DEWS2007 E2-3

CC-Optimizer: キャッシュを考慮した問合せ最適化器

辻　良繁†　川島　英之†

† 慶應義塾大学 理工学部 情報工学科
〒 223-8522 神奈川県横浜市港北区日吉 3–14–1
E-mail: †tsuji@ayu.ics.keio.ac.jp, ††kawasima@ics.keio.ac.jp

あらまし　CPU キャッシュとメモリのアクセスコストの乖離によりキャッシュミスが RDBMS の性能低下を起こすことが近年明らかになりつつある．命令キャッシュミスが生じる一原因に，RDBMS 内のオペレータ群の合計フットプリントが L1 命令キャッシュに収まらないことがある．これを改善すべく Zhou らはオペレータ実行順序を変更するバッファリングオペレータを提案した．同技法により RDBMS の性能は向上するが，Zhou らの成果は研究段階に留まっており，実際に RDBMS に適用するには不足点がある．それは最適化器にバッファリングオペレータ使用を決定させるアルゴリズムが示されてないことである．RDBMS におけるクエリ処理計画を最終的に決定するのは最適化器であるため，Zhou らの技法のみでは RDBMS の性能を現実的に向上させられない．そこで本論文では Zhou らの技法をも含むクエリ処理計画を選択可能な最適化器である，CC-Optimizer を実現した．CC-Optimizer の実現に際しては，最適化器がバッファリングオペレータを選択すべき状況を判断するアルゴリズムの新規提案と，同アルゴリズムの RDBMS への実装を行った．実験用 RDBMS には Zhou らと同様に PostgreSQL を用い，Linux Kernel 2.6.15，CPU Intel Pentium 4(2.40GHz) なる条件において実験を行った結果，CC-Optimizer を用いた RDBMS は既存 RDBMS に対する性能向上が示された．性能向上率は，OSDL DBT-3 における性能評価において，最大で 32.3%，総合で 6.08%であった．

キーワード　RDBMS，L1 命令キャッシュ，問合せ最適化器

CC-Optimizer: A Cache Conscious Query Optimizer

Yoshishige TSUJI† and Hideyuki KAWASHIMA†

† Department of Information and Computer Scicence, Faculty of Science and Technology, Keio University
Hiyoshi 3–14–1, Kohoku-ku, Yokohama, Kanagawa, 223–8522, Japan
E-mail: †tsuji@ayu.ics.keio.ac.jp, ††kawasima@ics.keio.ac.jp

Abstract　The difference between CPU access costs and memory access costs incurs performance degradations on RDBMS. One of the reason why instruction cache misses occur is the size of footprint on RDBMS operations does not fit into a L1 instruction cache. To solve this problem Zhou proposed the buffering operator which changes the order of operation executions. Although the buffering operator is effective, it cannot be applied for RDBMS in the real business. It is because Zhou does not show an algorithm for optimizer to select the buffering operator. Thus we realized the CC-Optimizer which includes an algorithm to appropriately select the buffering operator. Our contributions are the design of new algorithm on an optimizer and its implementation to RDBMS. For experimental RDBMS, we used PostgreSQL as Zhou did, and our machine environment was Linux Kernel 2.6.15，CPU Intel Pentium 4(2.40GHz). The result of preliminary experiments showed that CC-Optimizer was effective. The performance improvement measured by using OSDL DBT-3, was 32.3% in the greatest result, and 6.08% in all queries.

Key words　RDBMS，L1 Instruction Cache, Query Optimizer

1. はじめに

CPU とメモリの性能差の違いが RDBMS 性能劣化をもたらすことが明らかになりつつある．これまでの主たる研究動向は，L2 データキャッシュミス数を減らすアプローチであった [2], [5], [6].

それに対して Zhou らは L1 命令キャッシュミスを減らすことによる RDBMS 性能向上を提案した [7]．Zhou らの提案は，バッファリングオペレータと呼ばれる，オペレータをバッファリングするオペレータを導入することにより，従来は生じていた命令キャッシュミスを削減するものであった．

Zhou らの手法は，バッファリングオペレータが有利な条件では RDBMS の性能を向上させるが，それが不利な条件では逆に RDBMS の性能を低下させる．そして，有利・不利な条件が Zhou らには明らかにされていない．従って，Zhou らの研究結果だけでは，バッファリングオペレータを最適化器に使わせるには不十分である．最適化器がバッファリングオペレータを用いるには，それが有利・不利な条件を明らかにしていなければならない．

そこで本研究ではバッファリングオペレータを使用可能な最適化器を開発するために次の 4 ステップを踏む．(ステップ 1) まず，Zhou らの手法を再実装する．(ステップ 2) 次にバッファリングオペレータが有利・不利な条件を実験的に確認する．(ステップ 3) ステップ 2 で得たデータを元に，バッファリングオペレータを導入するアルゴリズムを決定する．(ステップ 4) 最後に，ステップ 3 で得たアルゴリズムを最適化器に導入する．

実験用 RDBMS には Zhou らと同様に PostgreSQL を用い，Linux Kernel 2.6.15，CPU Intel Pentium 4 (2.40GHz) なる条件において実験を行う．

本論文の貢献をまとめると，研究段階だったバッファリングオペレータを実用段階まで引き上げたこと，と言うことができる．

本論文の構成は次の通りである．2 節では従来研究について述べる．3 節では Zhou らの研究の問題点と，その問題を解決するアプローチを提案する．4 節では従来研究の再実装と実験的解析結果について述べる．5 節では問題を解決するシステム，CC-Optimizer について述べる．6 節では CC-Optimizer の評価について述べる．7 節では結論を述べる．

2. キャッシュを考慮した RDBMS の性能向上策

2.1 CPU キャッシュアーキテクチャと L2 データキャッシュに関する研究

CPU は階層的記憶構造を持つ．そして CPU のメモリアクセスはキャッシュメモリにより高速化される．キャッシュメモリは空間局所性と時間局所性の両方を考慮して CPU ダイ内部へ組み込んで配置される．例として図 1 に Pentium 4 のキャッシュメモリ構造を示す [7]．CPU はまずデータ/命令を求めて L1 キャッシュにアクセスする．それに失敗すれば L2 キャッシュにアクセスし，さらに失敗すればメモリにアクセスを行なう．キャッシュアクセスコストとメモリアクセスコストの差は 10 倍以上あるため [7]，キャッシュミスを削減するという研究テーマはここ 10 年ほど扱われてきた [5]．

削減効果の大きいキャッシュミスは L2 キャッシュミスである．そのため L2 キャッシュミスに関して幅広い研究が行なわれてきた．その中には，新しいキャッシュを考慮したアルゴリズム [2], [5], [6]，キャッシュを考慮した索引構造 [4], [8]，キャッシュを考慮したデータ構造 [1] などがある．

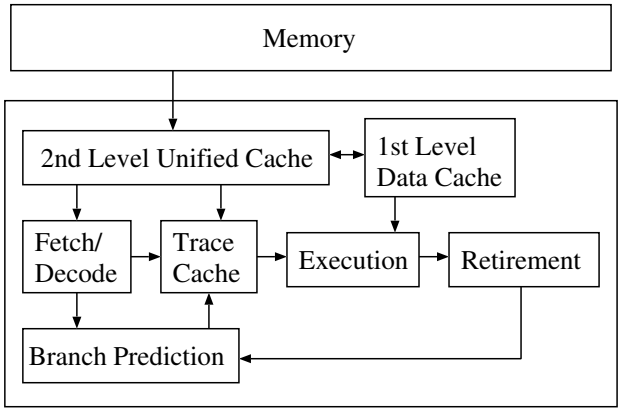

図 1　Pentium 4 キャッシュメモリ構造

```
select count(*) from item
```

図 2　単純な集約演算を含むクエリの例

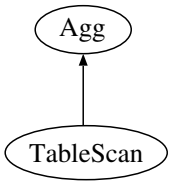

図 3　単純なプラン木

2.2 L1 命令キャッシュに関する研究

一方，L1 命令キャッシュを考慮した研究はそれほど行なわれて来なかった．そのなかで，現実の RDBMS を用いた実践的な研究を行なったものに，PostgreSQL を用いた Zhou らのバッファリングオペレータ研究 [7] がある．我々の研究は彼等の研究を踏まえて行なうために，ここでは彼等の研究を詳述する．

RDBMS がクエリ処理に必要とするオペレータの総サイズは，L1 命令キャッシュより大きい状況があり，この時に L1 命令キャッシュミスが発生し，RDBMS の性能が下げられてしまう．このような状況がどうして発生するのかを述べる．

そもそもオペレータとは RDBMS 内で，ある意味を持って行なわれる処理のことである．図 2 に示されるクエリを処理する場合を考える．

このクエリを処理するためには，item テーブルをスキャンするオペレータと，スキャンされた結果を集約するオペレータが必要になる．これらは PostgreSQL ではそれぞれ TableScan オペレータ，Agg オペレータとして実装されている．そしてこれらのオペレータは図 3 に示されるようなプラン木を構成する．

さて，このプラン木において Agg は親 (P)，TableScan は子 (C) と呼ばれる．そして P は C からタプルを一つずつ取り込んでは処理する．ここに問題が発生しうる．もしも図 2 の item テーブルの濃度が 8 であったならば，オペレータは次のように実行される．

PCPCPCPCPCPCPCPC

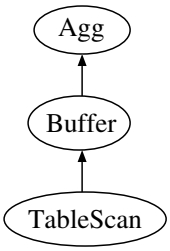

図 4　単純なプラン木へのバッファリングオペレータ

このとき，もしも P と C の合計フットプリントサイズが L1 命令キャッシュに収まらなければ，P も C もアクセスのたびに L1 命令キャッシュから追い出されるために RDBMS の性能劣化が引き起こされる．

この問題を解決する為に Zhou らはバッファリングオペレータを提案した．バッファリングオペレータは，小フットプリントで実装可能でオペレータをバッファリング可能なオペレータである．これにより C をバッファリングすることで，L1 命令キャッシュミスを減らすことができる．例えば図 3 にバッファリングオペレータを導入すると図 4 のようなプラン木が構成される．このとき，バッファリングオペレータのバッファリングサイズが 8 ならば，オペレータ実行順序を次のように変更可能である．

PCCCCCCCCPPPPPPP

この場合には C と P が連続して呼ばれるために，P と C の合計が L1 命令キャッシュサイズを上回る場合には，L1 命令キャッシュミスを削減できる．

文献 [7] によれば，PostgreSQL を用いた実験においてバッファリングオペレータは命令キャッシュミス回数を 80%程度削減し，クエリ性能を 15%ほど改善したとある．そのためバッファリングオペレータは有用な技術であると考えられる．

3. 課題と研究方法

3.1 課題：バッファリングオペレータの問題点

前節でバッファリングオペレータが有用な技術であることを述べた．しかし，同技術を実用的な観点から見た時には，問題がある．その問題とは，同技法がクエリ処理の流れに組み込まれていないことである．バッファリングオペレータは条件によっては性能向上に働くが，条件によっては性能劣化に働いてしまう．そのため，実用的な観点からは，バッファリングオペレータ使用を判断する機構が必要になる．

クエリ処理の最終段階ではプランナが提出した幾つかのパス木を最適化器が選択するため，バッファリングオペレータ使用の判断は最適化器に委ねられるのが自然な設計であると我々は考える．

そこで我々は本研究の課題を次のように設定する．

バッファリングオペレータ使用判断を含む最適化器：
CC-Optimizer の開発

3.2 研究方法

上記課題を解決するために，我々は次のような研究方法を採用する．

（1） Zhou らの技術の再実装と実験的解析

バッファリングオペレータ自身を PostgreSQL 上に再実装し，同オペレータが有利・不利な条件を実験的に解析する．

（2） 最適化器アルゴリズムの設計および実装

上記解析結果より，有利な条件時にはバッファリングオペレータを使い，不利な条件時にはバッファリングオペレータを使わないという，最適化器のアルゴリズムを決定する．そして同アルゴリズムを PostgreSQL の最適化器に実装する．

4. 再実装と実験的解析

4.1 再　実　装

命令キャッシュの追い出しを防ぐため，Zhou らの方法を参考にバッファリングオペレータの再実装を行った．実装は PostgreSQL-7.3.16 上で行った．

バッファリングオペレータは既存のオペレータへ変更を加えないよう，独立した新しいオペレータを追加する形で実装した．したがって起動は，5 節に示す CC-Optimizer によりプラン木の適切な位置にバッファリングを指示するノードを挿入し，エグゼキュータにより実行される形式を採った．このため，エグゼキュータに対して，バッファリングオペレータと既存のオペレータを同様に扱えるよう最小の変更を行った．

インターフェイスは PostgreSQL に既存のオペレータと同様，open-next-close インターフェイスを採用した．open 関数，および close 関数は子オペレータの生成したタプルへのポインタ，およびその状態を保持する配列の確保・解放を行う．ただし，Zhou らの実装とは異なり，確保される配列のサイズは固定でなく，CC-Optimizer により指定されるよう変更した．

next 関数は以下の 2 つの動作から成る．

（1） 配列が空の場合

配列が満たされるか，終端タプルが返されるまで子オペレータを続けて実行し，子オペレータの生成したタプルへのポインタを配列へ格納する．

（2） 配列に消費されていないタプルが存在する場合

配列に格納されているポインタを親オペレータへ返却する

バッファリングオペレータの目的は親オペレータ，子オペレータ間の命令キャッシュの追い出しを防ぐことである．バッファリングオペレータを挿入することで，親オペレータは子オペレータの next 関数の代わり，バッファリングオペレータの next 関数を実行する．バッファリングオペレータの next 関数が (1) の状態の際，子オペレータは続けて実行されるため，命令キャッシュミスを回避することができる．同様に (2) の状態の際，親オペレータが続けて実行される．

また，親オペレータ，および子オペレータとバッファリングオペレータ間において，命令キャッシュの追い出しが発生してはならす，next 関数は最小のステップで 2 つの動作を記述されるよう留意されなければならない．我々は Intel VTune により，コンパイル後の next 関数の命令サイズが十分小さいこと

```
select count(*),avg(point),avg(id),avg(id/2) from points
where id < 5000000 and point <= 0
```

図 5 集約演算

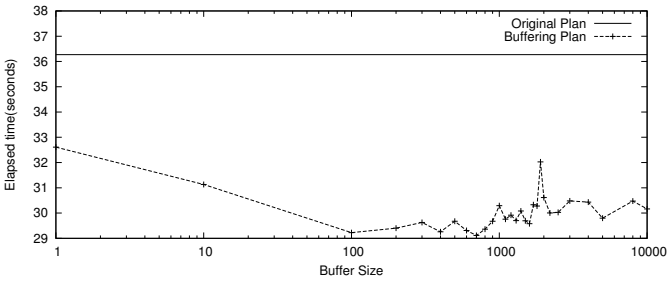

図 6 集約演算におけるバッファサイズを変化させた場合の実行時間の比較

```
select count(*), avg(p.point),avg(p.id),avg(n.id/2) from
points p, names n where p.id < 5000000 and p.id = n.id
and p.point >= 0
```

図 7 結合演算

を確認した．

4.2 実験的解析

本節において，バッファリングオペレータが有利・あるいは不利に働く条件を観察する．

4.2.1 バッファサイズの影響

バッファリングオペレータはバッファサイズを増やすほど，L1 命令キャッシュへ命令をロードする回数を減少させ，L1 命令キャッシュミス回数減少からクエリ実効時間の改善が期待できる．一方，バッファリングオペレータによって保持されるポインタの数が多くなれば，L2 データキャッシュミスの回数が増加する．従って，次の二つの実験を通して，バッファリングオペレータが持つべき，最適なバッファサイズを求めた．第一にバッファリングオペレータを一つだけ挿入する場合について測定する．実験には図 5 の SQL 文を使用した． 結果を図 6 に示す．図 5 に示されるクエリについては，バッファサイズ 1000 から 2000 の間において実行時間の改善が見られた．特にバッファサイズ 1000 において実行時間はオリジナルに対して 3.6%改善した．

第二にバッファリングオペレータを複数挿入する場合について測定を行なった． 結果を図 8 に示す．図 7 に示されるクエリにおいては，バッファサイズ 700 において最大 19.7%の改善が見られた．以上の二つの実験から，我々の実験条件においては，挿入された単一・または複数のバッファリングオペレータが持つべきバッファサイズの合計は 2000 程度が適当であると考えられる．

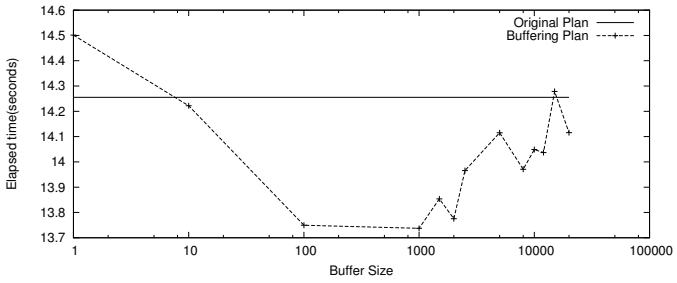

図 8 結合演算におけるバッファサイズを変化させた場合の実行時間比較

```
select count(*), avg(point), avg(id), avg(id/2) from points
where id < (子オペレータの実行回数) and point >= 0
```

図 9 濃度効果

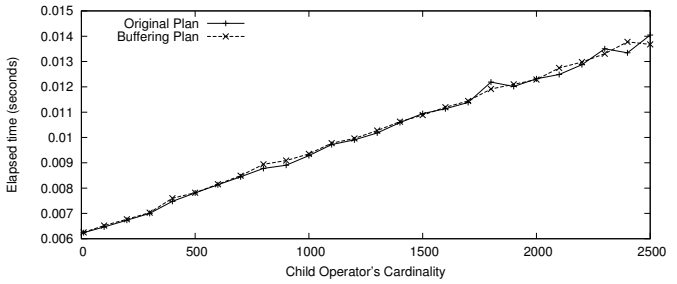

図 10 濃度を変化させた場合の実行時間の比較

4.2.2 濃度(注1)の影響

バッファリングオペレータは親オペレータと子オペレータの命令サイズの合計が L1 命令キャッシュよりも大きいとき，実行順序を入れ替えることで，L1 命令キャッシュを減らしクエリ処理全体の実行時間を減少させる．一方，バッファリングオペレータの実装は従来の RDBMS に対し，その初期化，終了処理などの余計な処理が加わる．したがって，子オペレータの実行回数の変化に応じて，この負荷が償却される閾値を以下の実験によって求めた．また，子オペレータの実行回数が前節で求めたバッファサイズ 2000 より小さいとき，バッファリングオペレータの負荷がより小さくなるよう初期化時に確保される配列のサイズを可変とした．このサイズは 5 節で述べる CC-Optimizer により指定される。実験には図 9 の SQL を使用した． 実験結果を図 10 に示す．実験結果は全体を通して，オリジナルプランとバッファリングプランの実行時間はほぼ同じであった．そこで本論文では，初めてバッファリングプランがオリジナルプランより高速となった 1500 を閾値として採用する．また，バッファリングオペレータが初期化時に確保する配列のサイズを可変とした結果，子オペレータの実行回数が非常に少ない時でもバッファリングプランは大きな遅延を起こさず，その負荷を非常に小さく出来たと言える。

(注1)：タプル数

表 1　PostgreSQL 命令サイズ

演算		命令サイズ
TableScan	without predicates	9KB
	with predicates	13KB
IndexScan		14KB
Sort		14KB
NestLoop Join	(inner) TableScan	11KB
	(inner) IndexScan	11KB
Merge Join		12KB
Hash Join	build	10KB
	probe	12KB
Aggregation	base	10KB
	COUNT	1KB 未満
	MAX	1.6KB
	MIN	1.6KB
	SUM	2.7KB
	AVG	6.3KB

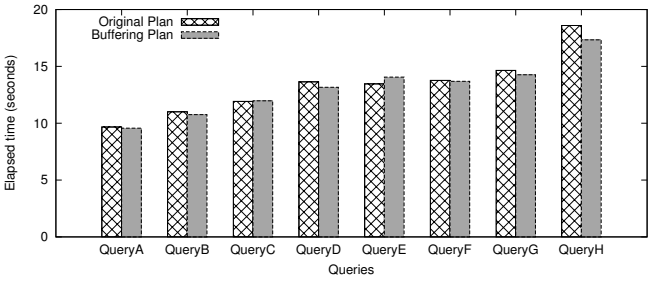

図 11　集約演算の変化による実行時間の比較

4.2.3　命令サイズ

表 1 は，Zhou らの研究によって示された PostgreSQL の主要モジュールの命令サイズである．バッファリングオペレータは順序的に実行される 2 つのオペレータの命令サイズが L1 命令サイズを超える場合に実行時間を改善するため，表 1 からバッファリングオペレータを挟み込むことで実行時間の改善を期待できるオペレータを推定することができる．ただし，コンパイル後の命令サイズは計算機環境によって異なり，また異なるモジュール間で同じ関数を共有する可能性があるため，これらの値を単純に足し合わせることはできない．

集約演算は大きな base 命令と個々の集約関数の命令から成る．従って，集約演算はクエリの複雑さにより命令サイズが変化する．実験は図 2 のクエリを用いて集約演算の複雑さを変えた場合の実行時間の変化を測定した．

実験結果を図 11 に示す．クエリ F，クエリ G に見られるように，比較的命令サイズの大きい avg 関数，sum 関数を含む 3 種類以上の集約関数が使用される時，バッファリングプランは実行時間の改善を見せた．一方，avg，sum 関数の 2 種類の集約関数の組み合わせを使用したクエリ D，クエリ E は，改善が見られる場合と見られない場合に分かれた．また，avg 関数のみを使用したクエリ H は，5 つ以上 avg 関数を使用する場合に改善を見せた．この現象は sum 関数でも同様に観察された．本研究では確実に実行時間の改善を見せる avg，sum 関数を含む 3 種以上の集約演算の場合にバッファリングオペレータは有効であると判断した．

```
select * from points where id < 5000000 and point >= 0
order by point desc
```

図 12　ソート演算

表 3　ソート演算における実行時間の比較

	オリジナルプラン	バッファプラン
実行時間 (秒)	91.6307795	90.9344472

```
select * from names n, (select count(*) from points group by
point) as t where t.count = n.id
```

図 13　サブクエリ演算

表 4　サブクエリ演算における実行時間の比較

	オリジナルプラン	バッファプラン
実行時間 (秒)	184.6216314	186.9337056

次に，ソート演算について，バッファリングオペレータの有効を確認した．実験には図 12 の SQL 文を使用した． ソート演算を親オペレータとする場合，改善度は小さいがバッファリングオペレータは有効であった．ただし，逆にソート演算を子オペレータとする場合，ソート演算はソート済みのタプルを返却するのみでその命令サイズは小さく，バッファリングオペレータは有効ではないと考えられる．

最後に，命令サイズの小さなモジュールにバッファリングオペレータを適用した場合の実行時間を比較した．実験は図 13 の SQL 文を使用し，バッファリングプランはサブクエリ演算のみを対象にバッファリングオペレータを付加した． 実行時間の比較を表 4 に示す．命令サイズの小さなモジュールに対して，バッファリングオペレータを適用した場合，性能劣化が見られた．

5.　CC-Optimizer：キャッシュを考慮した問合せ最適化器

前節より，バッファリングオペレータは優れた性能向上手法であるが，性能劣化を招く場合が存在するため，有利時のみ適用すべきである．

5.1　バッファリングオペレータ挿入アルゴリズムの提案

前節で得られた結果より，本実験環境では，図 14 の条件におけるバッファリングオペレータの挿入が適切であると言える．図 14 の条件を満たすバッファリングオペレータの挿入箇所が複数存在する場合，CC-Optimizer は，L2 データキャッシュへの収容可能性を考慮し，バッファサイズの合計が 2000 になるよう調整する．現在はバッファサイズの合計をバッファリングオペレータの挿入する箇所の数で等分割するよう実装する．

5.2　最適化器の実装

PostgreSQL 内部において、プラン木はオペレータと対応す

番号	内容
クエリ A	select count(*) from points where id < 5000000 and point >= 0
クエリ B	select count(*), avg(point) from points where id < 5000000 and point >= 0
クエリ C	select count(*), avg(point), sum(id) from points where id < 5000000 and point >= 0
クエリ D	select count(*), avg(point), sum(id), sum(id/2) from points where id < 5000000 and point >= 0
クエリ E	select count(*), avg(point), avg(id), sum(id/2) from points where id < 5000000 and point >= 0
クエリ F	select count(*), avg(point), max(point/2), sum(id/2) from points where id < 5000000 and point >= 0
クエリ G	select count(*), avg(point), avg(id), sum(id/2), max(point/2) from points where id < 5000000 and point >= 0
クエリ H	select count(*), avg(point), avg(id), avg(id/2), avg(point/2), avg(id/3) from points where id < 5000000 and point >= 0

表 2　集約演算の複雑さを変化させたクエリ

（1） 親ノードが 3 種類以上の集約関数を含み，子ノードがソート演算以外の主要モジュールである場合
（2） 結合演算を含む場合で子ノードがソート演算ではない主要モジュールである場合
（3） ソート演算であり，子ノードが主要モジュールである場合
（4） 上記 3 条件のいずれかを満たし，かつ，想定される処理タプル数が 1500 以上である場合

図 14　バッファリングオペレータ挿入アルゴリズム

るノードのリスト構造で実装されている．CC-Optimizer の呼び出しは既存のオプティマイザの最後に配置し，引数として既存のオプティマイザにより生成されたプラン木のルートノードが渡される．CC-Optimizer は適切な箇所にバッファオペレータと対応するバッファノードを追加し，引数として受け取った親ノードを返値として返却する．

CC-Optimizer の核となる部分の擬似コードを図 15 に示す．

CC-Optimizer は第 1 に，引数として既存のオプティマイザにより生成されたプラン木のルートノードを受け取り，CCO_recursive 関数により再帰的にプラン木を辿る．つまり，CCO_recursive 関数は，引数として受け取ったノードに子ノードが存在する場合，子ノードに対して先に CCO_recursive 関数を適用し，葉に近いノードからバッファリングオペレータの挿入判断を行う．

第 2 に，引数として受け取ったノードを親ノードとして，子ノードとの間で，前節アルゴリズムによりバッファオペレータが有効と判断される場合にリスト操作によって、プラン木へバッファノードを追加する．4.2.2 節に示した結果から，まず子ノードから 1500 以上のタプルが返されることが想定されることを確認する．この判断には PostgreSQL が事前に収集した統計情報を利用する．

次にプランの種類に応じて分岐処理を行う．

（1） 親ノードが集約演算の場合，avg, sum 関数と count 関数以外の 3 種類以上の集約関数を含む場合
（2） 結合演算を含む場合
（3） ソート演算を含む場合

上記を満たす場合，switch 文を早期脱出せず，10 行目で次の判断を行う．子ノードが 4.2.3 節で示したソート演算を除く，主

```
CCO_recursive(親プラン)
1  if (子プランがある) {
2    子プラン = CCO_recursive(子プラン)
3    if (子プランの想定実行回数 >= BO 有効の最小実行回数){
4      switch (親プランの種類) {
5       case 集約演算:
6        if (!(avg,sum 関数と count 関数以外の集約演算を含む))
7          break;
8       case 結合演算:
9       case ソート演算:
10         if (子プランがソート以外の主要モジュール)
11           親プランと子プランの間に BO を挿入
12     } /* switch */
13   }  /* if */
14 } /* if */
15 return 親プラン

(注) BO: バッファリングオペレータ
```

図 15　CC-Optimizer の擬似コード

要モジュールである場合，CCO_recursive 関数はリスト操作により，親と子ノードの間にバッファノードを挿入する．最後に返値として，親ノードを返却する．

このような実装により CC-Optimizer は既存のオプティマイザを変更を必要としない．また、CC-Optimizer の処理は全体のクエリ処理の時間に比べ、十分に小さいことを確認した．

6. 評　　価

6.1 実験環境

実験環境には表 5 に示すハードウェアを使用した．ハードウェアは Zhou らの論文と類似した環境で評価を行うため，若干旧式のものを選択した．

6.2 実験内容

6.2.1 OSDL DBT-3

我々は，CC-Optimizer を用いた PostgreSQL の性能評価に OSDL DBT-3 [3] を用いた．OSDL DBT-3 は TPC-H を参考に作成されたデータベースベンチマークソフトウェアであり，TPC-H の簡易版性能評価ツールとしてオープンソースライセンスで提供されたものである．TPC-H テスト，および DBT-3

表 5　実 験 環 境

CPU	Pentium(R) 4　2.40GHz
OS	Fedora Core 5 (Linux 2.6.15)
Main-memory	1GB
Trace cache	12K uops
L1 D cache	8K
L2 cache	512K
C Compiler	GNU's gcc 4.1.0

に用意されたクエリは，大規模なデータにおいて，意思決定支援システムのパフォーマンスを測るもので複雑な 22 個のクエリが用意されている．特に特徴的な結果が示される 2 つの問合せのみ，参考として下記に示す．

- Query 20 (最良の結果が示される問合せ)

SELECT s_name, s_address **FROM** supplier, nation **WHERE** s_suppkey **IN** (**SELECT** DISTINCT (ps_suppkey) **FROM** partsupp, part **WHERE** ps_partkey=p_partkey **AND** p_name LIKE ':1%' **AND** ps_availqty > (**SELECT** 0.5 * SUM(l_quantity) **FROM** lineitem **WHERE** l_partkey = ps_partkey **AND** l_suppkey = ps_suppkey **AND** l_shipdate >= ':2' **AND** l_shipdate < date ':2' + interval '1 year')) **AND** s_nationkey = n_nationkey **AND** n_name = ':3' **ORDER BY** s_name;

- Query 12 (最悪の結果が示される問合せ)

SELECT l_shipmode, SUM(**CASE WHEN** o_orderpriority = '1-URGENT' **OR** o_orderpriority = '2-HIGH' **THEN** 1 **ELSE** 0 **END**) **AS** high_line_count, SUM(**CASE WHEN** o_orderpriority <> '1-URGENT' **AND** o_orderpriority <> '2-HIGH' **THEN** 1 **ELSE** 0 **END**) **AS** low_line_count **FROM** orders, lineitem **WHERE** o_orderkey = l_orderkey **AND** l_shipmode **IN** (':1', ':2') **AND** l_commitdate < l_receiptdate **AND** l_shipdate < l_commitdate **AND** l_receiptdate >= date ':3' **AND** l_receiptdate < date ':3' + interval '1 year' **GROUP BY** l_shipmode **ORDER BY** l_shipmode;

6.2.2　実験パラメータ

本実験では，DBT-3 の実行時に指定すべきパラメータを次のように定めた．

- スケールファクタ

スケールファクタはデータの規模を指定するパラメータである．スケールファクタが 1 の時，生成される試験用データのテキストサイズは 1GB となり，DBMS へ投入した後のデータベースクラスタのサイズは 4GB 以上となる．今回の実験ではスケールファクタを 1 に設定した．

- ストリーム数

ストリーム数はスループット測定時に並行実行するトランザクション数を指定する．今回の実験ではストリーム数を 1 に設定した．即ち，コンテキストスイッチの頻発が発生しない状況で評価を行った．(注2)．

(注2)：PostgreSQL はクライアント接続時に fork システムコールによりプロセスを新たに生成するため，ストリーム数がすなわちプロセス数となる．スレッドが生成される種類の DBMS ではこの限りではない．

表 6　DBT-3 クエリによる実行時間の比較

	オリジナル (秒)	CC-Optimizer (秒)	改善度 (%)
Q20	216.1	146.3	32.30
Q5	155.2	129.7	16.43
Q4	117.6	99.88	15.07
Q2	17.23	14.65	14.97
Q17	20.95	17.92	14.46
Q22	4.435	3.831	13.62
Q21	1169	1020	12.75
Q8	224.5	196.3	12.56
Q10	119.7	104.9	12.36
Q7	158.2	139.6	11.76
Q3	75.71	70.37	7.053
Q18	139.0	135.4	2.590
Q15	72.64	71.20	1.982
Q14	37.72	37.43	0.769
Q19	41.04	40.74	0.731
Q13	49.41	49.11	0.607
Q6	33.93	33.79	0.413
Q1	249.0	250.0	-0.402
Q16	17.70	17.79	-0.508
Q9	1221	1258	-3.030
Q11	5.789	6.018	-3.956
Q12	45.42	92.03	-102.6

表 7　DBT-3 全クエリの実行時間比較

	オリジナル (秒)	CC-Optimizer (秒)	改善度 (%)
ALL	4191	3936	6.084

また，ここで指定しなかった実行時パラメータの値については，DBT-3 が自動選択したものを使用した．

6.3　実 験 結 果

上記の実験条件において DBT-3 を実行した結果を問合せごとに示したものを表 6 に示す．

表 6 から，22 個中 17 個の問合せについて，CC-Optimizer を用いた場合は性能向上が示されていることがわかる．特に Q20 については 32.3%と大幅な性能向上が示された．

一方，残り 5 つの問合せについては性能劣化が見られた．Q12 については 100%以上の性能劣化が観察された．5 つの問合せの内，特に 3 つのクエリにおいて，3%以上の性能劣化を見せた．この原因は，CC-Optimizer がバッファリングオペレータを有効に選択できなかったことだと推測される．しかしその確証を得るにはバッファリングオペレータを用いた場合に比べて，L1/L2 キャッシュミスが何回向上したのかを測定する必要があり，我々の知識の限りにおいては，L1 キャッシュミスを厳密に測定する方法を獲得できなかった．それゆえ，この精密な調査については今後の課題とする．

また，DBT-3 の全ての実行結果をまとめたものを，表 7 に示す．同表より，全体では 6.08%程度の性能改善が CC-Optimizer によりもたらされていることがわかる．

7. 結　　論

Zhou らにより提案されたバッファリングオペレータは，状況によって DBMS の性能を向上させたり劣化させたりする．だが，Zhou らはバッファリングオペレータを動的に選択するアルゴリズムを示しておらず，他の研究者によっても未だそれは提案されていない．このため，バッファリングオペレータは研究段階に留まっており，実用段階に達していなかった．

そこで本論文ではバッファリングオペレータを実用段階へ引き上げるべく，同オペレータを状況に応じて選択可能な最適化器を実現することを，研究課題に設定した．

研究課題を解決する為に我々がとった研究方法は次の通りだった．(1) バッファリングオペレータを再実装，(2) バッファリングオペレータを定量的に評価し，バッファリングオペレータが有利・不利な状況を調査，(3) ステップ 2 で得たデータを元に最適化器のバッファリングオペレータ選択アルゴリズムを設計，(4) 最後に設計したアルゴリズムを RDBMS に実装し，標準的ベンチマークで評価.

ステップ (1,2) で要した実験用 RDBMS には，Zhou らと同様に PostgreSQL を用い追試実験を行った．その結果，バッファリングオペレータを用いた PostgreSQL は既存の PostgreSQL に対して性能向上を示した．性能向上率は，集約演算を含む問合せについては最大 3.2%，結合演算を含む問合せについては最大 19.7% だった.

そしてステップ (3,4) では，ステップ 2 の追試実験で得たデータを元に，動的にバッファリングオペレータを選択可能な最適化器，CC-Optimizer を設計し，PostgreSQL-7.3.16 に実装した．22 種類の問合せを含む標準ベンチマークである DBT-3 を用いた実験の結果，最大で 32.3%，総合で 6.08%の性能改善が示された.

一方で，本論文で実現した CC-Optimizer はある 1 つのアーキテクチャという，極めて限定的な状況にしか適用できない．あらゆるアーキテクチャにおいてバッファリングオペレータを動的選択できるように CC-Optimizer を改善することが今後の課題である.

以上より，我々は本論文の貢献を，現段階においては 1 つのアーキテクチャにおいて，大幅な性能向上をもたらす DBMS 最適化器を実現したこと，と結論する.

文　　献

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. of VLDB Conference*, 2001.

[2] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proc. of VLDB Conference*, 1999.

[3] *OSDL DBT-3*. http://old.linux-foundation.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-3/.

[4] J. Rao and K. A. Ross. Making b+ trees cache conscious in main memory. In *Proc. of ACM SIGMOD Conference*, 2000.

[5] A. Shadtal, C. Kant, and J.F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. of VLDB Conference*, pp. 510–521, 1994.

[6] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proc. of VLDB Conference*, 2003.

[7] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. of ACM SIGMOD Conference*, pp. 191–202, 2004.

[8] 高見澤秀久, 有次正義. 配列を用いたキャッシュコンシャスな索引木の提案, 2002.