CC-Optimizer: キャッシュを考慮した問合せ最適化器

CC-Optimizer: A Cache Conscious Query Optimizer

辻 良繁 ♥ 川島 英之 ♦
Yoshishige TSUJI Hideyuki KAWASHIMA

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していないことから，Zhou らの技法のみでは現実的な使用は難しい．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．

One of the reasons why instruction cache misses occur is associated with the increasing footprint size of RDBMS operations. To solve this problem, Zhou proposed the buffering operator. By changing the order of operation executions, it reduces miss ratio. The method cannot be applied for RDBMS in the real business since it degrades performance in a certain situation. We realized the CC-Optimizer which provides an algorithm to select the buffering operator appropriately. Our contributions are the design of new algorithm on an optimizer and its implementation to RDBMS.

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．

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 [1]．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．

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．

本論文の貢献は，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である．

本論文の構成は次の通りである．2 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．3 節では問題を解決するシステム，CC-Optimizer について述べる．4 節では CC-Optimizer の評価について述べる．5 節では結論を述べる．

2. 再実装と実験的解析

2.1 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，3 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

♥ 学生会員 慶應義塾大学 tsuji@ayu.ics.keio.ac.jp
♦ 正会員 筑波大学 kawasima@cs.tsukuba.ac.jp

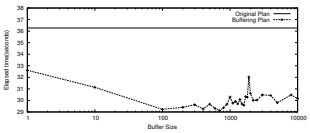

図 1 集約演算におけるバッファサイズを変化させた場合の実行時間の比較
Fig. 1 Execution Time of An Aggregation Query for Varied Buffer Sizes

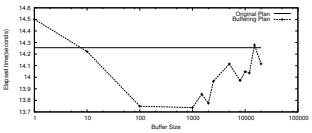

図 2 結合演算におけるバッファサイズを変化させた場合の実行時間比較
Fig. 2 Execution Time of A Two-table Join Query for Varied Buffer Sizes

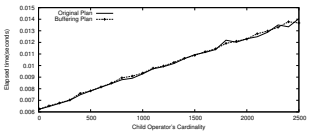

図 3 濃度を変化させた場合の実行時間の比較
Fig. 3 Cardinality Effects

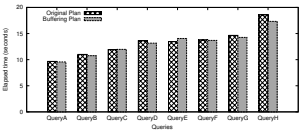

図 4 集約演算の変化による実行時間の比較
Fig. 4 Execution Time for Varied Aggregation Queries

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ず，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を確認した．

2.2 実験的解析

2.2.1 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 4 つの集約演算を含む問合せを用いた．

結果を図 1 に示す．図 1 より，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．実験には 2 つの結合演算を含む問合せを用いた．結果を図 2 に示す．特に，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た．

2.2.2 濃度の影響

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 3 節で述べる CC-Optimizer により指定される。実験では，2.2.1 節で用いた問合せにおける選択タプル数を制御した．

実験結果を図 3 に示す．実行時間は全体を通して，オリジナルプランとバッファリングプラン共にほぼ同じであった．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。

2.2.3 命令サイズ

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 命令キャッシュサイズを超える場合に実行時間を改善するため，Zhou らの研究により示される PostgreSQL の命令サイズから，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．実験は表 1 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 4 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

表 1 集約演算の複雑さを変化させたクエリ
Table 1 Varied Aggregation Queries

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

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．

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．ソート演算を親オペレータとする場合，0.76% と改善度は小さいが，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．

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．実験はサブクエリ演算を含むクエリを使用し，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．実験の結果，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，1.25% 程度の性能劣化が見られた．

3. 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．

3.1 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 5 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 5

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 5 バッファリングオペレータ挿入アルゴリズム
Fig. 5 Algorithm for Buffering Operator Insertion

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．

3.2 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 6 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．2.2.2 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．

次に分岐処理を行う．条件を満たす場合，switch 文を早期脱出せず，10 行目で次の判断を行う．子ノードがソート演算を除く，主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の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．

```
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 6　CC-Optimizer の擬似コード
Fig. 6　Pseudo-Code of CC-Optimizer

表 2　実験環境
Table 2　System Specifications

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

4.　評価

4.1　実験環境

実験環境には表 2 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．

4.2　実験内容

我々は，CC-Optimizer を用いた PostgreSQL の性能評価に OSDL DBT-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．

4.3　実験結果

実験では，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た．最大 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．TPC-H クエリの 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，全体では実行時間にして 6.08% の性能改善が CC-Optimizer によりもたらされた．

5.　結論

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，次のようにして実現した．(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) 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．

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

[2] 辻良繁, 川島英之. CC-Optimizer: キャッシュを考慮した問合せ最適化器. 電子情報通信学会 第 18 回データ工学ワークショップ, 2007.

辻 良繁 Yoshishige TSUJI
慶應義塾大学理工学部情報工学科在学中. 情報処理学会学生会員. 日本データベース学会学生会員.

川島 英之 Hideyuki KAWASHIMA
博士 (工学). 筑波大学大学院システム情報工学研究科，同大学計算科学研究センター，講師.