情報処理学会論文誌　データベース　Vol.7　No.2　104–116 (June 2014)

推薦論文

アウトオブオーダ型クエリ実行に基づく プラグイン可能なデータベースエンジン加速機構

早水 悠登[1,a)]　合田 和生[3]　喜連川 優[2,3]

受付日 2013年12月22日, 採録日 2014年4月4日

概要：アウトオブオーダ型クエリ実行とは，動的タスク分解と非同期入出力発行に基づくクエリ実行方式であり，従前の同期入出力発行・逐次処理に基づく実行方式と比べて，大規模データに対する選択的クエリ実行において高い性能を発揮することが知られている．本論文では，既存データベースエンジンにおけるクエリ実行の挙動を変えることなく，その処理性能をアウトオブオーダ型クエリ実行と同水準まで向上させるために，アウトオブオーダ型クエリ実行に基づくデータベースエンジン加速機構を提案する．当該機構は既存エンジンのクエリ実行と並行して当該クエリを協調的にアウトオブオーダ型実行し，バッファプールを介して先行的にデータベースページを供給することで，既存エンジンの入出力待ち時間を縮減し，大幅な高速化を実現する．本論文ではオープンソースデータベース管理システム PostgreSQL を対象とした加速機構の試作実装 PgBooster の構成法を示すとともに，ミッドレンジ級のサーバ・ディスクストレージからなる環境において評価実験を行い，その高速性を明らかにする．

キーワード：アウトオブオーダ型クエリ実行，OoODE，データベースエンジン加速機構，クエリ処理

Pluggable Database Engine Booster Based on Out-of-order Query Execution

Yuto Hayamizu[1,a)]　Kazuo Goda[3]　Masaru Kitsuregawa[2,3]

Received: December 22, 2013, Accepted: April 4, 2014

Abstract: Out-of-Order Query Execution is a query execution method which is based on dynamic task decomposition and asynchronous I/O operations. For selective queries on large scale data, it outperforms existing query execution methods. This paper proposes a pluggable database engine booster based on Out-of-Order Query Execution, which improves query execution performance of an existing engine to the same degree of Out-of-Order Query Execution with maintaining the compatibility of the engine. In parallel to a query execution in the engine, the booster runs Out-of-Order Execution of the query in a coordinated manner, and supplies database pages to the engine on ahead via the buffer pool. That reduces I/O waiting time of the engine and improves its performance significantly. This paper describes development of PgBooster, a prototype implementation of database engine booster, and its experimental evaluations on a mid-range class environment with a server and a disk storage system.

Keywords: Out-of-Order Query Execution, OoODE, database engine booster, query processing

1. はじめに

昨今の計算機システムにおいては演算・記憶資源の高密度化・大規模化傾向が著しい．プロセッサのマルチコア化が進み，商用プロセッサにおいて 8 コア程度はすでに一般

本論文の内容は 2013 年 11 月の WebDB フォーラム 2013 にて発表され，同シンポジウムプログラム委員会により情報処理学会論文誌データベースへの掲載が推薦された論文である．

[1] 東京大学大学情報理工学系研究科
Graduate School of Information Science and Technology, the University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[2] 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[3] 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo, Meguro, Tokyo 153–8904, Japan
[a)] haya@tkl.iis.u-tokyo.ac.jp

的である．また1,000コアを見据えた取り組みも始まっており [1]，計算機あたりのプロセッサコア密度が増加傾向にあることは明らかといえよう．他方，ストレージにおいては，ハイエンドクラスのストレージシステムでは1,000ディスクドライブを有するものも珍しくない．ストレージ仮想化技術の浸透や，昨今のビッグデータに対する機運の高まりなども後押しして，搭載ドライブ数・入出力帯域幅増加の動きはハイエンドシステムに限らず広まりを見せつつある．全世界で生み出されるデータ量は2年ごとに2倍になるとも予想されており [2]，計算機システムのデータ管理を担うデータベース技術において，このように高密度化・大規模化を続けるハードウェア環境を活用することが求められている．

著者らの研究グループでは**アウトオブオーダ型データベースエンジン（OoODE）** [3], [4], [5] なる高速データベースエンジンの開発を進めている．OoODEは**アウトオブオーダ型**と称するクエリ実行方式に基づき，クエリ実行を動的にタスク分解して非同期入出力を高多重で発行し，入出力完了を契機として並列タスク実行を駆動する．高多重入出力発行により入出力帯域を高効率に活用可能とし，並列タスク実行により複数コアの演算性能を活用可能とする点に特色があり，従前の**インオーダ型クエリ実行方式**，すなわち同期的入出力と逐次的演算実行に基づくクエリ実行方式とくらべて，特に中程度の選択性を有するアドホックな分析クエリ実行において高い性能を発揮することが知られている [4], [5]．

ここでは，アウトオブオーダ型クエリ実行方式をデータベースシステムにおいて実現し，その高速化を図る際のソフトウェアの大規模性・複雑性の問題に着目したい．データベースシステムは中心的機能のみでも100万行から1,000万行規模の大規模ソフトウェアであり，多様な周辺ソフトウェアとのインターオペラビリティが求められる．従前のデータベースシステムは，その根幹たるインオーダ型データベースエンジンの同期的・決定的な挙動を前提として構築されている．アウトオブオーダ型クエリ実行方式を実現するための1つのアプローチには，システム全体を再設計することがあげられ，アウトオブオーダ型実行方式の高速性を最大限に活用するという利点があるが，他方，その実現は必ずしも容易ではない [6], [7]．これに対して，本論文はより簡便なアプローチを模索したい．データベースエンジンにおけるクエリ実行方式をインオーダ型から変えることなく，アウトオブオーダ型クエリ実行の高速性を実現することができれば，現有するデータベースシステムの機能性を損なわずに高速性を享受することができる．工学的観点から価値あるアプローチといえよう．

本論文では，既存のデータベースエンジンにおけるクエリ実行方式をインオーダ型から変更することなく，その性能をアウトオブオーダ型データベースエンジンと同水準まで向上させるために，アウトオブオーダ型クエリ実行機能をプラグイン可能なデータベースエンジン加速機構として組み込む手法を提案する．当該加速機構は，既存のインオーダ型データベースエンジンがクエリを実行する傍らで，その進行にあわせて協調的に同じクエリをアウトオブオーダ型によって実行する．これによりバッファプールを介して先行的にデータベースページを供給し，インオーダ型実行器の入出力待ち時間を大きく縮減する作用をもたらし，とりわけ選択的なアドホック分析クエリ実行において大幅な高速化を実現する．提案する加速機構を用いた場合，等価なクエリを二重に実行することとなるが，昨今のデータベースシステムではプロセッサコアよりむしろ入出力が律速要因となることが多く，余剰プロセッサコアの活用によって入出力効率の向上を達成する方法と考えることができる．また当該機構によりアウトオブオーダ型クエリ実行方式の有効性を確認した後に，データベースエンジン自体にアウトオブオーダ型クエリ実行器を融合するなどの方法により，データベースエンジン自体の段階的なアウトオブオーダ型化の足がかりとすることもできる．本論文においては，著者らが文献 [8] で示した加速機構の構成法に加えて，バッファマネージャにおけるページ置換ポリシがクエリ処理性能に与える影響について議論する．

本提案手法の有効性を評価するために，著者らはオープンソースデータベース管理システムPostgreSQLを対象として，加速機構の試作実装PgBoosterを開発した．本論文ではその構成法を示すとともに，160ディスクドライブを有するストレージおよび24コアサーバからなるミッドレンジ級[*1]システムにおいて，業界標準ベンチマークTPC-Hを用いた評価実験を示し，その高速性を明らかにする．

本論文の構成は次のとおりである．2章では加速機構の概要を説明し，3章では加速機構の中核をなすアウトオブオーダ型クエリ実行の協調的制御アルゴリズムについて議論する．4章では加速機構を考慮したページ置換アルゴリズムについて議論する．5章では試作実装PgBoosterについて説明する．6章ではPgBoosterの評価実験について述べ，7章で関連する研究について言及した後，8章で本論文をまとめる．

2. データベースエンジン加速機構

一般的なデータベースエンジンは，大きく分けてクエリ実行器とバッファマネージャの2つのコンポーネントから構成される．クエリ実行器はクエリ実行プランに従ってデータベース演算を行う実行主体である．演算にデータベースページが必要な場合，バッファマネージャにページ

[*1] 必ずしも正確な定義があるわけではないものの，市場においてはサーバ・ストレージ製品群のうち高性能・高信頼性の高価格製品群であるハイエンド級と，低性能・低信頼性の低価格製品群であるローエンド級の中間層の製品群のことを指し，ミッドレンジ級との表現が用いられることが多い．

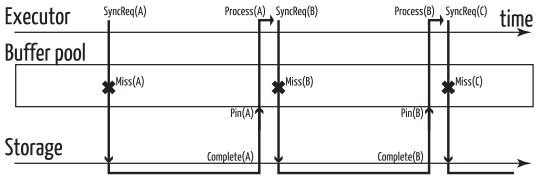


図 1　インオーダ型クエリ実行
Fig. 1　In-order query execution.

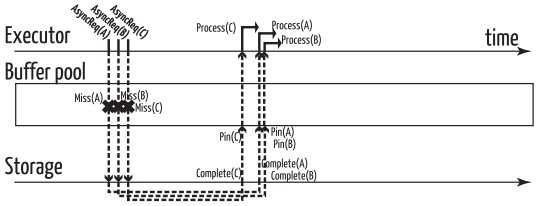


図 2　アウトオブオーダ型クエリ実行
Fig. 2　Out-of-Order query execution.

要求を発行して当該ページを取得する．バッファマネージャは，ストレージとの入出力，および主記憶上のページキャッシュ空間であるバッファプールの管理を担うコンポーネントである．実行器からページ要求をうけると，その要求がバッファヒットする，つまりバッファプール上に当該ページが存在する場合にはそれを返す．バッファミスする，つまり当該ページがバッファプール上に存在しない場合，ストレージに対して入出力要求を発行して当該ページデータを取得し，バッファプールに格納したうえで実行器へ返す．

クエリ実行器がインオーダ型クエリ実行方式を採用する場合の挙動模式を**図 1** に示す．インオーダ型クエリ実行器は，演算に必要なページ要求を同期的に発行し，その取得を待ってからページに対する演算処理を実行する．そのため，単一クエリ実行の論理的なアウトスタンディング入出力数[*2]はたかだか 1 であり，ストレージが高い入出力帯域を有していたとしてもそれを活用できず，クエリ実行時間において入出力待ち時間が支配的となる．仮に必要なデータを事前にバッファプールへ読み込んでおくことが可能であれば入出力待ちは回避できるが，アドホックなクエリ実行など要求されるデータが予測できない場合にはその限りではない．

一方，クエリ実行器がアウトオブオーダ型クエリ実行方式を採用する場合の挙動模式を**図 2** に示す．アウトオブオーダ型クエリ実行器は非同期入出力要求を多重発行し，ページのバッファプール読み込み完了を契機として演算実行を駆動する．これにより，入出力帯域を高効率に活用することが可能となり，特に中程度の選択性を有するクエリ実行において高い性能を示すことが知られている [5]．

ここで，アウトオブオーダ型クエリ実行器による高効率

[*2] アウトスタンディング入出力とは，ストレージデバイスに対して発行され，いまだ完了に至っていない仕掛中の入出力を意味する．

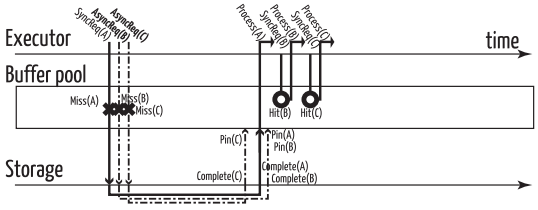


図 3　先行的なアウトオブオーダ型クエリ実行をともなうインオーダ型クエリ実行
Fig. 3　In-order query execution with leading Out-of-Order execution.

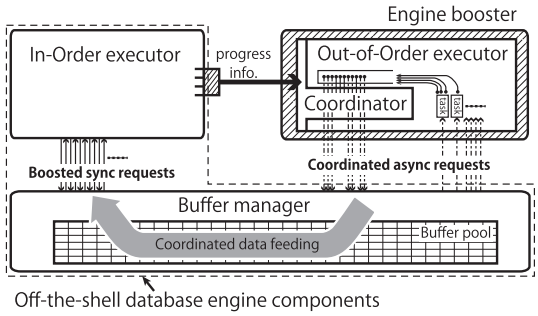


図 4　既存データベースエンジンと加速機構の概要
Fig. 4　Overview of off-the-shell database engine with Engine Booster.

な入出力帯域の活用という特性を，インオーダ型クエリ実行器から利することを考えてみたい．両実行器において同一のクエリを実行する場合，クエリ実行方式による実行順序の違いはあるものの，最終的には等価な処理を行うため，実行過程でそれぞれの実行器がアクセスするページの集合は一致する．つまり**図 3** に示すように同一クエリを両実行器で同時に実行すると，アウトオブオーダ型クエリ実行器が先行してバッファプールへと読み込んだページが，後続するインオーダ型クエリ実行器へと供給される形となり，インオーダ型クエリ実行器の入出力待ち時間を縮減することが期待できる．ただしアウトオブオーダ型クエリ実行器は，必ずしもインオーダ型クエリ実行器に都合の良い順序でページを読み込むとは限らない．有限バッファプール容量のもとでページ供給が有効に作用するためには，インオーダ型クエリ実行器の挙動を考慮してアウトオブオーダ型クエリ実行器を制御する仕組みが必要である．

本論文では，アウトオブオーダ型クエリ実行器とアウトオブオーダ型クエリ実行コーディネータからなる加速機構を構成する手法を提案することで，この仕組みが実現可能であることを示す．既存データベースエンジンに当該加速機構を組み込んだときのコンポーネント概要を**図 4** に示す．このデータベースエンジンでは，クエリ実行プランが与えられると，インオーダ型クエリ実行器と加速機構のアウトオブオーダ型クエリ実行器が同時に駆動される．加速機構のコーディネータは，プローブを介してインオーダ型クエリ実行器の進行状況を随時把握し，その入出力待ち時

間縮減に有効と見込まれる非同期入出力が優先的に発行されるよう，アウトオブオーダ型クエリ実行器を協調的に制御する．また全件走査などインオーダ型クエリ実行器によっても入出力帯域を十分活用することができ，アウトオブオーダ型クエリ実行器によるさらなる高速化が期待できない場合においては，加速機構の動作を無効化し，従来どおりインオーダ型クエリ実行器のみでクエリを実行する．この枠組みにおける既存データベースエンジンの加速効率は，コーディネータの協調制御アルゴリズムによって決定されるため，その設計が提案手法の有効性を左右する鍵となる．当該アルゴリズムについては，3 章において議論する．またページ供給はバッファマネージャを介して実現されるため，バッファマネージャにおけるページ置換ポリシが供給効率に影響を与える可能性が考えられる．4 章においてこの点について考察し，加速機構を考慮したページ置換ポリシの拡張について議論する．

3. アウトオブオーダ型クエリ実行の協調的制御アルゴリズム

3.1 クエリ実行過程の記述

アウトオブオーダ型クエリ実行器によるページ供給のタイミングが早過ぎると，インオーダ型クエリ実行器によって使用される前に当該ページは追い出されてしまう可能性が高い．つまり先行的にページ読み込みを行う順序が，ミクロなレベルではインオーダ型クエリ実行器のページ要求順序と異なっていてもよいが，マクロなレベルでは時間的に相関性を持つよう制御することが，アウトオブオーダ型クエリ実行コーディネータに求められる．

ここでクエリ実行過程において，どのような順序でページ要求が発行されるかについて考えたい．クエリ実行器が要求するページを決定するのは，**データベース演算** f [*3]と，その演算を実行する**コンテクスト** c である．たとえば B^+ 木索引から索引エントリを取得するという演算を考えると，検索キーや B^+ 木葉ページ内のスキャンポインタなど，演算の適用対象データがコンテクストである．この演算 f とコンテクスト c をあわせて**演算インスタンス** $i = \langle f, c \rangle$ と呼ぶことにする．演算インスタンスが決定されれば，その実行において要求されるページもまた決定されるので，ページ要求の発行順序は，演算インスタンスの実行順序によって把握することができる．

1 つのクエリ実行は，1 つの初期演算インスタンスの実行をもって開始される．演算インスタンス i が実行器により実行されると，新たに実行すべき複数の演算インスタンスを生じる場合があり，これらの次々と生み出される演算インスタンスの実行によりクエリ実行が展開される．たとえば**図 5** に示す 2 表の索引検索とネステッドループ結合からなるプランを考える．初期演算インスタンスとして，R 表の該当索引エントリを取得する演算 f_1 の演算インスタンス i_1 が実行されると，この例では 4 件のエントリが該当し，各エントリの指すタプルを取得する演算 f_2 の演算インスタンス i_2，i_8，i_x，i_y が生じる．演算インスタンスの生成関係をエッジとし，演算インスタンスをノードとすると，図に示すように演算インスタンスが木構造をなすものと考えることができる[*4]．この木構造をクエリの**演算インスタンス木**と呼ぶ．

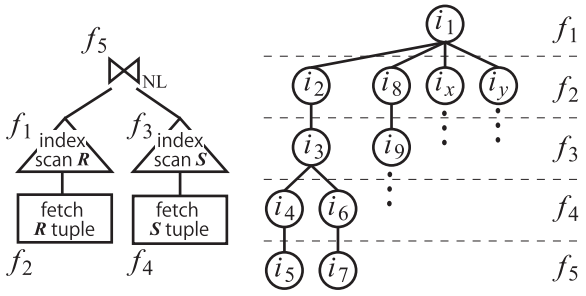


図 5　2 表の結合を行うクエリ実行プランと演算インスタンス木
Fig. 5　A query execution plan of joining two tables and its execution instance tree.

クエリ実行における演算インスタンスの実行順序は，演算インスタンス木の展開順序によって考えることができる．インオーダ型クエリ実行器は，一般に演算実行制御用にスタックを持ち，スタックから演算インスタンスを 1 つ取り出して実行し，結果として生じる子演算インスタンスをスタックに積む，という動作を繰り返すことでクエリを実行する．1 つの演算インスタンスの実行結果として複数の子演算インスタンスが生じた場合には，インオーダ型クエリ実行器は所定の順序に従いこれらをスタックに積む．この所定の順序は，インオーダ型実行器の具体的な実装方式やデータ格納方式から判別可能であり，データベースの状態・クエリが同一であれば同一の順序となるものとする．たとえば図 5 においては，クエリ開始時に i_1 がスタックに積まれるため，インオーダ型クエリ実行器はまず i_1 を実行し，結果として生じる演算インスタンスをスタック頂上から順に i_2，i_8，i_x，i_y と並ぶように積む．このとき，この 4 つの演算インスタンスは先に実行されるものから順に i_2，i_8，i_x，i_y であるので，i_2 は i_1 の 1 番目の子，i_8 は 2 番目の子，というように，親を同じくする演算インスタンス間に順番が与えられる．その後，インオーダ型クエリ実行器が i_2 を実行すると，新たな演算インスタンス i_3 が生じてスタックに積まれるため，i_3 が i_8，i_x，i_y に先立って実行される．以下同様にして演算インスタンスが次々と実

*3　ここでいうデータベース演算はあくまで概念的な処理単位であり，実際のデータベースエンジンにおけるクエリプラン表現などと 1 対 1 対応している必要はない．

*4　演算の種類によっては，複数の親が合流して演算インスタンスを生成するグラフ構造もとりうるが，その場合複数の演算インスタンスを仮想的に 1 つと見なして全体を木構造としたうえで，当該仮想インスタンスの内部で再帰的に木構造を編成することで，本章での議論を適用することが可能である．

行されてゆくと，結果としてインオーダ型クエリ実行器においては，深さ優先探索の順序で演算インスタンス木展開が行われる．つまり，深さ優先探索の順序で演算インスタンス木の各ノード i に番号を振っていき，その番号の大小で演算インスタンスの順序関係を規定すると，これはインオーダ型クエリ実行器が演算インスタンスを実行する順序関係と一致する．

また，根から演算インスタンス i に至るまで，各分岐で選択した子番号の配列である**経路** $\mathrm{route}(i)$ を定義することで，木における演算インスタンスの位置を特定することが可能となる．たとえば図 5 においては，演算インスタンス i_6 は，$i_1 \rightarrow i_2$ (1 番目の子) $\rightarrow i_3$ (1 番目の子) $\rightarrow i_6$ (2 番目の子)，とたどることができるため，$\mathrm{route}(i_6) = [1, 1, 2]$ である．2 つの経路 $\mathrm{route}(i)$，$\mathrm{route}(j)$ について，各分岐で選択した子番号が完全に一致するときには $\mathrm{route}(i) = \mathrm{route}(j)$ であると定義し，また 2 つの異なる経路 $\mathrm{route}(i)$，$\mathrm{route}(j)$ について，次のいずれかの条件を満たすことを $\mathrm{route}(i) < \mathrm{route}(j)$ と定義する．

- $\mathrm{route}(i)$，$\mathrm{route}(j)$ の各分岐で選択する子番号を根から順次照合してゆき，異なる子番号を選ぶ最初の分岐において，$\mathrm{route}(i)$ のほうが子番号の小さい子を選択する
- $\mathrm{route}(i) \neq \mathrm{route}(j)$ かつ，$\mathrm{route}(i)$ が $\mathrm{route}(j)$ に含まれる

このように $\mathrm{route}(i) < \mathrm{route}(j)$ を定義すると，インオーダ型クエリ実行器が i を j の先に実行することと，$\mathrm{route}(i) < \mathrm{route}(j)$ であることは同値となる．

なお，ここに示した演算インスタンスの編成は一例にすぎない．演算インスタンスの粒度は任意に設定可能であるが，粒度設定が性能に与える影響の考察に関しては別稿に譲りたい．

3.2　協調制御アルゴリズム

あるクエリ実行に際して，演算インスタンス木を構成する全インスタンスの情報が事前に与えられると仮定すると，インオーダ型クエリ実行器によるページ発行順序をすべて把握できる．すなわち，アウトオブオーダ型クエリ実行器が先行的に供給するページのうち，インオーダ型クエリ実行器の利用前にバッファから追い出されるものを判別することが可能である．特に利用可能なバッファプール容量，およびページ置換ポリシが既知のときには，入出力待ち時間の縮減に寄与するページを正確に把握することができるため，いっさい無駄のないアウトオブオーダ型クエリ実行器制御アルゴリズム（**神託アルゴリズム**）を構築可能である．ただし実際のデータベースシステムにおいて，演算インスタンス木はクエリ実行を通して順次生成されてゆくものであり，クエリ実行完了をもって初めて木を構成する全インスタンスの情報を知ることが可能となる．つまり，演算インスタンス木を構成する全インスタンスの情報を事前に知ることは困難であるため，神託アルゴリズムに対する近似的な制御アルゴリズムを構築する必要がある．

単純な制御アルゴリズムとしては，演算インスタンスの順序関係のみを考慮して実行スケジュールを行う方法が考えられる．すなわち，アウトオブオーダ型クエリ実行器において生成され，まだ実行されていない演算インスタンスのなかで，インオーダ型クエリ実行器における実行順序が早い演算インスタンスほど優先的に実行をスケジュールするというアルゴリズムである．このアルゴリズムは，1 つの優先度キューを演算インスタンスキューとし，演算インスタンス i の根からの経路を優先度とすることで構成可能である．このアルゴリズムによれば，無制御のアウトオブオーダ型実行と比べて，ページアクセス系列の時間的相関性が高まり，インオーダ型クエリ実行器のバッファヒット率向上が期待される．一方で有限のバッファプール容量を考慮しないため，アウトオブオーダ型クエリ実行器がバッファプール容量を超えて先行的ページ読み込みを行うことは抑制できない．

バッファプール容量の範囲内で先行的ページ読み込みが行われるためには，アウトオブオーダ型クエリ実行器が先行的に実行してもよい演算インスタンスの範囲（**先行実行可能範囲**）を把握する必要がある．仮に，演算インスタンス木を構成する全インスタンスが既知であるとすると，インオーダ型クエリ実行器が実行中の演算インスタンスを始点として，深さ優先探索順で木をトラバースしながら各インスタンスのページアクセス量を足し合わせることで，先行実行可能範囲を算出することができる．しかし実際のクエリ実行中には，**図 6** (a) のように演算インスタンス木は部分的にのみ展開された状態であるため，トラバースを行うことができない．

そこで，最終的に生成される演算インスタンス木を予測しながらクエリ実行を進めることで，バッファプール容量を超過する先行的ページ読み込みを抑制することを図る**スライディングウィンドウ順序制御アルゴリズム**を提案する．提案するアルゴリズムでは，図 6 (b) に示すように演算インスタンス木の未知部分木を予測により補うことで，完全に正確ではないものの先行実行可能範囲を近似的に算出できることが期待される．

図 6 (b) に示す演算インスタンス木の未知部分を予測する具体的な方策として，提案するアルゴリズムにおいては，実際の演算インスタンス木とは別に予測のための演算インスタンス木（**予測演算インスタンス木**）を構築する．演算インスタンス木はクエリ実行が進行するにつれ徐々に生成されてゆくのに対し，予測演算インスタンス木はクエリ実行前に全体を生成した後，クエリ実行時に動的な補正が繰り返し行われ，最終的には実際の演算インスタンス木と同じ形状に収束する．以降，予測演算インスタンス木の

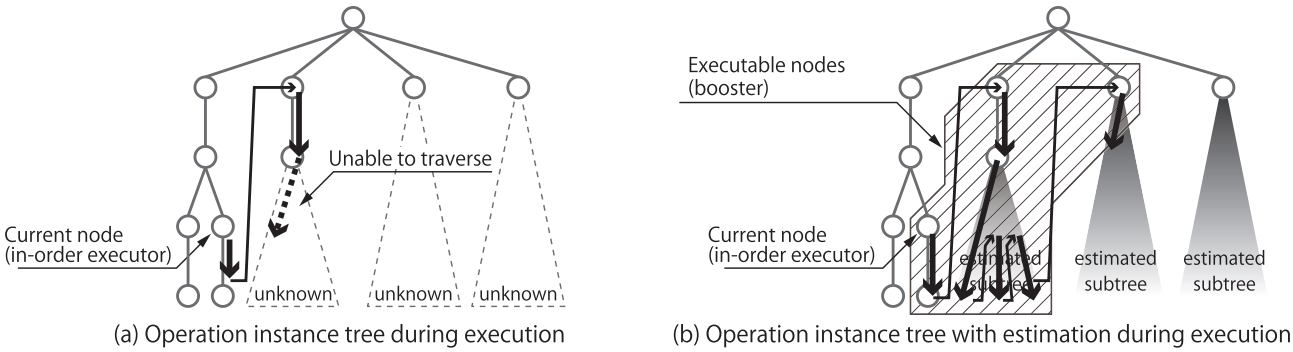


図 6　演算インスタンス木と先行実行可能範囲

Fig. 6　An operation instance tree and a range of instances which can be executed on ahead.

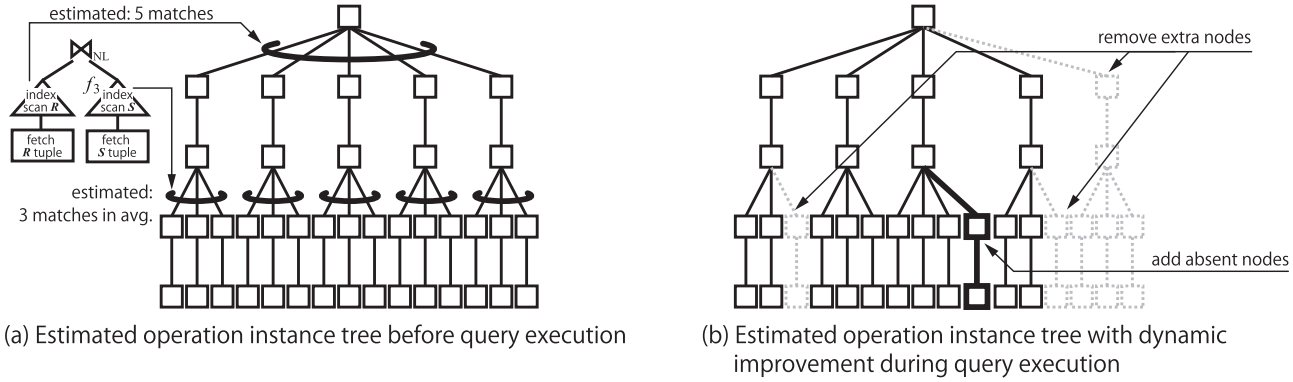


図 7　予測演算インスタンス木の生成とクエリ実行時の動的補正の一例

Fig. 7　An example case of generation and dynamic correction of an estimated operation instance tree.

静的生成手順と動的補正手順を説明した後に，予測演算インスタンス木を用いて先行実行可能範囲を求める手順を説明する．

まず予測演算インスタンス木の静的生成は，データベースが有する各種統計情報[*5]を活用し，各演算インスタンスから生成される子演算インスタンスの平均数を見積もり，当該見積りに従って予測演算インスタンスを生成することで行う．たとえば図 5 に示したクエリプランについて，R 表索引検索で 5 件，S 表索引検索で平均 3 件の索引エントリが該当することが統計情報から推定されたとする．このときには，R 表索引検索の演算インスタンスを親として，R 表タプル取得の演算インスタンスが 5 個生成されると予測されるので，その分だけ予測演算インスタンス木にノードを生成する．また，それぞれの S 表索引検索の演算インスタンスを親として，S 表タプル取得の演算インスタンスが 3 個ずつ生成されると予測されるので，その分だけ予測演算インスタンス木にノードを生成する．その結果として，**図 7** (a) に示す予測演算インスタンス木がクエリ実行前に生成される．

[*5] クエリ実行プラン生成過程で候補の取捨選択を行うために，通常データベースシステムはクエリ実行プランから生じるタプル数を見積もるためのヒストグラム情報，最頻値情報，カーディナリティ情報などの統計情報を有している．

クエリ実行が進行するにつれて，予測演算インスタンス木と実際の演算インスタンス木との差異，すなわち各演算インスタンスにおける子演算インスタンス生成数の違いが明らかとなるので，そのつど予測演算インスタンス木を補正するために，実際の演算インスタンス木と比べて余剰なノードは削除し，不足するノードを追加する．たとえばクエリ実行前に図 7 (a) に示すように R 表索引検索は 5 件該当すると予測して予測演算インスタンス木を生成し，実際のクエリ実行では R 表索引検索で 4 件しか該当しなかった場合，1 件分余剰に部分木が生成された状態となるため，当該部分木を削除する．また S 表索引検索ではそれぞれ 3 件該当すると予測して予測演算インスタンス木を生成し，ある S 表索引検索で 4 件該当した場合，1 件分の部分木が不足した状態となるためこれを予測演算インスタンス木に追加する．このように，クエリ実行前には図 7 (a) に示す予測演算インスタンス木が生成されるが，クエリ実行時の動的補正の結果図 7 (b) に示す状態へと変化するような場合が考えられる．予測演算インスタンス木に随時補正を行うと，最終的には実際の演算インスタンス木と同一の木構造へと収束する．

このようにして時々刻々と更新される予測演算インスタンス木に基づいて，当該アルゴリズムは実行スケジュール可

能な演算インスタンスを制限する**演算インスタンスウィンドウ**を決定し，これによってアウトオブオーダ型クエリ実行器の制御を行う．演算インスタンスウィンドウは，演算インスタンス i の経路 route(i) の制約 $w_{low} \leq \text{route}(i) \leq w_{up}$ であり，この制約を満たす演算インスタンスのみを実行可能インスタンスと見なすことで，バッファプール容量を超過しない範囲での先行的データ読み込みが行われることが期待される．w_{low} と w_{up} はインオーダ型クエリ実行器におけるクエリ実行の進行に合わせて，**図 8** に示す手順で更新を行う．

(1) w_{low} を route(i_{IN}) に設定する．ただし，i_{IN} はインオーダ型クエリ実行器が実行中の演算インスタンスを指すものとする．
(2) 予測演算インスタンス木上で，route(i'_{IN}) $= w_{low}$ なる予測演算インスタンス i'_{IN} を特定する．予測演算インスタンス木における経路の定義は演算インスタンス木におけるものと同様とする．
(3) i'_{IN} を基点として，深さ優先探索の順序でトラバースを開始する．トラバース中は，各予測演算インスタンス i の実行に必要なページ数 #page(i) を足し合わせてゆき，$\sum$ #page(i) がバッファプール容量 B を超過する 1 つ手前の予測演算インスタンスでトラバースを停止する．そして当該インスタンス $i'_{\max}$ の経路 route($i'_{\max}$) を w_{up} に設定する．

この手順を，インオーダ型クエリ実行器が実行する演算インスタンスが更新されるたびに行い，ウィンドウを順次スライドさせてゆく．

アウトオブオーダ型クエリ実行器における演算インスタンスの実行スケジュールに際して，上記手順で得られた演算インスタンスウィンドウ $[w_{low}, w_{up}]$ の範囲内にある演算インスタンスのみを実行スケジュール対象とすることで，先行的読み込みが行われるデータベースページが，バッファプール容量を超過しない範囲に制限されることが期待される．また演算インスタンスウィンドウの算出においては，各演算インスタンス実行において要求されるページが相異なることを仮定しているが，実際には異なる演算インスタンスが同一のページを要求することがしばしばあるため，バッファプールの正味使用量は容量 B を下回ると考えられる．

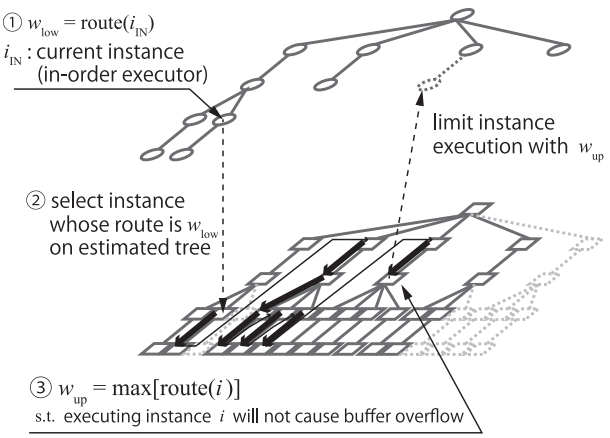


図 8　演算インスタンスウィンドウの更新手順
Fig. 8　Procedure of updating operating instance window.

4. 加速機構を考慮したページ置換ポリシ

加速機構のアウトオブオーダ型クエリ実行器による先行的ページ読み込みは，バッファマネージャへのページ要求発行によって行われる．そのため，先行的に供給されたページが実際にインオーダ型クエリ実行器によって消費されるか，すなわちインオーダ型クエリ実行器が当該ページを要求するまでバッファプール上に保持されるか否かは，バッファマネージャの採用する**ページ置換ポリシ**に委ねられる．バッファマネージャは新たなページ要求を受付けると，当該ページ格納領域を確保すべく，ページ置換ポリシに従って選択されたページをバッファプールから追い出す．この際，先行的に供給され未消費であるページが追い出し対象として選択されたとすると（**未消費ページ追い出し**），インオーダ型クエリ実行器は当該ページに関して入出力待ち時間の縮減効果を得ることができない．すなわち，加速機構を用いるデータベースシステムにおいては，ページ置換ポリシの性質として未消費ページ追い出しの発生頻度が低いことが望まれる．

ページ置換ポリシにおいては，追い出し対象ページは参照局所性の低いものが優先して選択されることが一般的であるが，未消費ページはアウトオブオーダ型クエリ実行器にのみ参照されているのに対し，消費済みページは両クエリ実行器から参照されているために参照局所性が高いと判断され，結果として未消費ページが優先して追い出し対象として選択されてしまう可能性が考えられる．その一例を**図 9** に示す．この例では，インオーダ型クエリ実行器がページ A から I まで全 9 ページを順に要求するクエリ処理を，容量 6 ページのバッファプールを用いて行う．まず，アウトオブオーダ型クエリ実行器がページ A から F まで

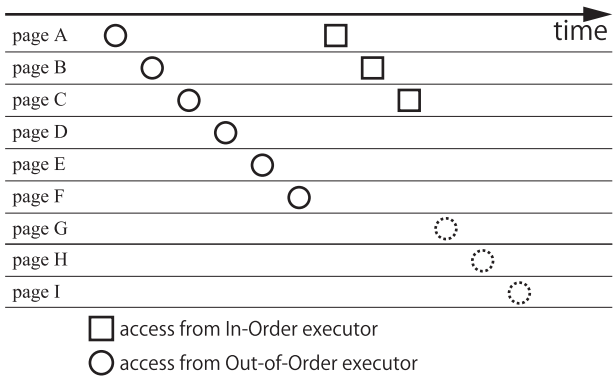


図 9　未消費ページ追い出しの発生する一例
Fig. 9　An example of not-consumed page eviction.

順に要求した後に，インオーダ型クエリ実行器がページ A から C まで順に要求し，図示した時点で全 6 ページはすでに解放済みで追い出し可能であるとする．次にアウトオブオーダ型クエリ実行器がページ G を要求すると，ページ A から F の中で追い出し対象を選択する必要が生じる．このとき，インオーダ型クエリ実行器が次に要求するのはページ D であるが，ページ D は最終参照時刻が最も古く，参照回数が 1 回であるため，多くのページ置換ポリシにおいて追い出し対象として選択される可能性が高い．ページ D が追い出されると，その後のインオーダ型クエリ実行器のページ D 要求はバッファミスする．また後続するページ H，I の要求によって，ページ E，F も同様に未消費ページ追い出しがなされ，インオーダ型クエリ実行器がバッファミスする可能性が高いと予想される．

この問題は，ページ置換ポリシにおける追い出し対象選択の優先度基準として，未消費ページか否かという情報を加味するよう拡張することで，その発生を大幅に抑制することができる．具体的なポリシ拡張の方法は 1 通りに限らないが，最も単純なアプローチとしては，ページの管理情報に未消費ページであるか否かを示す状態変数を付与し，追い出し対象ページ選択の際には，まず未消費ではないページの中から従前のポリシに従って追い出し対象を選択し，追い出し可能ページが見つからない場合には未消費ページの中から従前のポリシに従って追い出し対象を選択する，といったものが考えられる．このアプローチをとる場合には，追い出し対象選択の論理はほとんど従前のものを変更することなく実現可能であり，アウトオブオーダ型クエリ実行器が動作しない限りバッファマネージャの挙動は従前と同様である．

5. PostgreSQL 版プロトタイプ実装

提案する加速機構の試作実装として，オープンソースデータベース管理システム PostgreSQL を対象とした加速機構モジュール PgBooster を開発した．PgBooster を用いることで，PostgreSQL の既存エンジン（クエリ実行器）のクエリ実行方式は変更することなく，そのクエリ実行を飛躍的に高速化することができる．また，PgBooster が無効化されている場合には，性能を含め振舞いのいっさいが通常版 PostgreSQL と同様となる．

通常版 PostgreSQL のクエリ処理フローを次に示す：(1) 構文解析器がクエリ構文解析木を生成し，(2) 最適化器がクエリ実行プランを生成し，(3) クエリ実行器がクエリ実行プランを実行し，実行結果をユーザに送信する．PgBooster を組み込んだ場合，基本的なフローは変わらず，(2) の後に生成されたプランを PgBooster へ送信するというステップが追加される．PgBooster はプランを受付けると，加速効果が見込まれる場合には当該プランのアウトオブオーダ型実行を開始し，さもなくばいっさいの処理を行わない．このように処理フローを設計することで，既存クエリ実行器の挙動はいっさい変えることなく，効果の見込まれる場合のみ PgBooster による加速を実施できる．

また実装レベルにおいても，PostgreSQL が依拠する 1 機能 1 プロセスというモデルと，プロセス間共有メモリ・シグナルによるプロセス間通信という枠組みに則り，既存クエリ実行器を加速する枠組みを構築した．すなわち，既存クエリ実行器のプロセスに従属する形で PgBooster のプロセス群（アウトオブオーダ型クエリ実行コーディネータプロセスおよびアウトオブオーダ型クエリ実行器プロセス）を立ち上げ，アウトオブオーダ型クエリ実行器は既存実行器と論理的に独立したメモリ空間で動作する．

PgBooster による加速を実現するために，次にあげる点に関しては PostgreSQL 本体に変更を加えている：(A) クエリ実行プランを PgBooster に送信するため，プランを共有メモリ領域へと複製し，PgBooster にシグナルで通知する．(B) PgBooster の協調制御に必要な情報として，既存クエリ実行器で実行している最中のデータベース演算インスタンスを把握し，定期的に規定の共有メモリ領域へと書き出す．(C) バッファマネージャにおける GCLOCK ページ置換ポリシに 4 章の議論を反映させるために，バッファプール上のページ管理情報に未消費ページを示すフラグ変数を追加するとともに，アウトオブオーダ型クエリ実行器によってバッファプールに読み込まれたページは当該フラグを真に設定し，インオーダ型クエリ実行器により 1 度でもアクセスされたページは当該フラグを偽に設定する処理を追加する．ただし，いずれも PostgreSQL 本来のクエリ実行方式に変更を加えるものではない．

最新バージョンの PostgreSQL [*6] は本体のみで約 230 万行のプログラムコードからなっている．一方で，PgBooster の開発に係るコードの規模は，PostgreSQL 本体に対する変更 (A)(B)(C) のための書き換え 16 行および追加 2,193 行，PgBooster 自体の新規コード 4,371 行のみであり，PostgreSQL 全体の 0.3%程度である．PostgreSQL 本体の変更は処理追加がほとんどであり，バージョン管理システムを用いて最新版の PostgreSQL に追随し続けることも比較的容易である．

6. 評価実験

6.1 実験環境

提案手法の評価実験を実施するにあたり，ミッドレンジクラスのサーバとディスクストレージを備える実験システムを構築した．サーバは 4 プロセッサ 24 物理コア[*7] と 32 GB の主記憶を搭載し，64 bit 版 RedHat Enterprise Linux Server

[*6] 2013/9/5 時点の最新バージョンは PostgreSQL 9.2.4.
[*7] 1 プロセッサあたり 6 物理コアを有する Intel Xeon X7460 2.66 GHz を 4 プロセッサ搭載．

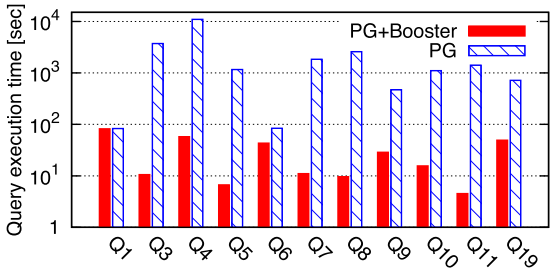


図 10　TPC-H 類似クエリ実行時間
Fig. 10　Execution time of modified TPC-H queries.

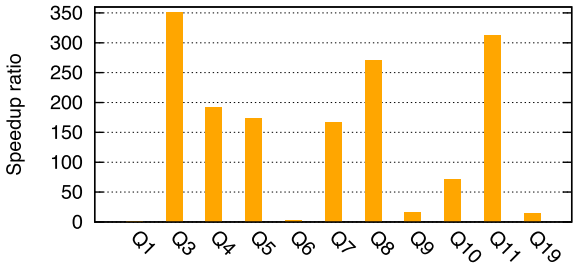


図 11　PG+Booster による TPC-H 類似クエリ加速率
Fig. 11　PG+Booster speedup ratio of modified TPC-H queries.

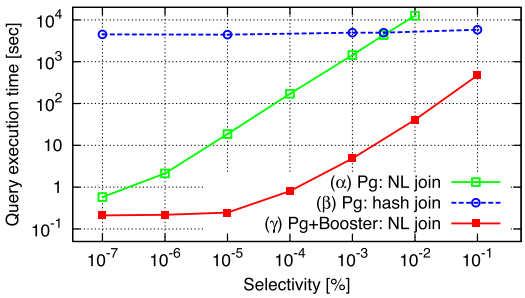


図 12　各実行方法における選択率とクエリ実行時間の関係
Fig. 12　Selectivity v.s. query execution time for each execution method.

5.8[*8]が OS として動作する．ディスクストレージは 160 台の 450 GB 15,000 rpm FC HDD と，8 GB のキャッシュメモリを持つディスクシステム IBM DS5300 からなる．ディスクシステム内では 8 HDD ごとに 7D+1P の RAID グループが編成され，1 つの RAID グループあたり 1LU，合計 20LU を構成した．サーバとディスクシステムは 8 本の 4 Gbps ファイバチャネルにより接続され，20LU をソフトウェア RAID0 によってストライピングし，ext4 ファイルシステムによってフォーマットしてデータベース領域とした．

加速機構は選択性を有する分析的クエリ実行を大幅に高速化することが期待されるため，評価実験においては，分析的ワークロードの標準的なデータセットとして意思決定支援系ベンチマーク TPC-H [9] のデータセットを採用し，Scale Factor = 1,000 で生成した 1 TB のデータセットを利用した[*9]．クエリ実行時間を測定する際には，測定前に PostgreSQL を再起動し，また OS のファイルキャッシュおよびストレージコントローラのキャッシュを消去するという手順を毎回行った．PostgreSQL のバッファプール容量は 256 MB として実験を行った．

以降の説明では，通常版 PostgreSQL を指して **PG**，また PgBooster が有効化された PostgreSQL を指して **PG+Booster** と省略する場合がある．

6.2　TPC-H 類似クエリによる処理性能評価

PgBooster によるデータベースエンジン加速効果を確認するため，TPC-H 規定クエリのうち現時点で PgBooster の実装が実行をサポートする 11 クエリを用いて性能評価実験を行った．実験に際しては，PgBooster による大幅な性能向上が見込まれるクエリの性能評価を目的とするため，各クエリの選択率が 1%以下となるよう調整を行ったうえで，PG および PG+Booster における実行時間を測定した．

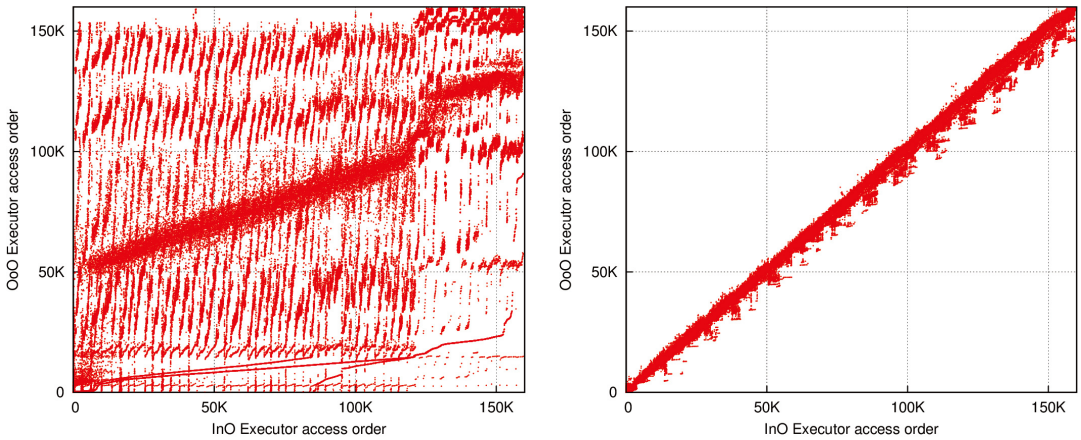
それぞれのクエリ実行時間を**図 10** に，また PG に対する PG+Booster の加速率を**図 11** に示す．このグラフより，複数のクエリにおいて PG+Booster は 100 倍以上の加速率を実現し，最大で 350 倍に達することが確認できた．クエリの中でも，Q3 や Q8，Q11 に代表されるように，索引走査に起因する入出力パターンのランダム性が高く，PG では 160HDD の入出力帯域をほとんど活用できていない傾向が強いクエリほど加速率が高くなる結果となった．一方で加速率が低いクエリは，シーケンシャル性が高いアクセスパターンを生じ，順次先読み効果によって PG でも入出力帯域を活用できる傾向が強いものである．いずれのクエリについても PG+Booster は PG の処理性能を下回ることはなかった．

以上の結果より，PgBooster は従前の PostgreSQL のクエリ処理性能をベースラインとして，これまで入出力帯域を有効活用できなかったクエリの処理性能を大幅に加速可能であることが確認された．

6.3　クエリ選択率と加速効果

本実験では，PgBooster の有効領域を明らかにするため，(α) PG でネステッドループ（NL）結合および索引走査を用いたプランを実行，(β) PG でハッシュ結合および全件走査を用いたプランを実行，(γ) PG+Booster で NL 結合および索引走査を用いたプランを実行，という 3 つの実行方法それぞれにおいて TPC-H Q3（customer 表，orders 表，lineitem 表の結合クエリ）を実行し，その実行時間を測定した[*10]．実験に際しては，Q3 の選択条件を調整すること

[*8] OS カーネルは Linux 2.6.18-308.el5.
[*9] PostgreSQL ロード後の容量は約 2.5 TB.
[*10] いずれの場合も left-deep であり，外表から順に customer，orders，lineitem の順で結合するプランを用いた．



(a) PG+Booster without Coordinator　　(b) PG+Booster with Coordinator

図 13　インオーダ型・アウトオブオーダ型クエリ実行器におけるページアクセス時間相関
Fig. 13　Temporal correlation of access patterns of In-order and Out-of-Order executor.

でクエリの選択率を最小で 10^{-7}%から最大で 0.1%の間で変化させて測定を行った．

結果を**図 12** に示す．x 軸は選択率を，y 軸はクエリ実行時間を表す．いずれも対数軸となっていることに注意されたい．(α) の NL 結合は実行時間が選択率にほぼ比例する一方，(β) のハッシュ結合では全件走査の入出力時間に律速され，実行時間は 4,500 秒から 6,000 秒程度と選択率による影響は比較的小さい．

NL 結合を PG+Booster で実行する (γ) のクエリ実行時間は，論理的にはアウトオブオーダ型実行によって入出力が多重化された分だけ (α) が高速化されたものである．実験結果においても，始動コストが顕在化する選択率 10^{-7}%から 10^{-5}%を除けば，実行時間は選択率にほぼ比例し，(α) に対して 200 から 300 倍前後の高速化が達成された．また測定を行った範囲においては，PG+Booster の NL 結合実行は PG のハッシュ結合実行よりも高速であった．

本実験により，選択率 10^{-7}%から 0.1%のクエリに関しては，PG のいずれの実行方法よりも PG+Booster が高速であり，PgBooster による加速効果を確認することができた．

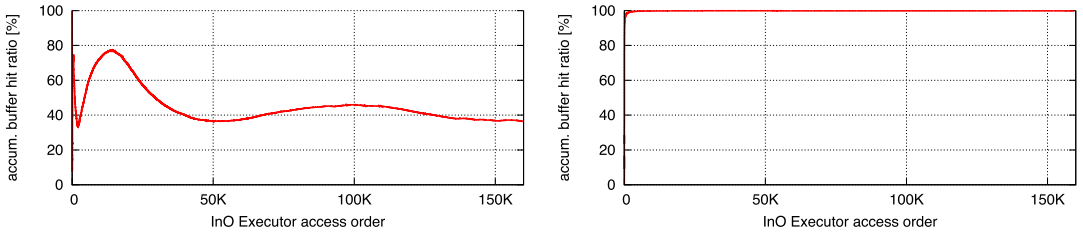
6.4　アウトオブオーダ型クエリ実行の協調的制御アルゴリズム評価

本実験では，アウトオブオーダ型クエリ実行コーディネータの制御により，ページアクセス系列の時間的相関性が実際に達成されていることを確認するため，(a) コーディネータを無効化した場合，(b) コーディネータを有効化した場合のそれぞれについて，PG+Booster で選択率調整済みの TPC-H Q3 を実行した．

結果を**図 13** に示す．図 13 は，1 つの点が 1 回のデータベースページアクセスを示し，x 軸はインオーダ型クエリ実行器における当該アクセスの発行順序，y 軸はアウトオブオーダ型クエリ実行器における発行順序を表す．つまり，インオーダ型クエリ実行器とアウトオブオーダ型クエリ実行器がまったく同じ順番でデータベースページアクセスを行うとすべての点は直線 $x = y$ 上に存在する状態となり，逆に時間的相関性が小さいほど点は広い範囲に拡散した状態となる．

まず図 13 (a) より，コーディネータを無効化した場合はインオーダ型クエリ実行器とアウトオブオーダ型クエリ実行器のページアクセス系列は，ほとんど時間的相関性が見いだせないことが分かる．一方でコーディネータを有効化した場合には，図 13 (b) に示すようにページアクセス系列の時間的相関性が高く保たれたことが分かる．この結果から，当該クエリ実行においてコーディネータの制御アルゴリズムが有効に動作することが確認できた．

また，加速機構がインオーダ型クエリ実行器にもたらすページ供給作用を示したグラフが**図 14** である．x 軸は，インオーダ型クエリ実行器における各データベースページアクセスの発行順序を表し，y 軸は当該データベースページアクセスまでのインオーダ型クエリ実行器の累積バッファヒット率を表す．図 14 (a) のコーディネータ無効状態では，実行開始直後はカタログページアクセスによってヒット率が 100%となるが，テーブルアクセスが始まると累積ヒット率は 33%程度までいったん低下した．その後，無秩序ながらもアウトオブオーダ型クエリ実行器によってバッファプールにページが先行的に読み込まれるため，約 13,000 回目前後のアクセスで 76%まで累積ヒット率は上昇したものの，以降は再び低下し 40%前後を推移した．一方で図 14 (b) のコーディネータ有効状態では，クエリ開始直後から累積ヒット率が 99%以上を維持し，最終的には 99.8%という結果となった．以上より，当該クエリ実行において加速機構が有効に作用し，インオーダ型クエリ実行器におけるバッファヒット率を大幅に向上させたことが分



(a) PG+Booster without Coordinator　　(b) PG+Booster with Coordinator

図 14　インオーダ型クエリ実行器における累積バッファヒット率の推移
Fig. 14　Accumulative buffer hit ratio of In-order executor over query execution.

かる．

7. 関連研究

提案手法の加速機構は，既存データベースに先行的にページ供給を行うため，ある種のデータ先読み機構という見方もできる．入出力インテンシブなソフトウェアにおいて，先読みは入出力遅延を隠蔽するための有効な手段であり，アクセスの順次性を用いた先読み [10], [11]，アクセス履歴からの予測による先読み [12]，アプリケーションから与えられるヒント情報に基づく先読み [13], [14]，データブロック間の相関性マイニングに基づく先読み [15] など様々な方法がある．これらはいずれもアプリケーションと分離された階層において先読みを行う方法であり，論理的に等価なアプリケーション処理を異なる方式で実行して先読み効果を実現する提案手法とは本質的に異なる．データベースシステムのクエリ実行プランに基づいた先読みとしては，検索に該当する索引ページエントリに対して先読みを行う方法 [16]，当該処理をストレージコントローラにおいて多段展開する方法 [17], [18] などがあるが，いずれもクエリ処理の入出力を部分的に予測し先読みするものであり，データベースエンジンと連携してクエリ処理そのものを行うことで先読み効果を得る提案手法とは異なる．

アプリケーション処理を重複して実行することで高速化を図るアプローチは，プロセッサアーキテクチャのレベルでは，余剰コアを用いた先行的実行による CPU キャッシュミス低減手法が数多く研究されており，汎用の実行コア以外の特別な回路を必要としない手法 [19], [20] などがある．一方で，データベースシステムのレベルで異なる実行方式を組み合わせる手法は我々の知る限りにおいて見られず，さらに既存エンジンの挙動を変えないという特徴を持つ提案手法は新規な取り組みである．

アウトオブオーダ型クエリ実行器における動的タスク分解は，データベース演算における一種のループ分解と見なすことができる．データベース演算のループ分解については，データフロー解析を用いたプログラム変換による手法 [21]，機械学習を用いたワークロードごとの最適なループ分解方法選択 [22] といった取り組みが見られる．

8. おわりに

本論文では，既存データベースエンジンのクエリ実行方式を変更することなく，アウトオブオーダ型データベースエンジンと同水準まで処理性能を向上させることを目的として，アウトオブオーダ型クエリ実行器およびアウトオブオーダ型クエリ実行コーディネータからなる加速機構を既存データベースエンジンに組み込む手法を提案した．そして，オープンソースデータベース管理システム PostgreSQL を対象とした試作実装 PgBooster の開発を通して，既存データベースエンジンのクエリ実行方式を変えることなく加速機構を設計・実装することが可能であることを確認した．1 TB の TPC-H データセットを用いて PgBooster の加速効果を評価した結果，最大で Q3 の 350 倍，複数のクエリで 100 倍以上の加速効果を確認した．また PgBooster を用いることで，Q3 の選択率が 10^{-7}%から 0.1%の領域においては，通常版 PostgreSQL のいずれのクエリ実行方式よりも高い処理性能を達成可能であると確認した．さらに，インオーダ型およびアウトオブオーダ型クエリ実行器の微視的挙動を観測し，PgBooster におけるアウトオブオーダ型クエリ実行コーディネータの制御アルゴリズムが有効に機能して加速効果を生んでいることを確認した．

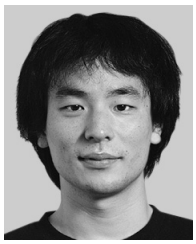
今後は複数の結合方式や書き込みを含むクエリ実行，複数クエリの同時実行などといったより複雑なワークロードにおいても本手法の有効性を検証するとともに，アウトオブオーダ型クエリ実行の協調制御アルゴリズムを高度化し，さらに多様なクエリに対して精度を高めてゆきたい．

謝辞　本研究の一部は内閣府最先端研究開発支援プログラム「超巨大データベース時代に向けた最高速データベースエンジンの開発と当該エンジンを核とする戦略的社会サービスの実証・評価」，および日本学術振興会科学研究費補助金（特別研究員奨励費）24·8381 の助成により行われた．

参考文献

[1]　Lis, M., Ren, P., Cho, M.H., Shim, K.S., Fletcher, C.W., Khan, O. and Devadas, S.: Scalable, accurate multicore simulation in the 1000-core era, *Proc. IEEE Interna-*

tional Symposium on Performance Analysis of Systems and Software, *ISPASS '11*, pp.175–185, IEEE Computer Society (2011).
[2] Gantz, J. and Reinsel, D.: The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, Technical report, IDC (2012).
[3] 喜連川優，合田和生：アウトオブオーダー型データベースエンジン OoODE の構想と初期実験，日本データベース学会論文誌，Vol.8, No.1, pp.131–136 (2009).
[4] 合田和生，豊田正史，喜連川優：アウトオブオーダ型データベースエンジン OoODE の試作とその実行挙動，第 5 回データ工学と情報マネジメントに関するフォーラム (2013).
[5] 早水悠登，合田和生，喜連川優：アウトオブオーダ型データベースエンジン OoODE によるクエリ処理性能の実験的評価，第 5 回データ工学と情報マネジメントに関するフォーラム (2013).
[6] 喜連川優：新原理「非順序型実行」で，データ処理速度 1000 倍も可能に，*NII Today*, No.59, pp.4–5 (2013).
[7] 清水　晃，徳田晴介，田中美智子，茂木和彦，合田和生，喜連川優：アウトオブオーダ型データベースエンジン OoODE におけるタスク管理機構の一実装方式の評価，第 5 回データ工学と情報マネジメントに関するフォーラム (2013).
[8] 早水悠登，合田和生，喜連川優：アウトオブオーダ型クエリ実行に基づくプラグイン型データベースエンジン加速機構，第 6 回 Web とデータベースに関するフォーラム，WebDB Forum 2013 (2013).
[9] Transaction Processing Performance Council, available from ⟨http://www.tpc.org/⟩.
[10] Feiertag, R.J. and Organick, E.I.: The Multics Input/Output system, *Proc. 3rd ACM symposium on Operating systems principles*, *SOSP '71*, pp.35–41, ACM (1971).
[11] Smith, A.J.: Sequentiality and prefetching in database systems, *ACM Trans. Database Syst.*, Vol.3, No.3, pp.223–247 (1978).
[12] Palmer, M. and Zdonik, S.B.: Fido: A Cache That Learns to Fetch, *Proc. 17th International Conference on Very Large Data Bases*, *VLDB '91*, pp.255–264, Morgan Kaufmann Publishers Inc. (1991).
[13] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, J.: Informed prefetching and caching, *Proc. 15th ACM symposium on Operating systems principles*, *SOSP '95*, pp.79–95, ACM (1995).
[14] Mowry, T.C., Demke, A.K. and Krieger, O.: Automatic compiler-inserted I/O prefetching for out-of-core applications, *Proc. 2nd USENIX symposium on Operating systems design and implementation*, *OSDI '96*, pp.3–17, ACM (1996).
[15] Li, Z., Chen, Z., Srinivasan, S.M. and Zhou, Y.: C-Miner: Mining Block Correlations in Storage Systems, *Proc. 3rd USENIX Conference on File and Storage Technologies*, *FAST '04*, pp.173–186, USENIX Association (2004).
[16] Oracle Corporation: Performance and Scalability in DSS Environment with Oracle9i (2001).
[17] 向井景洋，根本利弘，喜連川優：高機能ディスクにおけるアクセスプランを用いたプリフェッチ機構に関する評価，第 11 回データ工学ワークショップ DEWS2000 (2000).
[18] 出射英臣，茂木和彦，西川記史，大枝　高：クエリプランを利用した先読み技術の開発と初期評価，第 16 回データ工学ワークショップ DEWS2005 (2005).
[19] Ganusov, I. and Burtscher, M.: Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors, *Proc. 14th International Conference on Parallel Architectures and Compilation Techniques*, *PACT '05*, pp.350–360, IEEE Computer Society (2005).
[20] Zhou, H.: Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window, *Proc. 14th International Conference on Parallel Architectures and Compilation Techniques*, *PACT '05*, pp.231–242, IEEE Computer Society (2005).
[21] Chavan, M., Guravannavar, R., Ramachandra, K. and Sudarshan, S.: Program transformations for asynchronous query submission, *2011 IEEE 27th International Conference on Data Engineering* (*ICDE*), pp.375–386 (online), DOI: 10.1109/ICDE.2011.5767870 (2011).
[22] Răducanu, B., Boncz, P. and Zukowski, M.: Micro Adaptivity in Vectorwise, *Proc. 2013 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '13*, pp.1231–1242, ACM (online), DOI: 10.1145/2463676.2465292 (2013).



早水 悠登　（学生会員）

平成 21 年東京大学工学部電子情報工学科卒業．平成 23 年同大学院情報理工学系研究科電子情報学専攻修士課程修了．現在，同専攻博士課程 3 年．高速データベースエンジンに関する研究に従事．日本学術振興会特別研究員 DC2.

合田 和生　（正会員）

平成 12 年東京大学工学部電気工学科卒業，平成 14 年同大学院工学系研究科電子情報工学専攻修士課程修了，平成 17 年同大学院情報理工学系研究科電子情報学専攻博士課程単位取得満期退学．同年博士（情報理工学）．日本学術振興会特別研究員，東京大学生産技術研究所産学官連携研究員等を経て，現在，東京大学生産技術研究所特任准教授．データベースシステム，ストレージシステムの研究に従事．日本データベース学会，ACM，IEEE，USENIX 各会員．

喜連川 優　（フェロー）

昭和 58 年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，工学博士．東京大学生産技術研究所教授，東京大学地球観測データ統融合連携研究機構長，平成 25 年 4 月より国立情報学研究所所長，平成 25 年 6 月より情報処理学会会長を務める．データベース工学の研究に従事．内閣府最先端研究開発支援プログラムを中心研究者として推進中．電子情報通信学会業績賞，情報処理学会功績賞，ACM SIGMOD Edgar F. Codd Innovations Award 受賞．ACM，IEEE，電子情報通信学会ならびに情報処理学会フェロー．

（担当編集委員　的野 晃整）