実際には挿入されなかったタプルがテーブルに挿入される。

この方法をもちいることによって越智らは強力なBDDの処理系をベクトル計算機上に実現した。

この節ではこの方法を拡張することによって射影質問のベクトル化アルゴリズムを構成し、次の節では結合質問のアルゴリズムを構成する。

越智らのアルゴリズムの場合と同様に、すべてのタプルの並列挿入がこのアルゴリズムの最初の段階で行なわれる。アルゴリズムの最も重要な点は衝突の取り扱いであるこの時、二つのタプルが同じである場合と、二つのタプルは異なっているがそれらのハッシュ値が同じである場合の二つの場合に起こる。いずれの場合においてもテーブルに挿入されているタプルはアルゴリズムを実行した後でも残しておくべきである。なぜならば、それらは真にユニークなタプルであるか、あるいは重複するタプルを代表しているタプルであるからである。もし、衝突のために実際に挿入されていないタプルが存在するならば、それはどちらの場合に含まれるかを決定しなければならない。これらの二つの場合を区別するために、最悪の場合にはタプル全体を比較する必要がある。しかし、パイプライン演算器は数十バイトの二つのタプルを一度に比較することはできないので、属性ごとか、文字列の４バイトごとに行なわなければならない。アルゴリズムの形式的な記述は次のようになる。

アルゴリズム１：重複するタプルの除去

step 1 いくつかの属性を消去した関係の組の集合をＴとする。またＴ´=U=U´=ϕとする。
step 2 ハッシュテーブルを初期化してから、各要素の属性の値からハッシュの計算を行い、ハッシュテーブルにすべての要素を登録する。その際ハッシュの衝突は無視してハッシュテーブルに重ね書きを行う（並列書き込み)。
step 3 このハッシュテーブルを引いてＴの各要素が実際に登録されているかどうか調べる。要素ｔがハッシュテーブルに登録されていれば、U=U∪{t}とし、もしハッシュテーブルに登録されていなければ、Ｔ´=Ｔ´∪{t}とする
step 4 Ｔ´の要素ｔ´に対して、対応するハッシュテーブルのエントリに登録されている要素と、ある一つの属性の値について比較を行う。もし異なっていれば、Ｔ´=Ｔ´-{ｔ´}、U´=U´∪{ｔ´}とする。

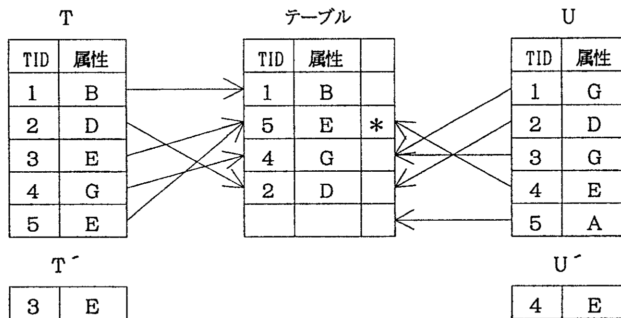

図4：結合演算

この操作をすべての属性について行う。最後までT´に残った要素はハッシュテーブルに登録されている要素と重複している要素なので取り除く。

step 5 U´=φならば終了。そうでなければ、T=U´、T´=U´=φとしてstep2 へ戻る。

ユニークなタプルと、重複するタプルを代表するタプルは、このアルゴリズムを実行した後、集合Uの要素として得られる。step 2からstep 5の過程はベクトル命令で実行でき、そのときのベクトル長は入力の関係のタプルの数に等しくなる。したがって、ベクトル計算機の高い性能が得られることになる。

3.4 結合

結合質問の処理の計算量は大きく、それを実行するための多くのアルゴリズムが研究されている。結合質問を処理するための主な方法として、入れ子走査法、ソートマージ法、ハッシュ法がある。通常のディスクベースのデータベースシステムの場合、ソートマージ法はこれらの三つの中で最良であり、低度の並列入れ子走査法よりも良いと考えられている[2]。しかしながら近年、大容量の主記憶が利用可能である場合の、ハッシュに基づく多くの結合アルゴリズムが研究されており、それらのアルゴリズムの計算量は基本的には関係のタプルの数に対して線形である。それゆえこれらのアルゴリズムは、入れ子走査法やソートマージに基づくアルゴリズムよりも高い性能を示すことが期待できる。我々の考えているベクトル計算機上の主記憶データベースは大容量の主記憶を備えているので、結合演算のアルゴリズムとしてハッシュに基づくアルゴリズムを選択することにする。その基本的なアイディアは射影演算で重複するタプルを除くアルゴリズムのものと同じである。つまり、まずすべてのタプルを衝突を無視してテーブルに並列に書き込み、次にすべてのタプルを実際にテーブルに書き込まれたものと、上書きによって消されてしまったものの二つのグループに分ける。しかし、結合演算において一つのタプルが複数のタプルと結合する場合を考慮しておかなければならない。テーブルに書き込まれているタプルが、もう一つの関係に含まれる複数のタプルと結合する場合には何も問題はない。問題となるのはテーブルに挿入した関係に含まれる複数のタプルが、もう一つの関係に含まれる一つのタプルと結合する場合である。ハッシュに基づくベクトル化結合アルゴリズムはハッシュ値の衝突を無視し、タプルをテーブルに重ね書きをする。したがって、テーブルのあるエントリでハッシュの衝突があったことを示す仕組みを用意しておく必要がある。そこで、アルゴリズムの二つめのステップで衝突を検出した場合には、そのエントリに印を付けることにする(図4)。アルゴリズムの形式的な記述は次のようになる。

アルゴリズム2：結合演算

step 1 関係RとSの組の集合をそれぞれT、Uとする。またT´=U´=φとする。

step 2 ハッシュテーブルを初期化してから、Tのすべての要素をハッシュテーブルに登録する。その際ハッシュの衝突は無視してハッシュテーブルに重ね書きを行う(並列書き込み)。

step 3 このハッシュテーブルを引いてTの各要素が実際に登録されているかどうか調べる。ハッシュテーブルに登録されていない要素tが存在する場合、そのことを示す記しを対応するハッシュテーブルのエントリに付け、T´=T´∪{t}とする。

step 4 Uのすべての要素uについてこのハッシュテーブルを引いて結合する。そのとき、対応するハッシュテーブルのエントリにstep 3で付けた記しがついている場合、U´=U´∪{u}とする。

step 5 U´=φならば終了。そうでなければT=T´、U=U´、T´=U´=φとしてstep 2 へ戻る。

step 2からstep 4のほとんどすべての過程はベクトル命令で実行でき、そのベクトル長は入力の関係のタプルの数に等しくなる。したがって、ベクトル計算機の高い性能が得られることになる。

また、step 2ではTとして二つの入力の関係のどちらを選ぶこともできるが、各々の場合の効率の違いを考慮しておく必要がある。アルゴリズムのstep 4での衝突はそれほどでもないが、アルゴリズムのstep 2での衝突は、アルゴリズムの終了を遅らせるのであまり望ましくない。それで、キー属性とキーでない属性の結合の場合は、結合属性がキーである方の関係をRに選ぶのがよい。それ以外の場合には、タプルの数が少ない方の関係をRに選ぶのがよい。

4．実現

前章で述べたアルゴリズムを使って、我々は京都大学大型計算機センターのベクトル計算機FACOMVP-2600(主記憶512MB)上に主記憶データベースのプロトタイプを実現した。それらのアルゴリズムの他に、システムを改善するために重要な三つの点を以下に述べる。

4.1 記憶配置

データベースシステムは常に大量のデータを記憶しており、それらはしばしば変更されたり、新しいデータが付け加えられたり、部分的に消去されたりするので、我々のプロとタイプシステムもこれらのことに対処できるものでなければならない。したがって、データ量の増減に対応するために一つの関係のタプルは幾つかのブロックに分割して記憶する必要がある。残念ながら、現在のところ自動ベクトル化コンパイラはFortranのものだけが提供されているので、Fortranでシステムをインプリメントする必要がある。Fortranでは配列だけがデータ構造として許されているので、データは3次元の配列に記憶されることになり、三つの添え字はそれぞれブロックの通し番号、ブロック内でのタプルの相対的な位置、タプ

ル内での属性の相対的な位置を示すことになる。
　質問処理においてすべてのタプルのある属性を操作する場合に、連続アクセスを行なうことが可能になるので、ブロック内でのタプルの相対的な位置を表している添え字を最も左の添え字とするのが適当である。２章で示したように、等間隔アクセスやリストベクトルアクセスといった他のアクセスの方法もロード/ストアパイプラインで実行されるけれども、連続アクセスはそのアドレス計算の容易さからこれら三つのうちで最も高速である。

4.2 中間結果のデータ構造

　質問を処理している際の中間結果のデータ構造はしばしばデータベースシステム全体の性能に影響を与える。現在のデータベースシステムは、質問処理の手段としてTID(Tuple IDentifier、タプル識別子)を使うことはほとんどないが、主記憶データベース上ではどのデータも容易にアクセスすることができるので、我々は中間結果を記憶しておくのにTIDを使うことにした。ソートするすべてのデータが主記憶上におかれている内部ソートの場合と同じように、質問処理の際にタプルを移動したり複写したりすることは、ポインタを使って間接的にデータにアクセスするよりも計算量が多いと考えられるので、それらはこの場合には良い方法ではない。
　したがって、３章で述べたすべてのアルゴリズムは、どのタプルが結果の関係ができているのかを決定するフェーズと、入力の関係から結果の関係へタプルを実際に複写するフェーズの二つのフェーズからなることになる。すべての中間処理は、中間結果のタプルのTIDを使って行なわれる。これは我々の主記憶データベースのプロトタイプでの、質問処理の基本的で重要な技術である。

4.3 ハッシュテーブルの大きさ

　射影と結合のアルゴリズムでもちいられているハッシュテーブルの大きさを適切なものにすることは重要である。通常のハッシュテーブルの大きさは、挿入するデータの数の２倍以上にするべきであると考えられている。しかし、我々のアルゴリズムでもちいているハッシュテーブルはそれとは少し異なるので、新たに解析を行なう必要がある。
　どちらのアルゴリズムも入力の関係のすべてのタプルに対して走り、テーブルに挿入されなかったタプルに対してその主要な部分を繰り返す。そこで、何個のタプルが衝突のためにテーブルに挿入されずにアルゴリズムの後のサイクルに残されるかを見積もることにする。
　次の記号を使うことにする。

　m：テーブルのエントリの数
　n：テーブルに挿入するデータの数
　r：データの数のテーブルのエントリの数に対する割合 $(=n/m)$

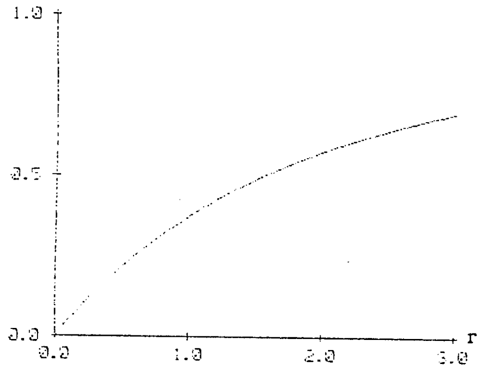

図5：残っているタプルの割合

　データの挿入において、ある一つのデータは確率 $1-1/m$ で、テーブルのあるエントリには書き込まれない。ゆえに、すべてのデータの挿入の間に、確率

$$(1-1/m)^n=(1-1/m)^{rm}=e^{-r} \quad (m\to\infty とする)$$

でテーブルのあるエントリにはデータが一つも書き込まれない。したがって、すべてのタプルを挿入した後の空のエントリの数は me^{-r} であり、データが書き込まれたエントリの数は

$$m-me^{-r}=m(1-e^{-r})$$

である。n 個のデータを挿入した後で、$m(1-e^{-r})$ 個のエントリにデータが書き込まれているので

$$n-m(1-e^{-r})=rm-m(1-e^{-r})=m(r-1+e^{-r})$$

個のデータが上書きされて実際にテーブルに挿入することができなかったことになる。アルゴリズムはこれらのデータに対して後のサイクルを実行することになる。したがって、アルゴリズムの一つのサイクルの後でテーブルに挿入されなかったデータの数は、元のデータの数の

表3: ウィスコンシンベンチマークの関係の仕様[1]

Attribute Name	Range of Values	Order	Comment
unique1	0-(MAXTUPLES-1)	random	unique, random order
unique2	0-(MAXTUPLES-1)	sequential	unique, sequential
two	0-1	random	(unique1 mod 2)
four	0-3	random	(unique1 mod 4)
ten	0-9	random	(unique1 mod 10)
twenty	0-19	random	(unique1 mod 20)
onePercent	0-99	random	(unique1 mod 100)
tenPercent	0-9	random	(unique1 mod 10)
twentyPercent	0-4	random	(unique1 mod 5)
fiftyPercent	0-1	random	(unique1 mod 2)
unique3	0-(MAXTUPLES-1)	random	(unique1)
evenOnePercent	0, 2, 4, ···, 198	random	(onePercent * 2)
oddOnePercent	1, 3, 5, ···, 199	random	(onePercent * 2)+1
stringu1	-	random	candidate key
stringu2	-	sequential	candidate key
string4	-	cyclic	

$$m(r-1+e^{-r})/rm=(r-1+e^{-r})/r$$

倍になる。

図５はこの式のグラフであり、水平軸はｒを表しており、垂直軸は式の値を示している。この図からｒがおよそ1.6の時、式の値はおよそ0.5になることが分かる。したがって、アルゴリズムがその主要な部分を数回繰り返すと、すべてのデータに対する処理は終り、アルゴリズムは終了することになる。次章で示すように、アルゴリズムのベクトル実行では10以上の加速率が得られるので、我々のアルゴリズムは通常のハッシュに基づくアルゴリズムよりも高速であると考えられる。以上の考察から導かれた結果から、入力の関係のタプルの数の1.6倍の大きさがあれば、ハッシュテーブルは十分大きいと言える。

５．性能評価

5.1 ベンチマークの関係

試作したプロトタイプシステムを評価するにあたって、質問処理の性能を評価するために設計されていることから、我々はウィスコンシンベンチマーク[1]をもちいることにした。このシステムはそのすべての質問を処理することはできないが、質問を高速に処理するように設計されているので、この選択は適切であると考えられる。元のウィスコンシンベンチマークの関係は１万個のタプルの大きさがあり、その大きさを変えることは困難であるので、Dewittは大きさが変えられるように最近それに修正を加え、関係の仕様は表３に示すようなものになった。

最初の13個の整数型の属性(unique1からoddonepercentまで)の意味を理解するのは容易である。stringu1とstringu2の二つの文字列属性は、ある手続きをもちいてそれぞれunique1とunique2から計算される。最後の文字列属性のstring4は四つの値を繰り返してとる。詳細は元の論文を参照されたい。

元のベンチマークの関係のタプルの数は１万であり、それは我々のシステムの性能を示すには少し小さすぎるので、我々はそれを10万個のタプルからなるように大きくした。以後をこの関係をHUNDKTUPと呼ぶことにする。また、結合質問のもう一つの関係として、関係HUNDKTUPの最初の１万個のタプルからなる関係APRIMEをもちいることとする。

5.2 ベンチマークの質問

ベンチマークでもちいた、選択、射影、結合、集約の四つの単純な質問を図６に示す。システムの本質的な性能を測るために、これらの質問は索引をもちいずに処理される。

[選択質問]
選択質問は選択率10％でタプルを選択する、つまり入力の関係HUNDKTUPから１万個のタプルを選択する。

[射影質問]
射影質問の射影率は４％であり、入力の関係HUNDKTUPから96％のタプルを重複するタプルとして取り除く。ウィスコンシンベンチマークの元の射影質問は１％の射影率を持つようになっていたが、関係の仕様が変更されたのでそれは４％に変わった。したがって、結果の関係には４千個のタプルが含まれることになる。

[結合質問]
結合質問はHUNDKTUPとAPRIMEの二つの関係の単純な結合であり、APRIMEの大きさはHUNDKTUPの1/10である。それぞれの結合属性はユニークであるので、結果の関係はAPRIMEのタプルの数と同じ数のタプルからなる。

[集約質問]
集約質問として、属性unique1の最小値を求める最小値関数を選んだ。属性unique1に索引があればそれを引くだけで質問は処理されるが、我々の実験では索引は使わずに関係HUNDKTUPのすべてのタプルを走査することによって処理された。このとき、４章で示した記憶の割付がうまく働くので、ベクトル実行の効率が向上している。

```
INSERT INTO TMP
SELECT * FROM HUNDKTUP
WHERE UNIQUE1 BETWEEN 2000 AND 11999
```

(a) 選択質問

```
INSERT INTO TMP
SELECT DISTINCT TWO,FROM,TEN,TWENTY,ONEPERCENT,STRING4
FROM HUNDKTUP
```

(b) 射影質問

```
INSERT INTO TMP
SELECT * FROM HUNDKTUP,APRIME
WHERE HUNDKTUP.UNIQUE1 = APRIME.UNIQUE1
```

(c) 結合質問

```
INSERT INTO TMP
SELECT MIN(UNIQUE1) FROM HUNDKTUP
```

(d) 集約質問

図５：ベンチマークの質問

5.3 ベンチマーク結果

表４は前節で示した質問のベンチマーク結果である。加速率を測るために、経過時間ではなくCPU時間を測定し、スカラ実行のCPU時間をベクトル実行のCPU時間で割って加速率を求めている。最後の集約質問は、他の質問がかなり大きい関係を結果として生成するのに対して、たった一つの整数だけを生成するので、そのCPU時間は他のものに比べてかなり小さいものになっている。一方、射影質問の処理において重複するタプルを除くためには72バイトの大きさのタプル全体を比較しなければならないので、そのCPU時間は他のものに比べてかなり大きくなっている。また、四つの質問のいづれの場合でも、10万個のタプルの関係に対する質問が0.1秒以下で処理されている。そして、ベンチマークの結果から、我々のプロトタイプシステムはすべての質問を10以上の加速率で処理し、ベクトル計算機の性能を十分に引き出していることが分かる。

実は、これらのプログラムはベクトル実行向きに最適化されており、スカラ実行にはあまり適していないので、真の加速率は実験結果の加速率よりも小さいかも知れないが、少なくとも10倍程度にはなっていると考えられる。また、加速率はインプリメンテーションやベクトル計算機によってももちろん変化するが、小さい数値になることはないであろうし、その場合でもこの実験で得られた結果は正しいものになると考えられる。

表4：ベンチマーク結果

	CPU時間(ms)		加速率
	スカラ実行	ベクトル実行	
選択	305	8	38
射影	2225	53	41
結合	686	20	34
集約	26	2	13

6．おわりに

本稿では関係データベース演算のベクトル化アルゴリズム、実現のための技術、プロトタイプシステムのベンチマーク結果を示した。ベンチマークの結果から、このプロトタイプシステムはベクトル計算機の能力を十分引き出しており、結合演算を含めて10万タプルの関係に対する質問をそれぞれ0.1秒以内に処理することが分かる。２章で示したようにベクトル計算機はベクトル長が長いときに限りその高い性能を達成するので、このプロトタイプシステムは単一のタプルの挿入、削除などの小さいデータの更新質問の処理においては高い加速率を示すことはないと考えられる。しかし、バルク更新質問は一度に多くのタプルの属性値を更新するので、高い加速率で処理することが可能かもしれない。このような質問を高い加速率で処理するようにプロトタイプシステムを拡張することは我々の将来の研究課題の一つである。また、幾つかの質問を一度に処理すると全体の効率が改善されることも考えられるので、その可能性を探ることも我々の将来の研究課題の一つである。

ところで、ベクトル計算機はハードウェア技術の進歩に伴って数年ごとに新型が発表されているが、現在のベクトル計算機のパイプラインアーキテクチャはかなり成熟しており、将来ベクトル計算機の新型が発売されても、そのアーキテクチャは現在のものとあまり変わらないと考えられる。したがって、現在のベクトル計算機向きに作られたソフトウェアは将来のベクトル計算機にも適したものとなり、ソフトウェアデータベースシステムはハードウェア技術の進歩にかかわらず、長い期間に渡って生き続けると考えられる。

最後に、データベースシステムの重要な課題である並行性制御とバックアップにも言及しておかなければならない。通常のディスクベースのデータベースシステムでは、質問処理の際にディスク待ちのオーバヘッドがあるので並行性制御は重要であり、それに関して多くの研究がなされてきた。しかし、主記憶データベースではディスク待ちのような時間を浪費する処理はなく、特にベクトル計算機上ではその処理時間は非常に小さい。しかも、並行性制御のための機構は質問処理にオーバヘッドを生じさせるかもしれず、したがって、我々はベクトル計算機上の主記憶データベースでは、通常のディスクベースのデータベースにおけるほど並行性制御は重要ではないと考える。

しかし、バックアップは主記憶データベースにおいても重要であり、その場合電池によるバックアップが行なわれることが多い。主記憶データベースではディスク待ちのようなオーバヘッドを避けることによってその高い性能を達成しているので、オーバヘッドを伴わない電池によるバックアップはこの場合には適切なものであり、あるいはまた[4]の透明なバックアップ機構も適切かもしれない。したがって、これらのバックアップの機構が、ベクトル計算機上の主記憶データベースシステムにおいてもちいられるならば、そのシステムでの質問処理もやはり計算拘束になると考えられ、その加速率は、我々のプロトタイプシステムの加速率と全く異なる、ということはないであろう。

以上のことから、高速なデータベースシステムを実現するためには、現在においても将来においても、ベクトル計算機上の主記憶データベースが有望であると考えられる。

謝辞

種々御議論いただいた上林研究室の皆様に感謝致します。

参考文献

[1] Dewitt,D.J.:The Wisconsin Benchmark:Past,Present, and Future,The Benchmark Handbook,pp.119-165,Morgan Kaufman(1991).

[2] Bitton,D., Dewitt,D.J. and Turbyfill,C.:Benchmarking database systems:A systematic approach,Proceedings of the 1983 Very Large Database Conference (1983).

[3] Ullman,J.D.:Principles of database systems,Computer science Press(1980).

[4] Kambayashi,Y., Takakura,H.:Realization of Continuously Backed-up RAMs for High-Speed Database Recovery,Proceedings of the Second International symposium on Database Systems for Advanced Applications (1991).

[5] Knuth,D.E.:The Art of Computer Programming,Vol.3, Addison-Wesley(1973).

[6] Ochi,H., Ishiura,N. and Yajima,S.:Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing,Proceedings of 28th ACM/IEEE Design Automation Conference(1991).

[7] 石浦菜岐佐,高木直史,矢島脩三:ベクトル計算機上でのソーティング,情報処理学会論文誌,Vol.29,No.4(1988).