

動的処理バケット選択手法に基づくハッシュ結合処理方式とその性能評価†

喜連川 優†† 中山 雅哉††* 高木 幹雄††

ハッシュ操作を用いた結合演算処理方式は、特に大容量のデータベースを扱う場合に効率良いことが示されてきた。しかし、これまでは分割するバケットのデータ分布を事前に予測できるとして性能評価するものがほとんどであり、分割後のデータ分布が予測と異なると不均一になる場合の処理性能については考慮されていなかった。本論文では、このような不均一なデータ分布をとる場合にも有効に処理できる新しいハッシュ結合方式を提案する。本方式は、分割後のデータ分布が変動しても主記憶サイズを越えるバケットが生成されないようにあらかじめ分割するバケット数を多くとる多分割 GRACE 方式の手法に、分割フェイズと結合フェイズの処理の一部をオーバーラップさせることで、入出力コストの削減を図るハイブリッドハッシュ方式の手法を融合したアルゴリズムをとっている。ここで、ハイブリッドハッシュ方式では静的にオーバーラップ処理するバケットを決定していたのに対して、本方式では分割フェイズで動的にこれを選択する方法をとるため（動的デステージング方式）、各バケットのデータ分布に因らず処理効率を高く保つことができる。また、あらかじめ分割バケット数を多くとることにより生ずるフラグメントページに対する入出力コストの増加は、結合フェイズの処理に先立ち、バケットのまとめあげ処理を行うことで解決している。

1. はじめに

関係データベースにおいて結合演算処理は、他の関係代数演算処理に比べて処理コストが高いことが知られており^{1)~7)}、これまでに、ネストループ結合方式^{1)~3)}、ソートマージ結合方式^{4)~5)}、ハッシュ結合方式^{6)~7)}等の種々の処理方式が提案されてきた。この中でもハッシュ結合方式は、大規模リレーションに対しても高速に処理できるため、最近特に注目されている方法である。文献 6), 9) では、ソートマージ方式等の他の結合演算処理方式とハッシュ結合方式の処理コストについて解析的評価が成されており、ハッシュ結合方式の有効性が明らかにされている。また、文献 8) では、試作システムを用いた性能評価が行われ、解析的評価とほぼ同様な結論が示されている。

しかし、これらの論文では、対象リレーションを分割処理する際に、各分割空間（バケット）にほぼ一様にタプルが分散することを仮定しており、各バケットのデータ分布が演算実行前から既知であるとして評価を行っている。通常、ユーザの発行する問合せでは、結合演算だけでなく射影演算や制約演算を伴うものが多く、この仮定は成り立たない。すなわち、各バケットのデータ分布が不均一になり、処理前に想定した分

布とは異なったものになる。本論文で提案する処理手法では、各バケットのデータ分布に応じて動的に処理手順を適合させるアルゴリズムを採用しており、従来のハッシュ結合方式に比べて柔軟性が高く、このような不均一なデータ分布に対しても効率良く演算処理を施すことができる。

以下、2章では、本論文を通して用いる用語の説明を行うとともに、これまでに提案されてきたハッシュ結合方式について概説する。

また、3章では、動的デステージング方式を導入することで、各バケットが不均一なデータ分布をとる場合でも柔軟に演算処理を行うことができる、新しいハッシュ結合方式（動的処理バケット選択型結合方式）のアルゴリズムについて述べている。

そして、4章では、各方式の解析的評価を行い、本論文で提案する方式の有効性を明らかにし、5章でそれらについてまとめている。

2. ハッシュ結合処理方式の概説

本章では、論文を通して用いる用語の定義と、解析的評価を用いる各パラメータについて解説し、従来提案されてきた各種のハッシュ結合方式—単純 GRACE 方式（単純 G 法）⁴⁾、多分割 GRACE 方式（多分割 G 法）⁴⁾、ハイブリッドハッシュ方式（HH 法）^{6), 8), 9)}—について概説する。

2.1 本論文で用いる用語の定義と解説

結合演算処理を施す 2 つのリレーションを各々 L

† A New Hash-based Join Method Using the Dynamic Destaging Strategy—Its Methodology and Performance—by MASAYA NAKAYAMA, MASARU KITSUREGAWA and MIKIO TAKAGI (Institute of Industrial Science, University of Tokyo).

†† 東京大学生産技術研究所

* 現在 豊橋技術科学大学知識情報工学系

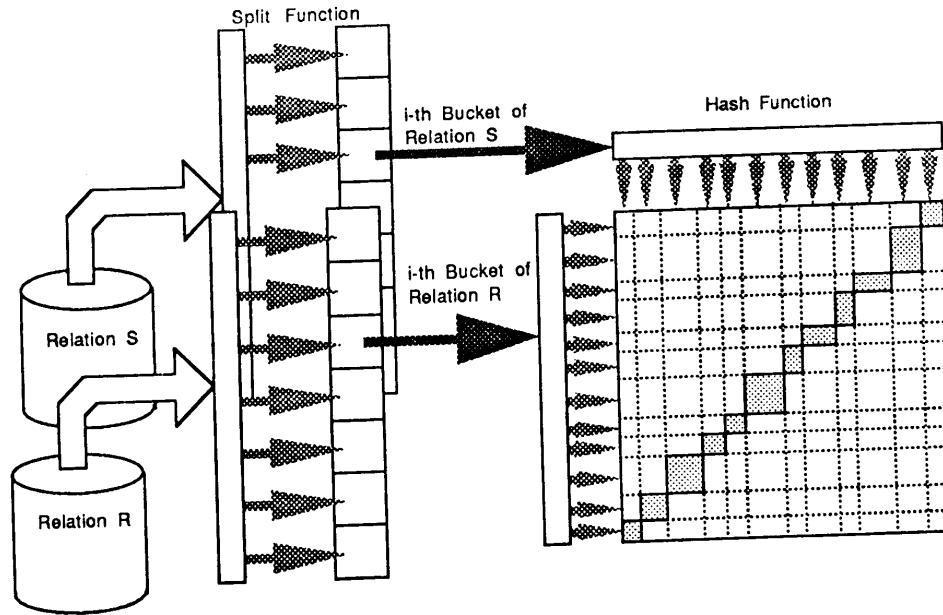


図 1 ハッシュ結合方式による処理手順の概要
Fig. 1 Processing overview of a hash-partitioned join method.

バイトの固定長タプルで構成された R および S とし、各リレーションのタプル数を $\{R\}$, $\{S\}$ と表記する ($\{R\} \leq \{S\}$ とする)。また、演算処理における入出力操作は固定長 (B バイト) のページを単位として行い、各リレーションのページ数をそれぞれ、 $|R|$, $|S|$ と表す。

ハッシュ結合方式では、まず、各リレーションを走査してタプルごとにハッシュ操作を施し (後のハッシュ操作と区別するため、これをスプリット操作と呼ぶ)、 H_s 個のバケットに分割する (分割フェイズ)。リレーションの分割処理が終了すると、各バケットごとに結合演算処理が実行される (結合フェイズ)。この時、結合フェイズでは、演算処理コストを低く抑えるために、分割フェイズとは異なるハッシュ関数を用いてバケット内のタプルをさらに H_h 個に分割して処理を施す方法がとられている (図 1)。

また、演算処理に用いられる主記憶領域は、図 2 に示すように 3 つの部分に分けて利用される。 N ページあるステージングバッファは、分割フェイズでは各バケットを中間リレーションに書き出す際の出力バッファとして用い、結合フェイズでは分割したリレーション R の各バケットをステージングしてハッシュ処理するのに用いられる。入力バッファは、分割フェイズで各リレーションをディスクからステージングするのに用いられ、出力バッファは、結合演算結果を結果リ

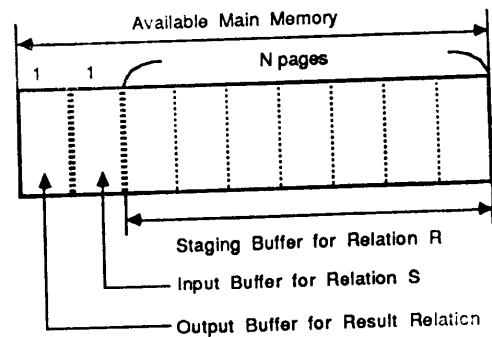


図 2 ハッシュ結合方式での主記憶の利用方法
Fig. 2 Memory allocation for join operations.

レーション RES に書き出す際に用いられる。図に示すように、入力バッファ、出力バッファとも、1 ページ分ずつ用意されている。

まず、分割フェイズにおける分割バケット数 H_s は、次節に示すように各方式で種々の値をとり、各バケットは、 R_i, S_i ($1 \leq i \leq H_s$) と表している。分割バケット数 H_s を小さく取る方式では、あらかじめ想定したとおりにデータが分散しない時、主記憶サイズを超えるバケット (あふれバケット) が生成されることになるが、これらは主記憶サイズ以下になるまで再分割されて演算処理が施される。逆に、分割バケット数 H_s を多く取る我々の方式では、各バケットのサイズが小さくなるため、いくつかまとめて (まとめあげバ

表 1 コスト評価式で用いるパラメータ
Table 1 List of parameters for the cost formulas.

パラメータ	パラメータの意味
INIT _{split}	スプリットテーブルの初期化コスト
INIT _{probe}	ハッシュテーブルの初期化コスト
BACKUP	管理テーブルの退避コスト
RESTORE	管理テーブルの復帰コスト
io	1 ページ当りの入出力コスト
split	1 タプル当りの分割コスト
hash	1 タプル当りの細分化コスト
move	1 タプル当りの移動コスト
comp	1 タプル分の結合処理コスト

ケット) 演算処理が施されることになる。この時のまとめあげバケット数を Hc で表し、各まとめあげバケットは、 kR ($1 \leq k \leq Hc$) で表現する。

これに対して結合フェイズにおける分割数 Hh は、結合演算処理コストを低く抑えるために、極力多く分割することが望ましく、次式のように定められる。

$$Hh = \left\lfloor \frac{B}{L} \right\rfloor * N \quad (1)$$

そして、バケット R_i をハッシュ操作により分割処理した時の各エントリは R_i' ($1 \leq j \leq Hh$) で表す。

このほかに、4章では、表 1 に挙げる各パラメータを用いてコスト式を導出している。

2.2 従来法によるハッシュ結合方式の概説

結合演算処理方式のうち、ハッシュ結合方式は他の方式(ネストループ結合方式、ソートマージ結合方式等)に比べて効率良く処理できるため^{6),8)}、これまでに種々の方式が提案されてきた^{4)~7)}。本節ではこれらの方法のうち、ハッシュ結合方式の基本となる単純 GRACE 方式(単純 G 法)⁴⁾のアルゴリズムと、あふれバケットへの対応策である多分割 GRACE 方式(多分割 G 法)⁴⁾および、シンプルハッシュ方式^{6),8),9)}との融合により入出力コストの削減を図るハイブリッドハッシュ方式(HH 法)^{6),8),9)}の各処理アルゴリズムについて概説する。

2.2.1 単純 GRACE 方式(単純 G 法)

リレーション R が主記憶サイズを超える場合は、次に示す 2 つのフェイズに分けて結合処理を行う。

まず、リレーション R を 1 ページずつ入力バッファに読み込み、各タプルにスプリット関数を適用してバケットに分割する。それぞれのタプルは、対応するバケットのステージングバッファを通して中間リレーションに書き出される。リレーション S についても同様の分割処理が施される(分割フェイズ)。

分割フェイズの処理がすべて終了すると、各バケットごとに結合演算処理を実行する(結合フェイズ)。この時、異なるスプリット値をとるバケット同士は結合される可能性がないため、等しいスプリット値をとるバケット同士のみ結合処理が施される。 i 番目のバケットに対する処理は以下ようになる。

バケット R_i をステージバッファに読み込み、各タプルにハッシュ操作を施してハッシュテーブルを作成する。その後、バケット S_i を 1 ページずつ入力バッファに読み込み、各タプルに同じハッシュ関数を適用してエントリに分け、同一ハッシュ値をとるエントリ同士で結合処理を施す。演算結果 RES_i は、出力バッファを通して結果リレーションに書き出される。

リレーション R が主記憶サイズ以下の場合には、最初から結合フェイズの処理が実行される。

この時結合フェイズでは、ハッシュテーブルを用いて結合演算処理を行うため、各バケット R_i が主記憶サイズ以下となる必要がある。そこで、単純 G 法では分割バケット数 $Hs_{単G}$ を次式のように定めている。

$$Hs_{単G} = \left\lceil \frac{|R|}{N} \right\rceil \quad (2)$$

2.2.2 多分割 GRACE 方式(多分割 G 法)

前節に示したように、各バケットのサイズが主記憶サイズと同程度になるように Hs をとると、各バケットのデータ分布がわずかでも不均一になった時、主記憶サイズを超えるあふれバケットが生成されることになる。例えば、図 3(a)に示すサイズのリレーションは、図 3(b)のように各バケットに均等にデータが分布すると仮定して t バケットに分割されるが、図 3(c)のようにわずかに分布が不均一になるだけで、主記憶サイズ N を超えるあふれバケットが生成されることになる。

あふれバケットに対しては、主記憶サイズ以下になるまで再度分割処理を施す必要があり、演算の処理コストが増大する。そこで、多分割 G 法では分割バケット数 $Hs_{多G}$ を次式のように定めている。

$$Hs_{多G} = N \quad (3)$$

分割バケット数を多くすると、各バケットのサイズを相対的に小さくすることができ、あふれバケットの生成が抑えられる。例えば、図 3(a)のリレーション R を N バケットに分割すると、各バケットのサイズは t ページ程度となる。このため、図 3(c)のように各バケットのデータ分布が不均一になってもあふれバケットは生成されない。

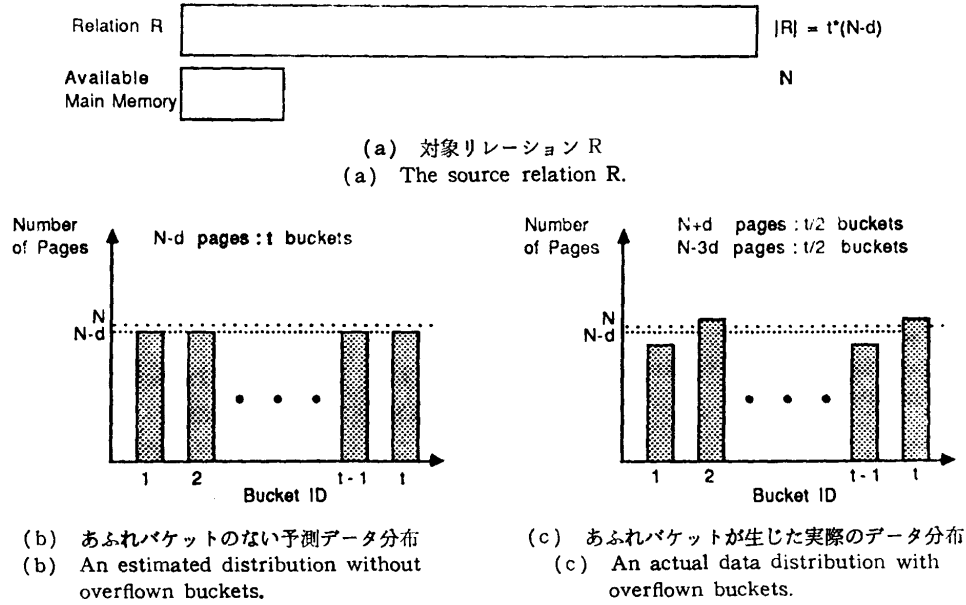


図 3 あふれバケットが生じるリレーション分割の例
Fig. 3 An example data distribution of buckets.

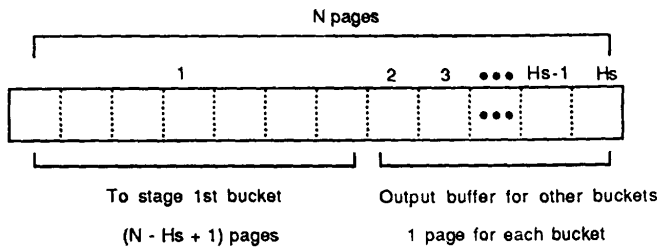


図 4 HH 法におけるステージングバッファの利用法
Fig. 4 Page allocation of staging buffer for hybrid hash method.

2.2.3 ハイブリッドハッシュ方式 (HH 法)

また、単純G法では、分割フェイズと結合フェイズを完全に分離しているため、分割フェイズで利用するステージングバッファは $H_s^{単G}$ ページ分あればよく、残りの $N - H_s^{単G}$ ページは利用されない。HH 法では、これらのページを利用して、リレーション R の分割処理時にバケット R_1 のステージングを同時に行い (図 4)、リレーション S の分割処理にオーバーラップして、このバケットの結合演算処理を実行することで、入出力コストの削減を図っている。これは、結合処理に先立ってリレーションの分割管理を行わないシンプルハッシュ方式^{6), 8), 9)}と、分割処理する単純G法を組み合わせた方式に相当する。

HH 法における分割バケット数 H_s^{HH} は次式で与えられる。

$$H_s^{HH} = \left\lceil \frac{|R| - N}{N - 1} \right\rceil + 1 \quad (4)$$

単純G法や多分割G法では、すべてのバケットが一樣なサイズになると仮定して H_s を決めていたが、HH 法では、分割フェイズとオーバーラップして処理するバケット R_1 のサイズと、他のバケットのサイズは次式に示すように異なると仮定している。

$$|R_1| = N - H_s^{HH} - 1$$

$$|R_i| = \frac{|R| - |R_1|}{H_s^{HH} - 1} \quad (2 \leq i \leq H_s^{HH})$$

また、HH 法では、あふれバケットに対する考慮がなされておらず、 H_s^{HH} が比較的小さい値になるため、各バケットのデータ分布が不均一になるとあふれバケットが生成される。しかし、先に示したように、バケット R_1 の入出力コストを削減する方法をとっているため、小規模な対象リレーションを扱う場合の処理効率が単純G法に比べて高く、図 3 (c) に示すような小規模なあふれバケットを扱う場合の効率低下をある程度低く抑えることができる。

3. 動的処理バケット選択型結合演算処理方式

本章では、HH 法と多分割G法を融合する新しいハッシュ結合方式 (動的G法) について解説する。

3.1 オーバーラップ処理バケットの動的選択

HH 法では、分割フェイズとオーバーラップして処

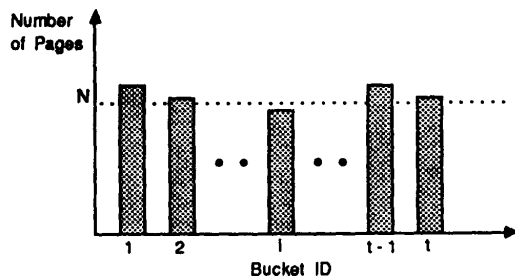


図5 バケット R_i があふれバケットとなるデータ分布例

Fig. 5 Data distribution of buckets with R_i bucket overflow.

理するバケットを静的に決定しているため、図5に示す分布のように、これがあふれバケットとなる場合は、特別な処理手順を用いる必要がある。

これに対して、オーバーラップ処理するバケットに $|R_i| \leq N - H_s + 1$ を満足するバケット R_i を選択できれば、処理アルゴリズムの一般性を保つことができる。我々の提案する動的G法では、以下に示すようにして、動的にオーバーラップ処理バケットを選択しながらリレーションの分割処理を行っている。

リレーション R の各タプルをバケットに分けてステージングする際に、ステージングバッファに空き領域がある限り、必要とするバケットに新しいページを割り当てていく。新たなページが割り当てられなくなると、その時点における各バケットのデータ分布をもとにして、中間リレーションに書き出すバケットが動的に選択される。ここで選択されるバケットは、既に中間リレーションに書き出されたページを持つバケットか、その時点で最も保持ページ数の多いバケットをとる（動的デステージング方式）^{10), 11)}。

この方法により、保持するページ数の少ないバケットは、分割フェイズ終了時まで主記憶中に残されることになり、オーバーラップ処理バケットとして用いることができる。

3.2 分割バケット数 H_s の決定方法とバケットのまとめあげ処理

動的G法では、 H_s バケットに分割した各バケットに対して $(N - H_s + 1)$ ページまでのステージング領域が与えられるため、

$$|R| \leq H_s \cdot (N - H_s + 1)$$

を満足する範囲で H_s を決定すれば、オーバーラップ処理バケットを用意できることになる。

ここで、分割バケット数 H_s を小さくすると、前章

で示したようにあふれバケットが生成されやすく、あふれバケットが生じた場合の入出力コストは、4章に示すように単純G法で約1.25倍程度必要となる。逆に、 H_s を大きくすると、各バケットを中間リレーションに書き出す際に用いられるバッファ領域が多くなり、オーバーラップ処理で削減できる入出力コストが小さくなる。そこで我々は、条件を満たす範囲の中央値をとるように $H_s^{\text{動G}}$ を決定している。

$$H_s^{\text{動G}} = \left\lfloor \frac{N+1}{2} \right\rfloor \quad (5)$$

また、このように分割数を多くすると、分割処理終了時に主記憶中に残されるバケットは1つになるとは限らない。この場合は、これらをまとめてバケット iR として結合演算を施すようにする。

$iR = \{R_i | R_i \in \text{分割処理終了時に全ページが主記憶上にあるバケット}\}$

バケットのまとめあげ処理は、中間リレーションに書き出されたバケットにも適用できる。この場合は、保持するタプルが1ページに満たないフラグメントページに対する入出力コストを削減するとともに、結合フェイズで用いるハッシュテーブルの初期化回数を抑えることができる。

$iR = \{R_i | R_i \in \text{主記憶サイズを超えない範囲でまとめたバケット群 } k\} \quad (2 \leq k \leq H_c)$

3.3 動的処理バケット選択型結合方式の処理手順

本論文で提案する動的G法による結合演算処理アルゴリズムは、以下ようになる（図6）。

まず、リレーション R を $H_s^{\text{動G}}$ 個のバケットに分割しながら主記憶上にステージングする（リレーション R の分割処理）。この時、ステージングに必要なページが確保できなくなると、その時点での各バケットのデータ分布をもとに動的に中間リレーションに書き出すバケットを決定して（動的な書出しバケットの選択）、空きページの確保を行う。リレーション R の分割処理が終了すると、各バケットは主記憶上に全ページがステージングされているものと、中間リレーションに書き出されたページを持つものに分離できる。前者はバケット iR としてまとめて結合演算処理を施すためにハッシュテーブルの作成を行い（ステージングバケットのハッシュ操作）、後者は、主記憶サイズ以下になる範囲でバケットのまとめあげ処理を施しながら各フラグメントページを中間リレーションに書き出していく（バケットのまとめあげと書出し処理）。

```

Join(R,S)
{
    Init_Split_Table();
    While ( Read_Page(R) != EOD ) { /* リレーション R の分割処理 */
        While ( Split(Tuple, &Sid) != EOP ) {
            If ( Free_Space == Empty )
                Dynamic_Destaging_Strategy(); /* 動的な書きだしバケットの選択 */
            Move(Tuple++, Sid); /* 対応バケットへの移動処理 */
        }
    }

    Init_Hash_Table();
    For( Sid = 1 ; Sid <= Hs ; Sid++ )
        If ( Is_Incore_Bucket(Sid) )
            While ( Hash(Tuple, &Hid) != EOB ) /* ステージングバケットのハッシュ操作 */
                Add_Hash_Table(Tuple++, Hid);
            else
                Bucket_Tuning(Sid); /* バケットのまとめあげと書きだし処理 */

    While ( Read_Page(S) != EOD ) { /* リレーション S の分割処理 */
        While ( Split(Tuple, &Sid) != EOP ) {
            If ( Is_Incore_Bucket(Sid) ) {
                Hash(Tuple, &Hid);
                Join_Operation(Tuple++, Hid); /* ステージングバケットの結合演算処理 */
            } else
                Write_Page(Tuple++, Sid); /* 中間リレーションへの書きだし処理 */
        }
    }

    For( Sid = 1 ; Sid <= Hs ; Sid++ )
        If ( ! Is_Incore_Bucket(Sid) )
            Bucket_Tuning(Sid); /* バケットのまとめあげと書きだし処理 */

    For( k = 2 ; k <= Hc ; k++ ) {
        If ( ! Is_Overflow_Bucket(Bid) ) { /* ハッシュ処理を用いた結合処理 */
            Init_Hash_Table();
            Stage_Bucket(kR);
            While( Hash(Tuple, &Hid) != EOB )
                Add_Hash_Table(Tuple++, Hid);
            While( Read_Page(kS) != EOD )
                While( Hash(Tuple, &Hid) != EOB )
                    Join_Operation(Tuple++, Hid);
        } else {
            Backup_Tables();
            Join(kR, kS); /* あふれバケットに対する再帰的処理 */
            Restore_Tables();
        }
    }
}

```

図 6 提案方式の処理アルゴリズム

Fig. 6 The join algorithms of the proposed method.

リレーション S の分割処理では、主記憶上にステージングされているバケット iR と同じスプリット値をとるタプルは、即座に結合処理が施され（ステージングバケットの結合演算処理）、それ以外のスプリット値をとるタプルは、各バケットごとに中間リレーションに書き出される（中間リレーションへの書出し処理）。リレーション S の走査が終ると、リレーション R と同じになるように、バケットのまとめあげ処理が施される（バケットのまとめあげと書出し処理）。これら一連の操作が分割フェイズに相当する。

結合フェイズの処理は、各まとめあげバケットごとに施される。まとめあげバケット iR が主記憶サイズ以下の場合には、これをステージングバッファに読み込んでハッシュテーブルを作成し、まとめあげバケット iS のステージングに伴って結合処理を施す（ハッシュ処理を用いた結合処理）。また、主記憶サイズを超える場合には、再帰的に分割処理が施されて、結合演算処理が実行される（あふれバケットに対する再帰的処理）。これらの操作が結合フェイズに相当する。

4. 不均一なデータ分布に対する性能評価

本章では、前章までに述べてきた各種ハッシュ結合方式について結合演算処理コスト式を導出して解析的に評価を行い、我々の提案する動的G法の有効性についてまとめている。

4.1 各ハッシュ結合方式の処理コスト式

ハッシュ結合方式の結合演算コスト式は、各処理フェイズのコスト式の和で表現することが望ましいが、結合処理の一部が分割フェイズで実行される HH 法や動的G法では式が複雑になるため、本論文では、リレーション（または、バケット）の分割処理に要するコスト $COST_{split}$ と、バケットの結合処理に要するコスト $COST_{probe}$ を用いて全体の処理コスト式を表現する。

分割処理コスト $COST_{split}$ は、バケットのまとめあげを行わない従来の方式では式(6)を用い、動的G法では、式(7)を用いる。

$$\begin{aligned} COST_{split}(R, S) \\ = [|R| + |S| + \sum_{i=1}^{Hs} (|R_i| + |S_i|)] \cdot io \\ + (\{R\} + \{S\}) \cdot (split + move) + INIT_{split} \end{aligned} \quad (6)$$

$$\begin{aligned} COST_{split}(R, S) \\ = [|R| + |S| + \sum_{k=1}^{Hc} (|_k R| + |_k S|)] \cdot io \\ + (\{R\} + \{S\}) \cdot (split + move) + INIT_{split} \end{aligned} \quad (7)$$

また、各バケットの結合処理コスト $COST_{probe}$ も同様に、まとめあげ処理の有無により2つに分けて表現し、従来の方式では式(8)を、動的G法では式(9)を用いている。

$$\begin{aligned} COST_{probe}(R_i, S_i) \\ = (|R_i| + |S_i| + |RES_i|) \cdot io \\ + (\{R_i\} + \{S_i\}) \cdot hash \\ + \sum_{j=1}^{Hh} \{R_i^j\} \cdot \{S_i^j\} \cdot comp + INIT_{probe} \end{aligned} \quad (8)$$

$$\begin{aligned} COST_{probe}({}_k R, {}_k S) \\ = (|{}_k R| + |{}_k S| + |{}_k RES|) \cdot io \\ + (\{{}_k R\} + \{{}_k S\}) \cdot hash \\ + \sum_{j=1}^{Hh} \{{}_k R^j\} \cdot \{{}_k S^j\} \cdot comp + INIT_{probe} \end{aligned} \quad (9)$$

これらの各処理コスト式を用いて、単純G法、多分割G法、HH法、動的G法の各方式の結合演算処理コスト式を導出すると、以下ようになる。

単純G法：

$$\begin{aligned} COST_{join}(R, S) \\ = COST_{split}(R, S) + \sum_{|R_i| \leq N} COST_{probe}(R_i, S_i) \\ + \sum_{|R_i| > N} [COST_{join}(R_i, S_i) \\ + BACKUP + RESTORE] \end{aligned} \quad (10)$$

多分割G法：

$$\begin{aligned} COST_{join}(R, S) \\ = COST_{split}(R, S) + \sum_{|R_i| \leq N} COST_{probe}(R_i, S_i) \\ + \sum_{|R_i| > N} [COST_{join}(R_i, S_i) \\ + BACKUP + RESTORE] \end{aligned} \quad (11)$$

HH法：

$$\begin{aligned} COST_{join}(R, S) \\ = COST_{split}(R, S) + \sum_{|R_i| \leq N} COST_{probe}(R_i, S_i) \\ + \sum_{|R_i| > N} [COST_{join}(R_i, S_i) \\ + BACKUP + RESTORE] \\ - 2 \cdot io (|R_1| + |S_1|) - \{S_1\} \cdot move \end{aligned} \quad (12)$$

動的G法：

$$\begin{aligned} COST_{join}(R, S) \\ = COST_{split}(R, S) + \sum_{|{}_k R| \leq N} COST_{probe}({}_k R, {}_k S) \\ + \sum_{|{}_k R| > N} [COST_{join}({}_k R, {}_k S) \\ + BACKUP + RESTORE] \\ - 2 \cdot io (|{}_1 R| + |{}_1 S|) - \{{}_1 S\} \cdot move \end{aligned} \quad (13)$$

式(10)～(13)に示すように、あふれバケットに対する処理コストは再帰的に定義されており、各方式で Hs の値も異なることから、各方式の処理コストを一般式の上で比較することは困難である。そこで次節では、具体的な例をもとにして、各方式の比較を行う。

4.2 不均一なデータ分布に対する各方式の性能評価

本節では、図3(a)に示すリレーションに対して、図3(b)のように各バケットが一様な分布をとる場合と、図3(c)、図7のように不均一なデータ分布をとる場合の処理コストを、各方式ごとに計算して評価を行っている。

ここで、 $N \gg d \approx t$ の場合には、多分割G法や動的G法では、各バケットのサイズが非常に小さくなり、図7のようにバケットサイズが2倍程度に変動する分布をとってもあふれバケットは生じない。これに対して、単純G法やHH法ではあふれバケットが発生するが、各あふれバケットは再分割処理で主記憶サイズ

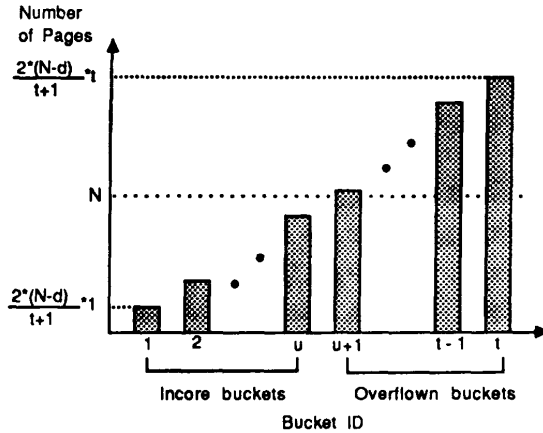


図 7 不均一なデータ分布をとるリレーション分割の例
Fig. 7 An unbalanced data distribution of buckets.

表 2 種々のデータ分布に対する入出力コスト比
Table 2 I/O cost ratio for each data distribution of buckets.

分 布	図 3 (b)	図 3 (c)	図 7
処理方式			
単純 GRACE 法	1.000	1.255	1.364
多分割 GRACE 法	1.005	1.030	1.020
H H 法	0.950	1.078	1.173
動的 GRACE 法	0.974	0.975	0.973

表 3 バケットのまとめあげ処理に伴う性能比較
Table 3 Performance evaluation with bucket size tuning.

分 布	図 3 (b)	図 3 (c)	図 7
まとめあげ			
処理 なし	0.978	0.991	0.978
処理 あり	0.974	0.975	0.973

以下のバケットに分割できると仮定する。

また、結合処理を実行する際に、ハッシュ値の衝突が少ない場合は、全体の処理コストを入出力コストのみで評価することができる。ここでも簡単のため、ハッシュ操作によりすべてのバケットは各エントリに一樣に分割できると仮定して、入出力コストを用いて各方式の評価を行っている。

表 2 には、 $N=1000$, $t=d=10$ とした時の各方式の入出力コストを単純 G 法の均一データ分布時のコストを基準とした相対値を示している。

多分割 G 法や動的 G 法では、ある程度データ分布が不均一になってもあふれバケットが生じないため、処理性能が変化せず、単純 G 法や HH 法に比べて柔軟性が高い。また、単純 G 法と HH 法（または、多分

割 G 法と動的 G 法）を比べることで、結合演算処理の一部を分割フェイズでオーバーラップして処理する方法が、あふれバケットが生成される状況でも有効であることがわかる。

さらに、動的 G 法におけるバケットのまとめあげ処理に伴う効果を調べるために、分割フェイズとオーバーラップして結合処理するバケットのみ、まとめあげ処理を行う場合と、中間リレーションに書き出すバケットにまとめあげ処理を行う場合の入出力コストを算出した結果を表 3 に示す。このように、バケットのまとめあげ処理はフラグメントページを減少させることで、データ分布によらず入出力コストをほぼ一定に保つ効果があることがわかる。

5. ま と め

本論文では、動的に結合演算処理を施すバケットを選択する方法を取り入れた結合演算処理方式（動的 GRACE ハッシュ結合方式）の提案を行い、従来からとられてきたハッシュ結合方式との比較を行って、その有効性を示した。

本方式では特に、分割したバケットのデータ分布が不均一になる場合でも効率良く演算処理を実行することができる点に特徴がある。これは、バケット分割数を大きくとるアルゴリズムを用いることで各バケットのサイズを小さく抑えることが主な要因である。

また、分割したバケットのうち、一部の結合演算処理を分割フェイズにオーバーラップさせて実行する方法をとることで、入出力コストの削減を図ることができる。HH 法では、オーバーラップ処理するバケットを演算処理前に静的に決定していたのに対して、我々の方法では、動的デステージング方式を用いて動的に処理バケットの選択を行っており、従来に増した柔軟性を与えることができる。また、各バケットのサイズが小さい場合は、主記憶上に残るバケットは 1 つになるとは限らないが、この場合はこれらのバケットをまとめてオーバーラップ処理することにしている。このようなバケットのまとめあげを用いた処理アルゴリズムは、分割バケット数の増加に伴うフラグメントページに対する入出力コストの増加や、結合演算処理に用いるハッシュテーブルの初期化コストを抑える効果がある。

文献 12) では、本論文で提案する結合方式をワークステーション上に実装した際の性能評価を行っており、より詳細な評価結果については別稿で報告したい。

参 考 文 献

- 1) Stonbraker, M. et al.: The Design and Implementation of INGRES, *ACM TODS*, Vol. 1, No. 3, pp. 189-222 (1976).
- 2) Astrahan, M. M. et al.: System R: Relational Approach to Database Management, *ACM TODS*, Vol. 1, No. 2, pp. 97-137 (1976).
- 3) Date, C. J.: *An Introduction to Database Systems* (Fourth Ed.), Addison Wesley (1983).
- 4) Kitsuregawa, M. et al.: Application of Hash to Data Base Machine and Its Architecture, *New Generation Computing*, Vol. 1, No. 1, pp. 63-74 (1983).
- 5) Bratbergsengen, K.: Hashing Methods and Relational Algebra Operations, *Proc. of Conf. on VLDB 84*, pp. 323-333 (1984).
- 6) DeWitt, D. et al.: Implementation Techniques for Main Memory Database Systems, *Proc. of SIGMOD*, pp. 1-8 (1984).
- 7) Yamane, Y.: A Hash Join Technique for Relational Database Systems, *Proc. of Conf. on Foundations of Data Organization*, pp. 388-398 (1985).
- 8) DeWitt, D. et al.: Multiprocessor Hash-Based Join Algorithms, *Proc. of Conf. on VLDB 85*, pp. 151-164 (1985).
- 9) Shapiro, L. D.: Join Processing in Database Systems with Large Main Memories, *ACM TODS*, Vol. 11, No. 3, pp. 239-264 (1986).
- 10) 中山ほか: クラスタリング技法に基づく大規模リレーションの結合演算処理方式とその評価, 第36回情報処理学会全国大会論文集, 3F-5(1987).
- 11) 中山ほか: 動的クラスタリング技法を用いた結合演算処理の性能評価, 信学技法, DE 87-21 (1988).
- 12) Nakayama, M. et al.: Hash-Partitioned Join Method Using Dynamic Destaging Strategy, *Proc. of Conf. on VLDB 88* (1988).



喜連川 優 (正会員)

昭和30年生。昭和53年東京大学工学部電子工学科卒業。昭和58年同大学院情報工学専門課程博士課程修了。工学博士。同年、東京大学生産技術研究所講師。現在、同研究所助教授。並列コンピュータアーキテクチャ、データベースマシン、データ工学等の研究に従事。電子情報通信学会、電気学会、IEEE、ACM 各会員。



中山 雅哉 (正会員)

1961年生。1984年慶応義塾大学工学部電気工学科卒業。1989年東京大学大学院工学系研究科情報工学専攻博士課程修了。同年より豊橋技術科学大学知識情報工学系助手。工学博士。データベースシステム、データベースマシンの研究・開発に従事。ACM 会員。



高木 幹雄 (正会員)

昭和35年東京大学工学部電気卒業。昭和40年同大学院博士課程修了。工学博士。同年同大生産技術研究所助教授。昭和54年同大教授。昭和59年機能エレクトロニクス研究センター長(兼)、現在に至る。昭和46~47年カリフォルニア大学(サンタバーバラ)研究員。昭和40年稲田賞、昭和59年TV学会丹羽・高柳賞業績賞。画像処理の研究などに従事。