オブジェクト指向データベースにおける結合演算方式の実装と性能評価

安村義孝†

インデックス付けされていないオブジェクト集合に対して，オブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式の2つの結合演算方式を提案する．これらは、オブジェクト識別子と可変長キーの概念により，従来のハッシュ結合演算方式とソートマージ結合演算方式を拡張した方式である．オブジェクトベースハッシュ結合演算方式では，オブジェクト識別子を利用してステージングバッファ内のバケットの多分割を実現し，オブジェクトベースソートマージ結合演算方式では，実キー値へのポインタを持つソート領域でソート処理を実行する．このようにして，ワーキングメモリサイズを減らすことができる．さらに，結合演算の前処理として，ポインタ検索方式を提案する．これにより，すべてのオブジェクトポインタを繰り返し遷移するような，冗長なディスクアクセスを避けることができる．これらの結合演算方式はわずかなメモリ量で実行可能なため，大量のオブジェクトキャッシュを必要とするオブジェクト指向データベースに有効である．提案した方式をオブジェクト指向データベース管理システムPERCIOに実装し，そのI/Oコストと CPU コストを解析した．ウィスコンシンベンチマークによる性能評価の結果，メモリ量が多い場合には，オブジェクトベースハッシュ結合演算方式がオブジェクトベースソートマージ結合演算方式よりも優れており，メモリ量が少ない場合には，両方式ともそれほど性能劣化がないことが確認された．さらに，関係データベース管理システムと比較すると数倍高速であり，結合キー値のデータ分布に影響しないことも確かめられた．

An Implementation of Join Processing on Object-Oriented Databases and its Performance Evaluation

YOSHITAKA YASUMURA†

For unindexed object collections, we propose two join methods, object-based hash and sort-merge join methods, where conventional hash-based and sort-merge join methods have been extended with the concepts of object identifier and variable-length key value. The object-based hash join method realizes a large number of bucket divisions in staging buffer using object identifiers, and the object-based sort-merge join method executes sort process in a sorting area pointing to real key values. Thus, the methods reduce working memory size. Pointer search method is also proposed for pre-join stage. It reduces repeating traversals of all object pointers and assists these join methods. Since the methods can be executed with tiny main memory, they are effective for object-oriented databases which require a large object cache. The proposed methods are implemented in an object-oriented database management system PERCIO, and their costs for I/O and CPU are analyzed. The performance evaluation using the Wisconsin benchmark shows that the object-based hash join method is more efficient than the object-based sort-merge join method for enough memory, and that both methods have no significant performance degradation for small memory. We also show that these join methods are several times as fast as those for a relational database management system and are not affected by the data distribution of join key values.

1. はじめに

オブジェクト指向データベース管理システム (Object-Oriented Database Management System; OODBMS) はCADやCASE，ネットワーク管理システム，ディレクトリ管理などのエンジニアリングアプリケーションをはじめとする高度なアプリケーション領域で利用されており，近年ではマルチメディア処理，文書管理，生産管理，ワークフロー管理，WWW コンテンツ管理にも応用されつつある．これらのアプリケーションが扱うデータは大量であり，複雑なデータ構造を持つという特長がある．オブジェクト指向データモデルではオブジェクト参照の概念を利用して，複数のオブジェクトに参照関係を持たせることができ，これらのオブジェクトにより複合オブジェクトが構築される．オブジェク

† NEC C&C メディア研究所
C&C Media Research Laboratories, NEC Corporation

ト参照はオブジェクトポインタまたはオブジェクト識別子（Object Identifier; OID）として表現され，データベース内の永続オブジェクトへの参照にも利用されている．オブジェクト間の関連はオブジェクト参照を辿ることでプログラム上で自由に遷移することができる．オブジェクトキャッシュ領域を効果的に使うことにより，大規模なデータベース内の大量のオブジェクト間を高速に遷移することが可能である．また，OODBMSもSQLやOQL等の問合せ言語を提供しており，宣言的な問合せも利用できる．

関係データベース管理システム（Relational Database Management System; RDBMS）はデータベース内のデータにアクセスするために宣言的問合せであるSQL文のみを採用しており，問合せ条件を与えるだけで目的のデータを取得することが可能である．これらの問合せ形式はOODBMSでも重要である[1]．しかし，非手続き的な問合せはアプリケーションプログラムを読みやすくするが，問合せ条件が複雑になるとシステムにとっては処理に必要なデータを容易に探すことができなくなってしまう[7]．特に，データ間の関連を辿るような場合には，RDBMSではテーブル間の何段にも渡る結合演算を動的に実行することになる．それらの結合されるテーブルに大量のデータが存在する場合には，膨大な処理コストがかかってしまう[3]．したがって，多くの研究者が初期の頃から効果的な結合演算のアルゴリズムを提案している[14)15)]．

結合演算処理は，データベースへのアクセスのためにSQLのみを採用している関係データベースでは必須のものであるが，オブジェクト指向データベースでは応用によっては必要でない場合もある[13]．既存のOODBMSには結合演算が実装されていないものがほとんどであるのもそのためである．しかし，OODBMSでは任意のオブジェクト集合（コレクション）が定義可能であり，アプリケーションの実行中に一時的にコレクションが作成されたりすることもあるので，互いに参照関係を持たないコレクション同士に対する問合せを行う場合に結合演算が必要になる．また，あるクラスの全てのインスタンスを表すエクステントのような任意の数のコレクションが実行時に定義されるため，OODBMSでの結合演算はRDBMSに比べてより多様でより複雑であると言える．さらに，OODBMSでの問合せはRDBMSよりも自由度が高い．例えば，別のオブジェクトへの参照ポインタや手続き呼び出し（メソッド）が問合せ条件式内に記述できる．結合演算の処理コストはアルゴリズムとデータ量にかなり依存するため，その実装には細心の注意が必要となる．OODBMSの問合せ処理に関する性能評価の文献は存在するが，従来は結合演算が考慮されていない[2]．

本稿では，大量の要素を持つオブジェクト集合に対するオブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式を提案する．これらの方式はRDBMSの結合演算として開発されているハッシュ結合演算方式とソートマージ結合演算方式に基づいているが，新たな特長としては，1）処理中のデータ参照にOIDを利用; 2）可変長の結合キー値への考慮; 3）バケットの動的なサイズ調整; 4）結合処理の前にポインタ検索処理の実行; 5）問合せ結果として一時オブジェクト集合の生成などがある．これらの特長を利用することで，従来のOODBMSにおける結合演算方式の研究[11)16)]とは異なり，メモリ利用効率の向上と冗長なディスクアクセスの削減によって，よりよい性能向上を図っている．本方式は特長4）の機能を除いてOODBMS PERCIO[17]に実装されている．第2節でPERCIOにおける問合せ処理の概要について述べる．第3節では，オブジェクトベース結合演算方式として、オブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式の処理内容についてそれぞれ説明する．第4節はこれらの結合演算方式のコスト式を示し，PERCIOで実装されたものによる性能評価結果に関する議論をする．最後の第5節で本稿をまとめる．

2. PERCIOの問合せ処理

PERCIOはNEC第二コンピュータソフトウェア事業部とNEC C&Cメディア研究所とで共同開発された商用のOODBMSであり，ODMG（Object Database Management Group）の標準化案[4]の仕様にしたがって開発され，いくつかの拡張を施してある．PERCIOのデータモデルはプログラミング言語C++に基づいているが，可変長クラスや動的スキーマ進化などの柔軟性も持っている．例えば，インスタンスがデータベースに格納された後でも，そのスキーマ定義（クラス定義）を動的に更新することが可能である．これらの機能を利用するために，PERCIOのデータ定義/データ操作を行う言語であるPERCIO/C++が提供されている．本節では，PERCIO/C++とPERCIOにおける問合せ処理について紹介する．なお，PERCIOはSQLやJava言語でも利用できる．

2.1 PERCIO/C++

PERCIO/C++はPERCIOのためのデータベースプログラミング言語であり，言語仕様はC++を拡張した形式になっている．PERCIO/C++には次のような

特長がある．

(1) C++と高い互換性があり，データベース内の永続オブジェクトは，ヒープやスタック上の一時オブジェクトと同様に扱うことができる．さらに，クラス定義により任意の型のオブジェクトをデータベースに格納することができる．

(2) 集合演算操作やコレクション（バッグ，リスト，可変長配列など），インデックス，トランザクションなどの概念が言語に統合されている．

(3) 動的スキーマ進化や埋め込み可変長配列，可変長クラスなど，動的にデータベースの構造が更新されるものを定義することができる．

(4) オブジェクト指向データベースにビューの概念が導入されている．

(5) 宣言的な問合せ構文をアプリケーションプログラム中に記述することが可能である．

PERCIOで利用されるクラスには固定長と可変長がある．前者はC++クラスと完全に互換性があり，クラス内のメンバ変数へのアクセスやポインタによるオブジェクト間の遷移はC++と同様に記述可能である．後者は実行時にクラスのサイズを変えられるという柔軟性を持ち，このクラスのインスタンスは可変長になり，埋め込みの可変長配列もクラス内に定義することができる．画像やテキストデータのように，クラス定義時にあらかじめサイズが決められないオブジェクトを扱う場合に有効である．

コレクション内の要素はB+木で実装されており，コレクションはB+木へのポインタによって表現されている．コレクション要素はオブジェクトポインタ（参照型）やC++基本型，構造体，クラス（以上は実体型）などの型を持つ．PERCIOでは，Set，Bag，List，String，Varray，Extentのコレクション型を持っている．これらは`Od_Set<T>`のようなクラステンプレートで定義され，集合演算操作はクラスライブラリとして提供されている．集合演算操作には，要素の追加/削除，先頭要素や末尾要素の取得，次要素や前要素の取得などの基本操作以外に，`select`や`join`，`group`などの集合演算が用意されている．これらの演算ではオブジェクト集合を返却し，それに含まれている要素は指定された条件を満たしている．

PERCIOでは，RDBMSで利用されているビューの概念をOODBMSに対しても導入している[8]．ビューには*selection view*と*join view*，*group view*の3種類がある．これらのビュー機能を利用するためには，クラス定義によってビュークラスを登録しておき，ビュークラスのインスタンスは上記の集合演算で生成される．RDBMSのビューと異なる点は，ビュークラスのインスタンスは元のクラスのインスタンスを直接に参照していることである．したがって，ビュークラスのインスタンスから指されたオブジェクトのメンバ変数を更新すると，データベース内のオブジェクトが更新される．*join view*は複数のクラスを1つのクラスとして見なすことができ，そのインスタンス変数とメソッドは元のクラスのものから継承される．`join`メソッドは，元のオブジェクト間の関係を指定するための，`join`インスタンスと呼ばれる新しいインスタンスの集合を生成して返却する．これらのインスタンスは，`Od_Join`のクラステンプレートから生成されるクラスに属する．

2.2 問合せ処理

PERCIO/C++の問合せ処理では，複数のクラスにわたる複雑な条件や入れ子問合せ，結合などを指定した宣言的な問合せ文が記述できる．このような問合せ文はブロック式と呼ばれる拡張構文を使うことによって，任意のオブジェクト集合に対して発行することが可能である．ブロック式の中には，オブジェクト変数や問合せ条件，問合せ結果の要素順序などが指定される．問合せ条件には，メンバ変数（入れ子構造や配列も含む）やメンバ関数，オブジェクトポインタも記述できる．`select`演算や`join`演算は問合せ条件式を満たすオブジェクトポインタのコレクションである一時的なオブジェクト集合を生成する．

アプリケーションプログラムの実行時に発行された問合せは，PERCIOカーネル内部にある問合せ処理システムによって処理される．ここで，与えられた問合せ条件式を解析して内部表現である条件木が生成される．次に，オプティマイザによって処理コストに基づく最適化を行い，最も処理効率がよくなるように条件木が更新される．あらかじめユーザが設定しておく統計情報も使われることがあり，最終的に問合せ処理の実行計画が立てられる．そして，その実行計画にしたがって適切なデータベースアクセス関数が選択され順次実行される．

データベースアクセス関数は，条件部分木（条件木の一部）のための検索アルゴリズムに相当する．これらは以下のような3種類の検索アルゴリズムに分類できる．

(1) 単純な順検索処理を行う`select`関数

(2) 条件部分木に2つ以上のオブジェクト変数が指定された場合に，結合演算処理を行う`join`関数

(3) 特定のメンバ変数に付与されたインデックスを利用する`select`関数

もしも，問合せ文に出現するメンバ変数にインデックスが付けられていたら，問合せ処理システムは自動的にそれらを利用する実行計画を立てる．データベースアクセ

ス関数は条件木のノードを表現するクラスのメンバ関数として定義されており，条件部分木のルートノードで実行される．

各データベースアクセス関数は中間集合を返却する．中間集合には，条件部分木の条件を満たすオブジェクトへのポインタと，結果集合の要素順序を決めたり他のデータベースアクセス関数の処理に必要となるいくつかのキー（メンバ変数）値が含まれている．問合せ全体の結果集合は，問合せ実行計画の最後のデータベースアクセス関数が実行された後に一括して生成される．この時，最後のデータベースアクセス関数により生成された中間集合の要素はOIDによってソートされ，結果集合のB+木が構築される．ただし，結果集合の要素順が指定されている場合は，OIDの代わりに該当するキー値でソートする．

2.3 問合せ例

PERCIOにおける問合せの例として，個人情報サービスのためのディレクトリ管理システムを考える．ディレクトリ管理システムは任意の資源のディレクトリ情報を管理するためのもので、各資源が持つ属性データにより様々な検索を行うことができる．ディレクトリ管理の対象は数万件から数十万件にもおよぶことがあるため，大量のディレクトリ情報を管理するにはOODBMSが適していると言える．

ディレクトリ管理システムのためのデータベースには，ディレクトリ階層を管理するディレクトリクラス（Directoryクラス），各資源に該当するエントリクラス（Entryクラス），属性データ値を格納する属性クラス（Attributeクラス）などのクラスが登録されているとする．ユーザから与えられたディレクトリパスにより，ディレクトリ階層を辿ってエントリを特定して必要な属性データを取得したり，属性データに対する検索条件により，該当するエントリを検索したりすることができる．以下は，これらの簡単な問合せ例であり，PERCIO/C++のブロック式による結合演算を用いている．

(1) 選択演算

```
Od_Collection<Directory*>* res1 = dir1->
  subordinate.select([
      where strcmp(entry->title,
                   dir2->entry->title) == 0;]);
```

dir1とdir2はDirectoryクラスのインスタンスへのオブジェクトポインタを値とする自動変数であり，dir1->subordinateはDirectoryクラスに定義されているdir1のサブパートを表す集合（コレクションオブジェクト）である．問合せ条件は，dir1->subordinate内のDirectoryクラスのインスタンスから指された個人情報（エントリ）の役職（titleメンバ変数）が，dir2から指された個人情報と同じものを検索するということである．この問合せの結果集合はres1にセットされる．

(2) 結合演算

```
Od_Collection<Directory*>* res2 = dir1->
  subordinate.select([this: D1;
      &dir2->subordinate: D2;
      where strcmp(D1->entry->tel,
                   D2->entry->tel) == 0;]);
```

この問合せは選択演算であるが，条件式に複数の集合があるため内部的に結合演算処理が実行される．ブロック式に現れるD1とD2はオブジェクト変数と呼ばれ，それぞれdir1->subordinateとdir2->subordinateの元集合における任意のオブジェクトポインタのエイリアスである．結果集合には，dir2のサブパートから指された個人情報と同じ電話番号（telメンバ変数）を持つdir1のサブパートへのオブジェクトポインタが格納される．

(3) 複合条件検索

```
Od_Collection<Directory*>* res3 = dir1->
  subordinate.select([this: D1;
      &dir2->superior->subordinate: D2;
      where strcmp(D2->entry->title,
                   "Manager") == 0 &&
        D1->entry->age == D2->entry->age;]);
```

問合せ条件式はAND（&&）またはOR（||）により複数指定することができる．各条件式にはポインタを記述でき，それらのポインタは元集合の各オブジェクトに対するデータベースアクセス関数内で遷移し，処理に必要なキー値がその都度取得される．また，問合せ文を入れ子にすることも可能なため，任意の複合条件検索を行うことができる．

(4) 結合ビュー生成

```
Od_Collection<Od_Join<Directory*, Entry*>*>*
res4 = dir1->subordinate.join([
    this: D; entries: E;
    where strcmp(D->name, E->dir->name) == 0 &&
      strcmp(D->entry->email, E->email) == 0;]);
```

本問合せは結合ビューを生成する問合せ例であり，内部的に結合演算処理が実行される．もし問合せ条件が満たされれば，Directoryクラスのインスタンスと Entryクラスのインスタンスが結合され，1つのインスタンスとして扱われる．このインスタンスの変数とメソッドは，元のインスタンスのものから継承される．

3. オブジェクトベース結合演算方式

PERCIOには，オブジェクトベースハッシュとオブジェクトベースソートマージ，ネストループ，インデックスを利用したものの4つの結合演算方式が実装されている．これらは問合せ処理システム内のデータベースアクセス関数として存在している．本節では，PERCIOでの実装を前提としたオブジェクト指向データベースにおける結合演算方式について述べる．まず，結合演算の前処理として，問合せ条件内のオブジェクトポインタ間を遷移するためのポインタ検索方式について述べる．次に，オブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式のアルゴリズムを説明する．以下の説明では，オブジェクト集合Rとオブジェクト集合Sが結合され，R内の要素数はS内の要素数よりも少ないと仮定する．

3.1 ポインタ検索方式

オブジェクトポインタによる任意の遷移パスを問合せ条件式に指定することができるので，問合せ処理システム側で問合せ条件式のキー値取得のために，問合せ処理実行中にそれらのポインタを遷移しなければならない．しかし，問合せ対象の元集合に含まれる各要素が処理される度に，オブジェクトポインタを遷移するのは高価である．もしも，あるクラスに膨大な数のインスタンスが存在し，複雑に関係付けられてデータベースに格納されている状況で，そのクラスのエクステント集合に関係を辿るような問合せ条件式が指定された場合には，いくつかのページを何度もアクセスしなければならなくなるかもしれない．このような処理は冗長になりがちであり，結合演算処理を行っている間に負荷のかかる処理方式は避けるべきである．

そこで，結合演算の前処理としてポインタ検索方式を提案する．本方式は結合演算対象の元集合に含まれる全てのインスタンスのオブジェクトポインタをサーチし，あらかじめ物理ページ順にそれらのオブジェクトポインタを並べ直しておく．すなわち，それらの要素が物理ページ情報を含んでいるオブジェクトポインタによりソートされる．元集合のOIDと遷移先のOIDのペアがソート処理中に構築され，最終的にポインタ集合が生成される．こうすることで，データが格納されている物理ページ順にアクセスできるようになる．

この方式は，オブジェクト指向データベースにおいてインスタンス属性値を取得するための結合演算方式であるPointer-Based Sort-Merge方式[16]に似ているが，Pointer-Based Sort-Merge方式では問合せ条件式に含まれる遷移パス内の任意の数のクラスに属するインスタンスを操作することができない．本方式のアルゴリズムは以下のように動作する（図1）．

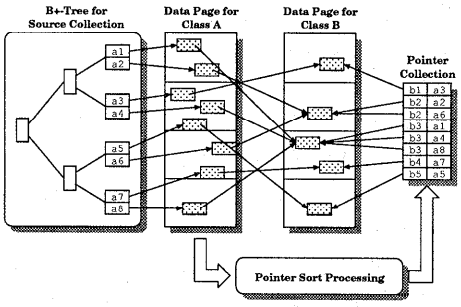


図1 ポインタ検索処理
Fig. 1 Pointer search processing

(1) パスに含まれる1つの遷移のためのポインタ値を元のポインタと共に問合せキャッシュ（問合せ専用のキャッシュ領域）内に配置する．これをその遷移をする全オブジェクトに対して繰り返す．

(2) 遷移先のポインタ値をキーとして，ソートアルゴリズムによりこれらの要素を問合せキャッシュ上でソートし，元のポインタと遷移されるポインタのペアを含んだポインタ集合を生成する．

(3) 以上の処理がパス内の最後の遷移まで繰り返し実行され，結合演算で必要なポインタ集合が生成される．

(4) パス内の最後の遷移の場合には，次の結合に必要なメンバ値を同時に取り出して，ポインタのペアと一緒に格納する．

こうすることで，結合演算の処理中にポインタを遷移するためのメモリ領域が必要なくなり，結合演算の演算領域のために大量のメモリページを割り当てることができるようになる．

オブジェクトベース結合演算では，ポインタ検索で取得されたポインタ集合をそのまま利用する．つまり，データベースにアクセスする代わりに，一時的に生成されたポインタ集合にアクセスすることで，結合キー値取得のためのメモリ領域が抑えられる．また，2.3節の問合せ例のうち，(1)の選択演算はオブジェクトベース結合演算が実行されないが，このポインタ検索方式のみを適用して，データベースアクセスの効率化を行うことが可能になる．

結合演算処理は図2に示すようにいくつかのモジュールから構成されている．結合演算処理のためのデータベースアクセス関数が呼び出される際には，結合される元集合と対応する条件部分木が問合せコントローラによって渡される．ポインタ検索では，条件部分木に指定された全てのオブジェクトポインタを遷移して必要な情

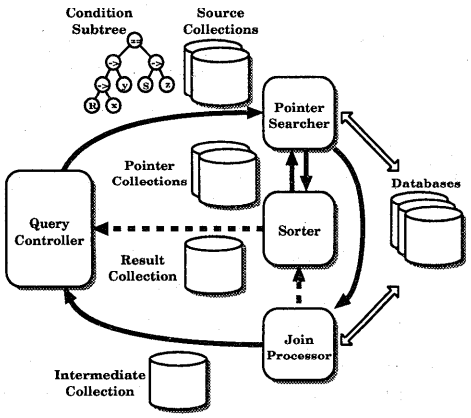


図2 結合演算処理
Fig. 2 Join processing

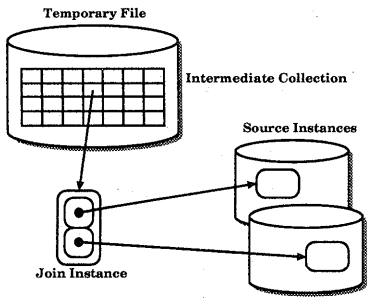


図3 結合インスタンス
Fig. 3 Join instance

報を取得し，ポインタ集合を生成する．このポインタ集合は要素順にソートされている．その後，結合演算処理が実行され，中間集合が生成される．結果の要素は R' または S' の要素か，図3に示したような結合インスタンス（OIDのペア）である．もしも，このデータベースアクセス関数が問合せ処理の最後であれば，中間集合のOIDをソートすることで結果集合のためのB+木が生成され，中間集合の代わりに結果集合が問合せコントローラに返却される．また，問合せ文に順序情報（order_by句）が指定されていれば，結果集合の生成の際に中間集合がそのキー値によってソートされる．

3.2 オブジェクトベースハッシュ結合演算方式

ハッシュ関数を利用した結合演算処理としては今までに数多くのアルゴリズムが提案されている[5)15)]．それらの中で，処理効率がよく比較的実装が容易なものとしてGRACEハッシュ結合演算方式[9)]がある．GRACEハッシュ結合演算方式は元々RDBMSのために考案された方式であるが，オブジェクト識別子を有効に利用してOODBMSに応用したオブジェクトベースハッシュ結合演算方式を開発した．

従来の多くの結合演算方式の研究では，結合するキー値が元集合内で均等に分散されたデータを想定して性能評価を行っていたが，実際にはそのような状況は稀である．そのため，分散された歪み（skew）に順応するために，結合演算の元集合を多数のバケットに分割する多分割方式を採用している．さらに，ビットフィルタや動的バケットサイズ調整などの機能も導入して，処理効率の向上を行っている．

GRACEハッシュ結合演算方式は分割フェーズ（Split Phase）と結合フェーズ（Join Phase）の2フェーズから構成される．本方式のアルゴリズムは以下のように実行される．

分割フェーズ

（1） R 内の全インスタンスについて，結合するキー値のハッシュ値を分割ハッシュ関数によって計算する．これらのハッシュ値はビット値に変換され，ビットフィルタの該当するビットが立てられる．

（2） インスタンスのOID，結合キー値，指定された順序キー値の組から構成される各要素を，結合キー値のハッシュ値により，メモリ（ステージングバッファ）上に展開された一時集合 R' の多数のバケット中に分散して格納する．

（3） R 内の全要素がステージングバッファに格納され，オーバフローバッファが生成されなければ，結合フェーズにおける S 内の要素を突き合わせる処理に飛ぶ．

（4） 生成された一時集合 R' が一時ファイルに書き出される．その際，サイズの小さい複数のバケットを1つのバケットにまとめ上げる．

（5） 以上の処理が S 内の全てのインスタンスにも適用され，一時集合 S' が生成される．ただし，ビットフィルタの処理については，ビット値に該当するビットが立っていなければ，その要素は処理されない．

結合フェーズ

（1） 一時集合 R' の任意のバケット R'_i 内の全要素の結合キー値のためのハッシュ値を結合ハッシュ関数により計算する．

（2） これらの要素をステージングバッファに格納し，いくつかのバケットに分散する．

（3） 展開されたバケット R'_i に対応するバケット S'_i の各要素のハッシュ値を，同じ結合ハッシュ値で計算する．

（4） この要素と同じハッシュ値を持つバケット内

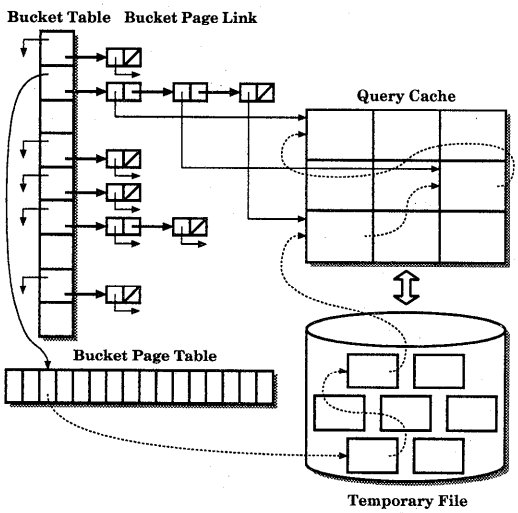


図4 バケット管理
Fig. 4 Bucket management

の全要素に対して結合キー値を比較する．もし結合キー値が同じであれば，これらの要素のペアが中間集合 IM に出力される．

(5) 以上の処理を R' と S' の全てのバケットについて実行する．

結合キー値は可変長データかもしれないが，本方式はOIDを利用しているため多数のバケット分割を実現できる．結合演算処理に必要なメモリページは，ステージングバッファのために利用される問合せキャッシュに動的に割り当てられる．さらに，分割フェーズで小さなサイズのバケットはより大きなバケットにまとめ上げられるため，一時集合の R' と S' にはほんのわずかなフラグメントページしか残らない．上記の2フェーズが実行される間，バケットは図4に示すように管理されている．各バケットはバケットテーブルに登録され，問合せキャッシュ上のメモリページはバケットページリンクとしてリンクされている．一時ファイル（図4での点線）内のものも含めて，全てのページには論理的なリンクも存在しており，その先頭ページはバケットテーブルのエントリから参照される．

実行されるデータベースアクセス関数は問合せ処理システム内の統計情報を利用して選択されるが，あらかじめ元集合の要素数を正確に把握することは不可能である．統計情報は必ずしも最新ではないからである．もしも元集合内にわずかな数の要素しか存在しなければ，GRACEハッシュ結合演算方式の2フェーズによるI/Oコストのオーバヘッドがかなりのものになってしまう．そこで，片方の集合の要素が全てステージングバッファに載った場合には，分割フェーズのステップ3のように単純ハッシュ結合演算方式にしたがって処理を続けるようにしている．GNハッシュ結合アルゴリズム[12]も動的にアルゴリズムを選択する方式であるが，本方式のアプローチではアルゴリズム選択のフェーズを必要としない．

3.3 オブジェクトベースソートマージ結合演算方式

ソートマージ結合演算方式は，RDBMSにおける結合演算処理では最もよく利用されるアルゴリズムである．それをOODBMSに適用したものがオブジェクトベースソートマージ結合演算方式であり，ソートマージ結合演算と同様にソートフェーズとマージフェーズを持つ．これらのフェーズでは，様々なソートアルゴリズム[18]が利用される．本方式は，オブジェクトベースハッシュ結合演算方式では不適切な条件の問合せを行う際に利用される．

アルゴリズムは次のように実行される．

ソートフェーズ

(1) 初めに，問合せキャッシュをデータコンテンツ領域とソート領域に分割する．前者はオブジェクトベースハッシュ結合演算方式と同様に，OIDといくつかのキー値からなる要素を含んでおり，後者はデータコンテンツ領域の要素へのポインタを含んでいる．

(2) R の各要素をメモリに入る分だけデータコンテンツ領域に読み込み，それらへのポインタをソート領域に構築する．

(3) これらの要素は結合キー値によりソート領域内でソートされ，連（run）として一時ファイルに格納される．

(4) 残りの要素も同様に処理され，R 内の全ての要素が連に分割される．

マージフェーズ

(1) R のための一時ファイルに格納された連を n-way マージを利用してマージする．n はデータコンテンツ領域のメモリページ数である．これにより，結合キー値によりソートされた一時集合 R' が生成される．

(2) 以上の処理は S に対しても適用され，一時集合 S' が生成される．

(3) R' と S' のそれぞれにカーソルが定義され，カーソルが各集合を移動しながら結合される要素を同時に検索する．

(4) もし問合せ条件が満たされると，それらの要素が結合され，中間集合 IM に結合インスタンスが出力される．

ソートするためのデータ値よりも要素サイズを小さく

表1 性能解析のパラメータ
Table 1 Parameters used in the analysis

パラメータ	定義
$\|R\|, \|S\|$	R と S のページ数
r, s	R と S の要素数
n_r, n_s	問合せ条件式に指定された R と S のパス数
key	R または S のキーのサイズ
oid	OID のサイズ
$page$	ページのサイズ
f_r, f_s	R と S の選択率
$\|QC\|$	問合せキャッシュのページ数
IO	ディスクページの読み書きの時間
$hash$	R または S のキーをハッシュする時間
$comp_{key}$	R または S のキーを比較する時間
$comp_{oid}$	OID を比較する時間
$move_{key}$	R または S のキーを移動する時間
$move_{oid}$	OID を移動する時間
$swap_{key}$	R または S のキーを入れ替える時間
$swap_{oid}$	OID を入れ替える時間
$join$	結合インスタンスを生成する時間

するために，ソート処理では要素を入れ替えるためにソート領域を利用する．この方式はとりわけ可変長データに効果的である．一方，メモリ領域が小さくて済むように，マージ処理は結合演算処理とは同時に実行しない．もし n がソートフェーズで生成される連の数より少ないと，2フェーズよりも多くのフェーズが必要になるからである．また，最初のソート時に全ての要素がデータコンテンツ領域に載った場合には，マージ処理をスキップすることが可能になる．

4. 性能評価

本節では，PERCIO で実装したオブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式の性能評価について述べる．初めにポインタ検索方式も含めたコスト式による性能解析を行い，ベンチマークテストによる性能評価結果を示す．さらに，RDBMS の結合演算処理でも同様のベンチマークテストを行い，性能を比較する．

4.1 性能解析

オブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式の性能解析を行う．ただし，バケット管理のオーバヘッドやビットフィルタリングなどの細かい処理コストは省略し，一時ファイルへのディスク I/O コストやキー値を比較するための CPU コストなどに注目する．性能解析で利用するパラメータを表1に示す．

まず，ポインタ検索方式に関して考える．オブジェクトのクラスタリングや遷移先のオブジェクトの重複などにより，一般的なコスト式を示すことは不可能であるが，ここでは遷移パスに含まれる各オブジェクト集合が別ストレージに格納されて，それら全てのページにアクセスすると仮定する．ポインタ検索では，遷移するパス毎にそれぞれの集合を読み込んでソートを行い，最終的にポインタ集合を生成するため，コスト式は R と S に対してそれぞれ次のようになる．

$$Cost_{Pointer\ Search\ R} = \sum_{i=1}^{n_r}(|R_i| \times IO) + r\log r \times (comp_{oid} + swap_{oid}) \times n_r + \left\lceil \frac{(oid + key) \times r}{page} \right\rceil \times IO \quad (1)$$

$$Cost_{Pointer\ Search\ S} = \sum_{i=1}^{n_s}(|S_i| \times IO) + s\log s \times (comp_{oid} + swap_{oid}) \times n_s + \left\lceil \frac{(oid + key) \times s}{page} \right\rceil \times IO \quad (2)$$

$|R_1| \sim |R_{n_r}|$ と $|S_1| \sim |S_{n_s}|$ は，それぞれの遷移パスに含まれる各オブジェクト集合のページ数である．このように，演算に必要なキー値を取得するために，データベースへのアクセスが1度で済むことができるようになる．

本稿で検討したオブジェクトベース結合演算アルゴリズムは，元集合 R と S を読み込み，R と S の中の条件式を満たす2つのインスタンスを結合し，結果集合のための B+ 木を生成しなければならない．そのためのコスト式は次のようになる．

$$Cost_{load} = (|R| + |S|) \times IO \quad (3)$$

$$Cost_{join} = join \times f_r r \times f_s s + \left\lceil \frac{oid \times f_r r \times f_s s}{page} \right\rceil \times IO \quad (4)$$

$$Cost_{generate} = (f_r r \times f_s s) \times \log(f_r r \times f_s s) \times (comp_{oid} + swap_{oid}) + \left\lceil \frac{oid \times f_r r \times f_s s}{page} \right\rceil \times IO \times 2 \quad (5)$$

ここで，前処理としてポインタ検索方式を行う場合には，元集合を読み込むコスト式 (3) はコスト式 (1) および (2) と，それぞれで生成されたポインタ集合を読み込むコストの和になる．

オブジェクトベースハッシュ結合演算方式は GRACE ハッシュ結合演算アルゴリズムまたは単純ハッシュ結合演算アルゴリズムを採用している．どちらのアルゴリズムが適用されるかは，初めにステージングバッファ上に展開される R 内のインスタンス数と結合キー値のサイズ

に依存する．GRACEハッシュ結合演算アルゴリズムは，RとSの全インスタンスを複数のバケットに分割して一時集合R'とS'を生成すること，再びR'の要素を使ってステージングバッファを構築すること，S'の要素をそれに突き合わせることから構成されている．コスト式はそれらの処理の和になる．

$$
\begin{aligned}
&Cost_{Object-based\ Hash\ Join} = \\
&Cost_{load} + Cost_{join} + Cost_{generate} + \\
&(hash + move_{key} + move_{oid}) \times r + \sum_{i=1}^{|QC|} \left(|R'_i| \times IO \right) + \\
&(hash + move_{key} + move_{oid}) \times s + \sum_{i=1}^{|QC|} \left(|S'_i| \times IO \right) + \\
&\sum_{i=1}^{|QC|} \left\{ |R'_i| \times IO + (hash + move_{key} + move_{oid}) \times r_i \right\} + \\
&\sum_{i=1}^{|QC|} \left[|S'_i| \times IO + \sum_{j=1}^{|QC|} \{ (hash + comp_{key} \times r_{ij}) \times s_{ij} \} \right] \\
&\left(|R'_i| = \left\lceil \frac{r_i}{\left\lfloor \frac{page}{oid+key} \right\rfloor} \right\rceil , \ |S'_i| = \left\lceil \frac{s_i}{\left\lfloor \frac{page}{oid+key} \right\rfloor} \right\rceil \right) \quad (6)
\end{aligned}
$$

もし，R内のOIDといくつかのキー値から構成される全ての要素のトータルサイズが割り当てられた問合せキャッシュのサイズよりも小さければ，ステージングバッファがそのまま利用され，それに続く処理が単純ハッシュ結合演算アルゴリズムにしたがって実行されることになる．その場合にはコスト式は次のようになる．

$$
\begin{aligned}
&Cost'_{Object-based\ Hash\ Join} = \\
&Cost_{load} + Cost_{join} + Cost_{generate} + \\
&(hash + move_{key} + move_{oid}) \times r + \\
&\sum_{i=1}^{|QC|} \{ (hash + comp_{key} \times r_i) \times s_i \} \qquad (7)
\end{aligned}
$$

一方，オブジェクトベースソートマージ結合演算方式では，最初にRとSのそれぞれの要素がソートされた連を含む一時ファイルが生成される．それらはソートされた一時集合R'とS'にマージされ，各一時集合に付与されたカーソルを利用して結合演算処理が実行される．コスト式は次のようになる．

$$
\begin{aligned}
&Cost_{Object-based\ Sort-Merge\ Join} = \\
&Cost_{load} + Cost_{join} + Cost_{generate} + \\
&\sum_{i=1}^{k} \{ r_i \times \log r_i \times (comp_{key} + swap_{key}) \} + \\
&\left\{ k \times \log k + (r-k) \times \frac{k}{2} \right\} \times (comp_{key} + swap_{key}) + \\
&\sum_{i=1}^{k} \left(|R'_i| \times IO \right) \times 2 + |R'| \times IO \times 2 + \\
&\sum_{i=1}^{l} \{ s_i \times \log s_i \times (comp_{key} + swap_{key}) \} + \\
&\left\{ l \times \log l + (s-l) \times \frac{l}{2} \right\} \times (comp_{key} + swap_{key}) + \\
&\sum_{i=1}^{l} \left(|S'_i| \times IO \right) \times 2 + |S'| \times IO \times 2 \\
&\left(k = \left\lceil \frac{(oid+key) \times r}{page \times |QC|} \right\rceil , \ l = \left\lceil \frac{(oid+key) \times s}{page \times |QC|} \right\rceil \right) (8)
\end{aligned}
$$

上の式では連のマージが1度しか起こらないと仮定したが，実際には問合せキャッシュの利用状況に依存して，連のマージは複数回起こることがある．または，全く起こらない場合もある．

4.2 性能評価結果

本稿で述べた結合演算方式の性能評価を行うために，ウィスコンシンベンチマーク[6]を利用して性能測定を行った．本ベンチマークテストはRDBMSにおけるSQL（主に問合せ処理）の性能評価のために設計されたものである．データ構造は拡張可能なリレーションを採用した．このリレーションには13個の整数型（4バイト）と3個の文字列型（52バイト）を含み，タプルサイズは208バイトになる．このリレーションをPERCIOのクラスにマップし，問合せはクラスのエクステント集合に対して発行する．PERCIO/C++で記述したクラス定義を図**5**に示す．実際には，Wisconsinクラスを基底クラスとする2つのクラスが存在し，それらのクラスのエクステント集合同士で結合演算を行う．

2つのクラスに含まれるインスタンス数の割合は10対1である．ベンチマークの問合せはインスタンス数の多い方のクラスに対する選択演算であり，インスタンス数の少ない方のクラスとの内部結合演算を含むものである．そして，結果集合内の全インスタンスに定義されている全メンバ変数がアクセスされる．問合せ結果のインスタンス数は，インスタンス数が多い方のクラスの10分の1になるように問合せ条件式を選んである．問合せが発行されてから結果集合の全てのインスタンスのメンバ変数がアクセスされるまでの間の経過時間を計測する．ベンチマークテストで利用したマシンはEWS4800/330（CPU: R4400）であり，メモリが48Mバイト，DBMSのキャッシュサイズは8Mバイトに設定した．

オブジェクトベースハッシュ結合演算方式の測定結果を図**6**～図**8**に示す．図を見て明らかのように，測定結果が問合せキャッシュサイズにかなり依存している．問合せキャッシュが小さい場合には，GRACEハッシュ結合演算方式で必要とされるディスクI/O回数が多くなってしまうため，それが性能に大きく影響している．問合

```
class Wisconsin : public Od_Pobject {
public:
//  Attribute Name       Range of Values      Order       Comment
    int unique1;         // 0-(MAXTUPLES-1)   random      unique, random order
    int unique2;         // 0-(MAXTUPLES-1)   sequential  unique, sequential
    int two;             // 0-1               random      (unique1 mod 2)
    int four;            // 0-3               random      (unique1 mod 4)
    int ten;             // 0-9               random      (unique1 mod 10)
    int twenty;          // 0-19              random      (unique1 mod 20)
    int onePercent;      // 0-99              random      (unique1 mod 100)
    int tenPercent;      // 0-9               random      (unique1 mod 10)
    int twentyPercent;   // 0-4               random      (unique1 mod 5)
    int fiftyPercent;    // 0-1               random      (unique1 mod 2)
    int unique3;         // 0-(MAXTUPLES-1)   random      unique1
    int evenOnePercent;  // 0,2,4,...,198     random      (onePercent * 2)
    int oddOnePercent;   // 1,3,5,...,199     random      (onePercent * 2) + 1
    char stringu1[52];   // -                 random      candidate key, unique
    char stringu2[52];   // -                 random      candidate key, unique
    char string4[52];    // -                 cyclic
};
```

図5 ウィスコンシンベンチマークのクラス定義
Fig. 5 Wisconsin class definition

せキャッシュを増やしていくと，それにつれて性能が改善する．**unique1** (integer, unique, random) のサイズは4バイトで**stringu1** (string, unique, random) のサイズは52バイトであるが，それらを結合キーとする測定結果は，いくつかの場合を除いてほとんど同じであった．これらの測定結果が同じでないところは，結合キーのサイズの違いにより単純ハッシュ結合演算方式が適用される範囲が異なっているためであり，コスト式 (6) と (7) で示した結合アルゴリズムの選択によって影響されている．

unique2 (integer, unique, sequential) を結合キーとする結合演算処理は**unique1**のものよりも性能がよくなっている．**unique2**を結合キーとする場合は，問合せ条件を満たすオブジェクトがエクステント集合の先頭部分に格納されている．そのため，結合演算処理に必要なインスタンスが格納されている物理的なページが固まっており，結合演算処理中や検索結果のインスタンスへのアクセスコストが削減されるのである．**onePercent**と**tenPercent**を結合キーとする結合演算処理は，問合せキャッシュが小さい時には他の属性を結合キーとするものより性能がよいが，問合せキャッシュが4Mバイトの時には悪くなってしまう．これは，要素を保持しているバケットがアンバランスで，バケット数を多くしたことがあまり効率的ではなかったためである．

次に，オブジェクトベースソートマージ結合演算方式の測定結果を図**9**～図**11**に示す．オブジェクトベースハッシュ結合演算方式の場合と異なり，問合せキャッシュのサイズに影響されず，これらの結果は非常に似通っている．もしも，結合演算処理におけるマージ処理の回数が同じであれば，コスト式 (8) で述べたように処理コストがほぼ同じになるのである．しかし，結合キー値の分散にはかなり依存することが分かる．例えば，**unique2**を結合キーとする結合演算処理は既にソート順にインスタンスが並んでおり，ソート処理のオーバヘッドが小さくなるため，他の結合キーに比べて測定結果が最もよくなっている．また，**unique1**を結合キーとする結合演算処理と**stringu1**を結合キーとするものは測定結果がほぼ同じであり，オブジェクトベースハッシュ結合演算方式の場合と同様に結合キー値のサイズには影響されていない．**onePercent**と**tenPercent**を結合キーとする結合演算処理の性能が他のものに比べて悪いのは，重複する結合キー値が多いために，マージ処理中の冗長な比較処理が増えてしまうからである．

図**12**は，元集合のインスタンス数の多い方の個数を300,000個に固定して，問合せキャッシュのサイズを変えることによって測定結果を比較したものである．問合せキャッシュのサイズが数Mバイトの時には，オブジェクトベースハッシュ結合演算方式がバランスのよい性能を持っていることは明らかである．上記したように，オブジェクトベースソートマージ結合演算方式の性能は結合キー値の分散にかなり依存する．また，問合せキャッシュのサイズが大きい場合でも，いくつかのケースではパフォーマンスが低下しているものがある．この1つの理由としては，オブジェクトキャッシュと問合せキャッシュのトータルサイズは常に同じ (8Mバイト) であり，問合せキャッシュのメモリページを追加するこ

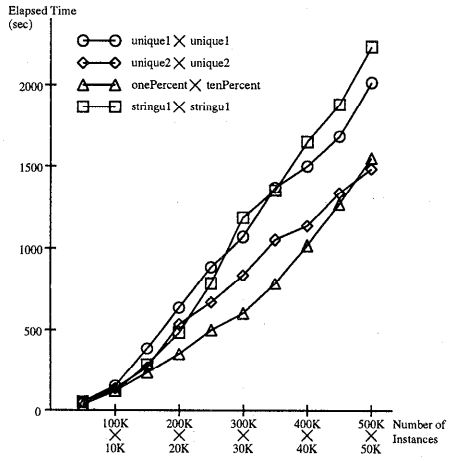


図 6 測定結果（ハッシュ）問合せキャッシュ = 256K バイト
Fig. 6 Performance result (hash) query cache = 256k bytes

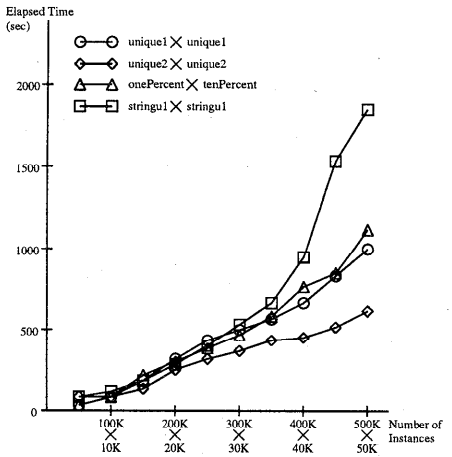


図 7 測定結果（ハッシュ）問合せキャッシュ = 1M バイト
Fig. 7 Performance result (hash) query cache = 1M bytes

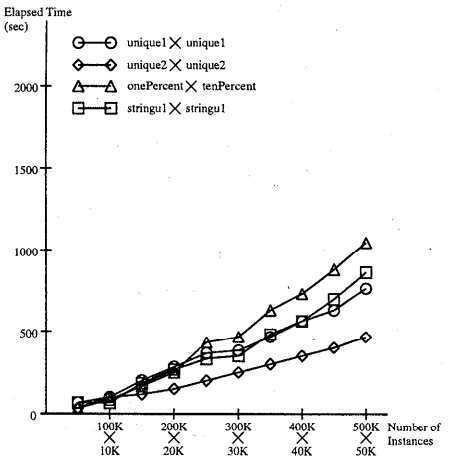


図 8 測定結果（ハッシュ）問合せキャッシュ = 4M バイト
Fig. 8 Performance result (hash) query cache = 4M bytes

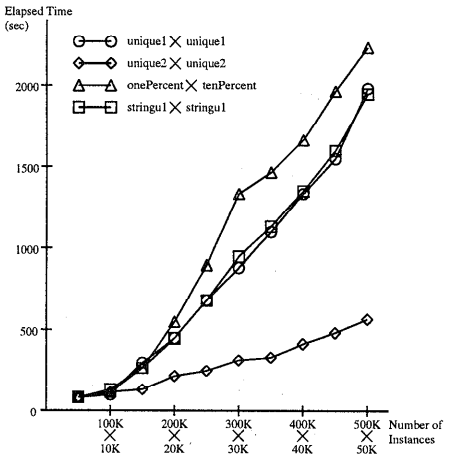


図 9 測定結果（ソートマージ）問合せキャッシュ = 256K バイト
Fig. 9 Performance result (sort-merge) query cache = 256K bytes

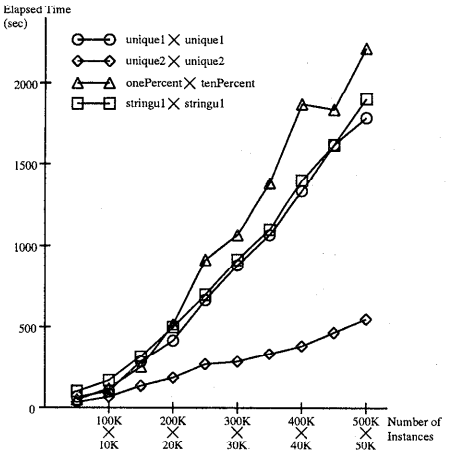


図 10 測定結果（ソートマージ）問合せキャッシュ = 1M バイト
Fig. 10 Performance result (sort-merge) query cache = 1M bytes

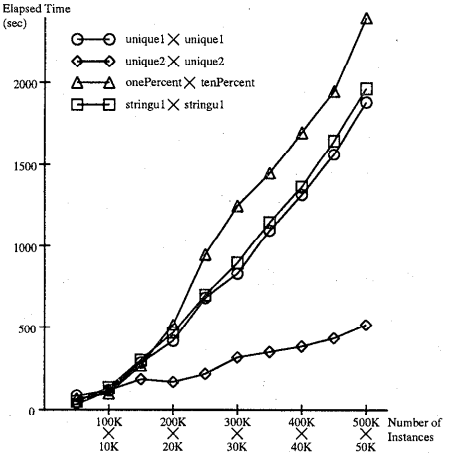


図 11 測定結果（ソートマージ）問合せキャッシュ = 4M バイト
Fig. 11 Performance result (sort-merge) query cache = 4M bytes

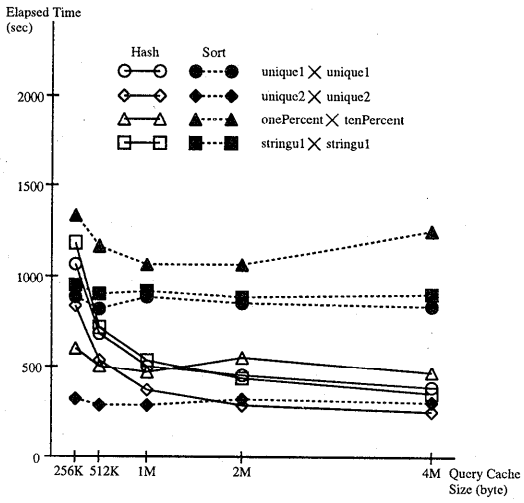


図12 問合せキャッシュサイズによる比較
Fig. 12 Comparison by query cache size

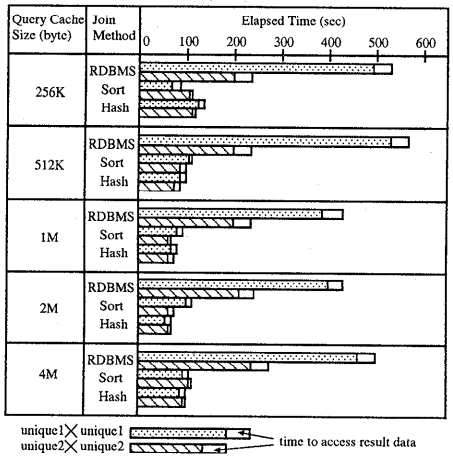


図13 PERCIOとRDBMSとの比較
Fig. 13 Comparison between PERCIO and RDBMS

とによりオブジェクトキャッシュが削減されてしまうからである．したがって，問合せ条件式の指定方法により，問合せキャッシュを適切なサイズに設定する必要がある．

4.3 RDBMSとの比較

本稿で述べたPERCIOの結合演算方式の有効性を確認するために，RDBMSのもの（ソートマージ結合演算方式を採用していると推測される）との比較も行った．図13は問合せキャッシュサイズ（RDBMSの場合はソート領域サイズ）の都合上，データ量が少ない（100K × 10K instances）場合の測定結果を示したものである．この図では，結果集合の全ての要素へのアクセス時間も含まれている．PERCIOの結合演算処理はRDBMSのものに比べて2～5倍性能がよいことが分かる☆．

OODBMSでの経過時間に対する結果データへのアクセスの割合は，RDBMSの場合よりも大きくなると考えられる．なぜなら，結果集合内のインスタンスのメンバ変数を取得するために，再びデータベースへのアクセスが必要だからである．しかし，アクセス時間の割合を示した結果はほぼ同じである．より多くのデータが用いられれば，この割合は異なってくるはずである．さらに，RDBMSでデータ分布がsequentialの結果はrandomのものに比べてほぼ2倍性能がよいが，PERCIOの場合にはそれほど違いが見られない．おそらく，PERCIOの場合にはハッシュやソートを行うためのメモリ領域が小さくて済むからであると思われる．つまり，結合キー値のデータ分布に影響されずに，安定した性能を維持できている．

5. 結　論

本稿では，OODBMSのための結合演算方式として，オブジェクトベースハッシュ結合演算方式とオブジェクトベースソートマージ結合演算方式を提案し，OODBMS PERCIOで実装されたこれら2つの結合演算方式の性能評価を行った．本結合演算方式は従来のGRACEハッシュ結合演算方式とソートマージ結合演算方式に基づいているが，OIDを有効に利用することとバケット分割を多数にすることで小さなメモリページでも実行できるように拡張されている．さらに，結合演算の前処理としてポインタ検索方式を利用することで，結合演算処理を行う際にオブジェクトポインタの遷移を取り除くことができ，問合せキャッシュの必要量を減らすことが可能になる（この部分のみPERCIOには未実装）．性能評価の結果，メモリ量が多い場合，オブジェクトベースハッシュ結合演算方式はオブジェクトベースソートマージ結合演算方式より性能がよい．また，RDBMSと比較するとPERCIOの方が2～5倍性能がよく，PERCIOの結合演算方式はメモリサイズや結合キー値のデータ分布に依存しないで性能が安定しているという特長がある．現在の実装はシンプルなものであり，問合せ条件に指定された2つの集合の結合演算しか処理できないため，今後はより複雑な結合演算にも対応していく必要がある．さらに，入れ子構造を持つ複合オブジェクトへの問合せ高速化に関して，問合せ言語仕様を含めた拡張も考えている．

☆ OODBMSとRDBMSではデータベースアクセス方法やシステムアーキテクチャが異なるため，単純にアルゴリズムの良否を比較することはできない．

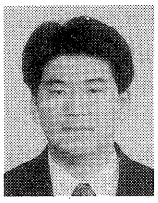
謝辞 NEC 第二コンピュータソフトウェア事業部および NEC 情報システムズ（株）の開発メンバには，PERCIO カーネルに関する有益なディスカッションをして頂き感謝致します．また，日頃から貴重な御助言を頂いたり，研究の相談をして頂いている NEC C&C メディア研究所の鶴岡邦敏研究部長に感謝致します．

参考文献

1) Banerjee, J., Kim, W. and Kim, K.C.: Queries in Object-Oriented Databases, *Proc. Data Engineering*, pp. 31–38 (1988).
2) Carey, M. J., DeWitt, D. J. and Naughton, J. F.: The OO7 Benchmark, *Proc. ACM SIGMOD*, pp. 12–21 (1993).
3) Cattell, R. G. G. and Skeen, J.: Object Operations Benchmark, *ACM Trans. Database Syst.*, Vol. 17, No. 1, pp. 1–31 (1992).
4) Cattell, R. G. G. and Barry, D. K.: *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann (1997).
5) DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R. and Wood, D.: Implementation Techniques for Main Memory Database Systems, *Proc. ACM SIGMOD*, pp. 1–8 (1984).
6) DeWitt, D. J.: The Wisconsin Benchmark: Past, Present, and Future, in *The Benchmark Handbook, Second Edition,* Gray, J. (ed.), Morgan Kaufmann, pp. 269–315 (1993).
7) Graefe, G.: Query Evaluation Techniques for Large Databases, *ACM Comput. Surv.*, Vol. 25, No. 2, pp. 73–170 (1993).
8) Kimura, Y. and Tsuruoka, K.: A View Class Mechanism for Object-Oriented Database Systems, *Proc. DASFAA*, pp. 269–273 (1991).
9) Kitsuregawa, M., Tanaka, H. and Moto-oka, T.: Application of Hash to Data Base Machine and Its Architecture, *New Generation Computing*, Vol. 1, No. 1, pp. 62–74 (1983).
10) Kitsuregawa, M., Nakayama, M. and Takagi, M.: The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method, *Proc. VLDB*, pp. 257–266 (1989).
11) Lieuwen, D. F., DeWitt, D. J. and Mehta, M.: Parallel Pointer-based Join Techniques for Object-Oriented Databases, *Proc. PDIS*, pp. 172–181 (1993).
12) Nakano, M. and Kitsuregawa, M: GN Hash Join Algorithm – A Robust Algorithm for Non-uniform Data Distributions –, *Proc. ADTI*, pp. 121–128 (1994).
13) Orenstein, J, Haradhvala, S., Margulies, B. and Sakahara, D.: Query Processing in the ObjectStore Database System, *Proc. ACM SIGMOD*, pp. 403–412 (1992).
14) Schneider, D. A. and DeWitt, D. J.: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *Proc. ACM SIGMOD*, pp. 110–121 (1989).
15) Shapiro, L. D.: Join Processing in Database Systems with Large Main Memories, *ACM Trans. Database Syst.*, Vol. 11, No. 3, pp. 239–264 (1986).
16) Shekita, E. J. and Carey, M. J.: A Performance Evaluation of Pointer-Based Joins, *Proc. ACM SIGMOD*, pp. 300–311 (1990).
17) 鶴岡邦敏, 木村裕, 波内みさ, 安村義孝: オブジェクト指向データベース管理システム PERCIO の開発と今後の課題, 電子情報通信学会論文誌, Vol. J79-D-I, No. 10, pp. 587–596 (1996).
18) Wirth, N.: *Algorithms + Data Structures = Programs*, Prentice-Hall (1976).

(平成10年 9 月20日受付)
(平成10年12月27日採録)

(担当編集委員 牧之内 顕文)



安村 義孝（正会員）

昭和 42 年生．平成 2 年慶應義塾大学理工学部電気工学科卒業．平成 4 年同大学院理工学研究科計算機科学専攻修士課程修了．同年日本電気（株）入社．現在 C&C メディア研究所主任．商用データベースの性能評価，オブジェクト指向データベース管理システム，WWW-DB 連携システムなどの研究開発に従事．情報処理学会，日本ソフトウェア科学会各会員．