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

早水 悠登[1,a)]　合田 和生[3]　喜連川 優[2,3]

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の挙動を変えることなく，その処理速度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機能をプラグイン型加速機構として組み込む手法を提案する．当該機構は既存エンジンのクエリ実行と並行して，協調的にクエリをアウトオブオーダ型実行する．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，既存エンジンの入出力待ち時間を縮減し，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，ミッドレンジクラスのサーバ・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．

1. はじめに

昨今の計算機システムにおける演算・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 コア程度は既に一般的である．また 1000 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の増加傾向は明らかと言えよう．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 1000 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] の開発を進めている．当該データベースエンジンは，**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る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**，即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クエリ処理において高い性能を発揮することが知られている [4][5].

本論文では，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 1000 万行規模のプログラムコードから成り，また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．インオーダ型データベースエンジンの挙動を保全しながら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

[1] 東京大学大学情報理工学系研究科電子情報学専攻
[2] 国立情報学研究所
[3] 東京大学生産技術研究所
[a)] haya@tkl.iis.u-tokyo.ac.jp

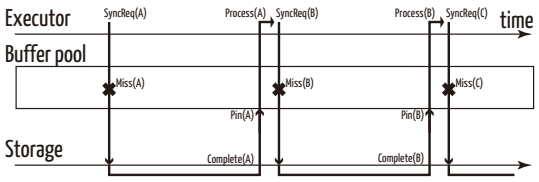


図 1 インオーダ型クエリ実行
Fig. 1 In-order query execution

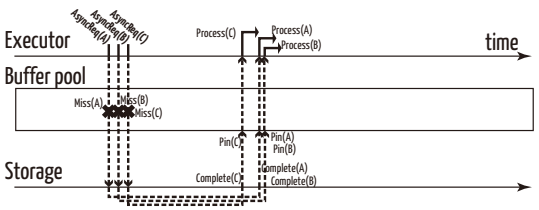


図 2 アウトオブオーダ型クエリ実行
Fig. 2 Out-of-Order query execution

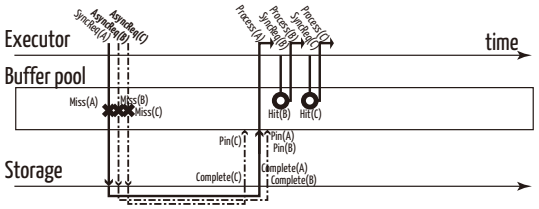


図 3 先行的なアウトオブオーダ型クエリ実行を伴うインオーダ型クエリ実行
Fig. 3 In-order query execution with leading Out-of-Order execution

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言えよう．

本論文では，既存のインオーダ型データベースエンジン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型の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，その挙動を変えることなく高速化を実現する．提案手法では等価なクエリ実行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，入出力効率の向上を余剰プロセッサコアの活用によって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，当該機構が内包する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，段階的なデータベースエンジン自体のアウトオブオーダ型化の足掛かりとすることもできる．

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 を対象として著者らが開発した加速機構の試作実装 PgBooster 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データベースサーバにて，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通りである．

第 2 節では加速機構の概要を説明し，第 3 節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．第 4 節では試作実装 PgBooster について説明する．第 5 節では PgBooster の評価実験について述べ，第 6 節で関連する研究について言及した後，第 7 節で本論文をまとめる．

2. 既存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上で実行器へ返す．

クエリ実行器はクエリの実行論理を司るコンポーネントであるので，インオーダ型データベースエンジンといった場合，クエリ実行器がインオーダ型クエリ実行方式を採用していることと同義である．その挙動の模式を**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は高々 1 であり，また入出力を待つ間プロセッサはアイドル状態となる．一方で，アウトオブオーダ型クエリ実行器は非同期的にページ要求を多重発行し，取得が完了したページごとに並列に演算処理が行われる．入出力要求は OS，ホストバスアダプタ，ストレージコントローラ，ディスクドライブそれぞれの中でスケジューリングされるため，その発行順序から生じる完了順序は毎回異なることが一般的であり，アウトオブオーダ型クエリ実行器の挙動は本質的に非決定的である．その

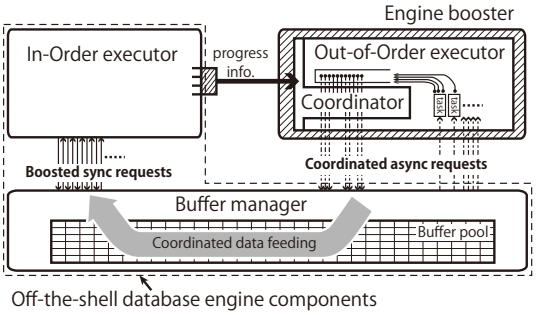


図 4 既存データベースエンジンと加速機構の概要
Fig. 4 Overview of off-the-shell database engine with Engine Booster

挙動を**図 2** に示す．高多重入出力発行と高並列演算実行により，単一クエリの実行であっても，複数のプロセッサコアの並列演算性能や，多数のディスクドライブからなるストレージの入出力帯域を高効率に活用し，特に中程度の選択性を持つクエリにおいて高いクエリ処理性能を有する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が優位性を持つ選択的クエリを，インオーダ型とアウトオブオーダ型の 2 つの実行器で同時に実行することを考えてみたい．この場合のインオーダ型クエリ実行器の挙動を**図 3** に示す．両実行器は最終的に同じ結果を出力するため，その過程でアクセスするページの集合も大部分は一致するはずである．アウトオブオーダ型クエリ実行器が先行的にクエリ実行を進め，インオーダ型クエリ実行器がこれから要求するページが予めバッファプールに読み込まれていたとすると，当該要求はバッファヒットして入出力待ちなしでクエリ実行を進行できる．これは，バッファプールを介して間接的に接続されたアウトオブオーダ型クエリ実行器からページが供給され，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作用がもたらされたと見なすことができる．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作用が有効に働くためには，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，それに応じてアウトオブオーダ型クエリ実行器の非同期入出力およびデータベース演算実行をスケジューリングし，協調的なアウトオブオーダ型実行によるページ供給作用の実現を図る．つまり，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，コーディネータのスケジューリング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．

次節において，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．

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

本提案手法が扱うクエリは読み込むデータ総量がバッファプール容量を大きく超えるものを想定している．アウトオブオーダ型クエリ実行器による先行的なページ読み込みのタイミングが早すぎると，インオーダ型クエリ実行器が後々要求するページであったとしても，要求される時にはバッファプールから追い出されており，ページ供給作用をもたらさない．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 [*1] と，その演算を実行する**コンテクスト** c である．例えば 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等，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以て開始される．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．例えば**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成るプランを考える．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

[*1] 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一対一対応している必要はない

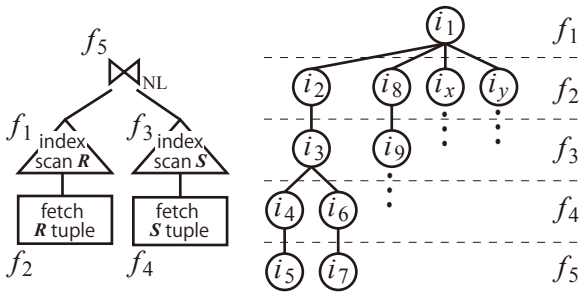


図 5 2 表の結合を行うクエリ実行プランと演算インスタンス木
Fig. 5 Query execution plan of joining two tables and its execution instance tree

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の木構造を考えることができる [*2]．この木構造をクエリの**演算インスタンス木**と呼ぶ．

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においては，実行中に要する空間計算量を最小とするのは深さ優先探索のマナーに従った順序であり，既存のインオーダ型クエリ実行器においてはそのように設計・実装されることが一般的である．つまり，深さ優先探索によって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．また任意のインスタンス i, j について，根からそれぞれに至る経路がわかっていれば，インスタンス木の全容が明らかでなくとも順序関係は決定できるという性質は自明である．

これら演算インスタンス木の構造は，インオーダ型，アウトオブオーダ型問わず実行器一般に対して規定されるものである．つまり，アウトオブオーダ型クエリ実行コーディネータにおける制御アルゴリズムは，演算インスタンス木におけるインオーダ型クエリ実行器の実行中ノード，演算インスタンス木の形状，その他バッファプール容量をはじめとする各種資源量などの情報をもとに，ページ供給作用を生じることが見込まれる演算インスタンスの実行スケジューリングを行えばよい．

ここで，1 クエリの実行にあたって利用可能なバッファプール容量が既知の一定量であり，またページ置換アルゴリズムが既知であり，クエリ実行前に当該クエリの演算インスタンス木全容が既知であると仮定する．この仮定のもとでは，演算インスタンス木の全容から，インオーダ型クエリ実行器が発行する全ページ要求の順序を調べることが可能であり，任意の時点において先行的読み込みがページ供給作用をもたらすか否かを判断可能であるので，一切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前述の仮定が成り立つことは稀であるため，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．即ち，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，最も小さい演算インスタンスから順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，生成された演算インスタンスのうち，インオーダ型クエリ実行器で実行される順番が早いものほど早く実行がスケジュールされるため，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．

本提案手法の評価に際しては，前述の制御アルゴリズムを改善した**スライディングウィンドウ順序制御アルゴリズム**を採用したが，紙面の制約上別稿にその詳細を譲り，ここではその概略を述べるに留める．このアルゴリズムは，データベースシステムの統計情報や，実行時に順次判明する演算インスタンス木の形状情報を利用し，ページ供給効果をもたらすと予想される演算インスタンスの上限と下限，即ち演算インスタンスウィンドウを時々刻々と更新する．そして，生成され未だ実行されていない演算インスタンスのなかで，ウィンドウの中にある演算インスタンスのうち，最小のものから順に実行をスケジュールする．

4. 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 が組み込まれ有効化された場合には，PostgreSQL の既存エンジン (クエリ実行器) 自体の挙動は保存されたまま，その処理性能が加速される．PgBooster が無効化された場合には，処理性能を含めその挙動は従来と同様となる．

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

[*2] 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上で，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，本節での議論を適用することが可能である．

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 はプランを受け付けると，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，さもなくば一切の処理を行わない．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一切変えることなく，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．即ち，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，PostgreSQL 本体に変更を 2 点加えているが，いずれも PostgreSQL 本来のクエリ処理挙動に影響を与えるものではない．具体的には，(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，また (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は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，という 2 点である．

最新バージョンの PostgreSQL [*3] は本体のみで約 230 万行のプログラムコードから成っている．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) のための書き換え 16 行および追加 2193 行，PgBooster 自体の新規コード 4371 行のみである．PostgreSQL 本体の変更は処理追加が殆どであり，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．

まとめると，本試作実装では設計レベルにおける論理的な挙動保全性，実装レベルでの既存エンジンと加速機構の論理的独立性，およびデータベース管理システム本体の更新に追随する容易性を考慮した上で開発され，その規模は総計でコード 7000 行以下，PostgreSQL 全体に対して 0.3%程度に抑えられている．

5. 評価実験

5.1 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 [*4] と 32 GB の主記憶を搭載し，64bit 版 RedHat Enterprise Linux Server 5.8[*5] が OS として動作する．ディスクストレージは 160 台の 450GB 15,000rpm FC HDD と，8GB のキャッシュメモリをもつディスクシステム IBM DS5300 から成る．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 4Gbps 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 によってフォーマットしてデータベース領域とした．

評価に用いたデータベースは，意思決定支援系ベンチマークの業界標準である TPC-H ベンチマーク [8] を，Scale Factor = 1000 で初期化したものであり，PostgreSQL ロード後の容量は約 2.5TB である．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．また PostgreSQL のバッファプール容量は 256MB 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．

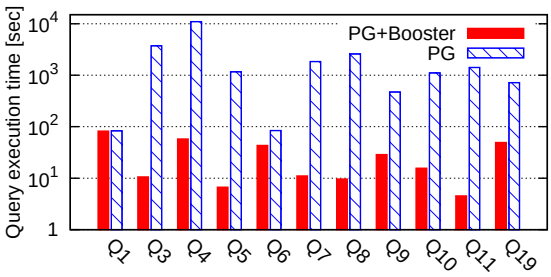


図 6 TPC-H 類似クエリ実行時間
Fig. 6 Execution time of modified TPC-H queries

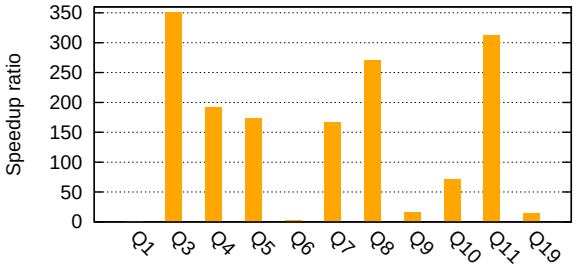


図 7 PG+Booster による TPC-H 類似クエリ加速率
Fig. 7 PG+Booster speedup ratio of modified TPC-H queries

5.2 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 11 クエリの選択率を小さく設定した上で，PG および PG+Booster における実行時間を測定した．

[*3] 2013/9/5 時点の最新バージョンは PostgreSQL 9.2.4
[*4] 1 プロセッサあたり 6 物理コアを有する Intel Xeon X7460 2.66GHz を 4 プロセッサ搭載
[*5] OS カーネルは Linux 2.6.18-308.el5

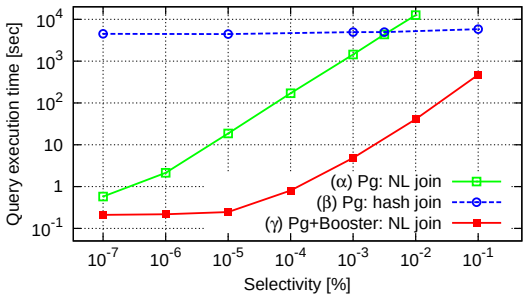


図 8 各実行方法における選択率とクエリ実行時間の関係
Fig. 8 Selectivity v.s. query execution time for each execution method

それぞれのクエリ実行時間を**図 6** に，また PG に対する PG+Booster の加速率を**図 7** 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 の入出力帯域を殆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，先読み効果によって PG でも入出力帯域を活用できているものであり，いずれの場合も 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．

5.3 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 [*6]．実験に際しては，Q3 の選択条件を調整すること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 8** 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．

[*6] 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

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．

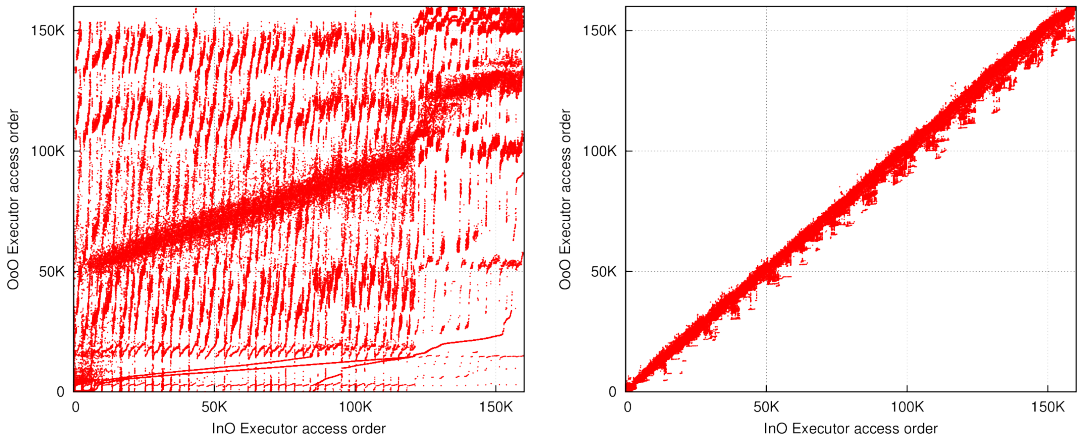
5.4 アウトオブオーダ型クエリ実行の協調的制御アルゴリズム評価

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 9** に示す．図 9 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実行器が全く同じ順番でデータベースページアクセスを行うと全ての点は直線 $x = y$ 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 9(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，殆ど相関性が見いだせないことがわかる．一方でコーディネータを有効化した場合には，図 9(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わかる．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．

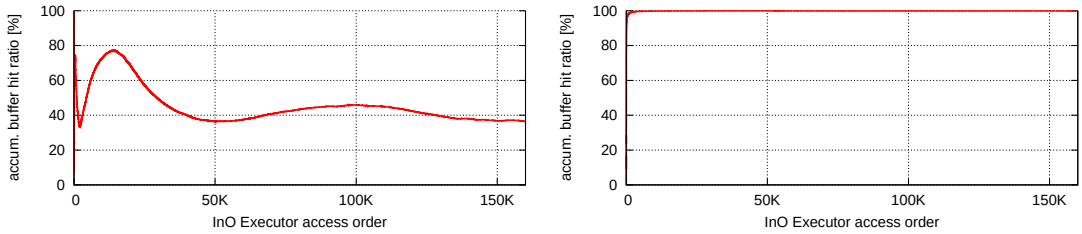
また，加速機構がインオーダ型クエリ実行器にもたらすページ供給作用を示したグラフが**図 10** である．x 軸は，インオーダ型クエリ実行器における各データベースページアクセスの発行順序を表し，y 軸は当該データベースページアクセスまでのインオーダ型クエリ実行器の累積バッファヒット率を表す．(a) のコーディネータ無効状態では，実行開始直後はカタログページアクセスによってヒット率が 100%となるが，テーブルアクセスが始まると累積ヒット率は 33%程度まで一旦低下した．その後，無秩序ながらもアウトオブオーダ型クエリ実行器によってバッファプールにページが先行的に読み込まれるため，約 13,000 回目前後の



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

図 9　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. 9　Temporal correlation of access patterns of In-order and Out-of-Order executor



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

図 10　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. 10　Accumulative buffer hit ratio of In-order executor over query execution

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 (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わかる．

6. 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 [9][10]，アクセス履歴からの予測による先読み [11]，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 [12][13]，データブロック間の相関性マイニングに基づく先読み [14] 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 [15]，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 [16][17] 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 [18][19] 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の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もつ提案手法は新規な取り組みである．

7. おわりに

本論文では，既存データベースエンジンの挙動を保全したまま，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挙動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．2.5TB の TPC-H ベンチマークを用いて 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・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 TPC-H ベンチマークのみではなく，より現実のデータに近い性質をもつデータセット・クエリにおいても提案手法の有効性を検証すると共に，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の高精度化を推し進めたい．

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, *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, Washington, DC, USA, IEEE Computer Society, pp. 175–185 (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] Transaction Processing Performance Council: `http://www.tpc.org/`.

[9] Feiertag, R. J. and Organick, E. I.: The Multics Input/Output system, *Proceedings of the third ACM symposium on Operating systems principles*, SOSP '71, New York, NY, USA, ACM, pp. 35–41 (1971).

[10] Smith, A. J.: Sequentiality and prefetching in database systems, *ACM Trans. Database Syst.*, Vol. 3, No. 3, pp. 223–247 (1978).

[11] Palmer, M. and Zdonik, S. B.: Fido: A Cache That Learns to Fetch, *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., pp. 255–264 (1991).

[12] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D. and Zelenka, J.: Informed prefetching and caching, *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, New York, NY, USA, ACM, pp. 79–95 (1995).

[13] Mowry, T. C., Demke, A. K. and Krieger, O.: Automatic compiler-inserted I/O prefetching for out-of-core applications, *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, New York, NY, USA, ACM, pp. 3–17 (1996).

[14] Li, Z., Chen, Z., Srinivasan, S. M. and Zhou, Y.: C-Miner: Mining Block Correlations in Storage Systems, *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, Berkeley, CA, USA, USENIX Association, pp. 173–186 (2004).

[15] Oracle Corporation: Performance and Scalability in DSS Environment with Oracle9i (2001).

[16] 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).

[17] 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).

[18] Ganusov, I. and Burtscher, M.: Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors, *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, Washington, DC, USA, IEEE Computer Society, pp. 350–360 (2005).

[19] Zhou, H.: Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window, *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, Washington, DC, USA, IEEE Computer Society, pp. 231–242 (2005).