博士論文

同期・非同期協調制御機構を備えた アウトオブオーダ型データベースエンジンに関する研究 (A Study on Out-of-Order Database Engine with a Coordination Mechanism between Synchronous Execution and Asynchronous Execution)

早水 悠登

目次

図目次

表目次

第 1 章

序論

1.1 はじめに

1.1.1 クエリ実行の同期・非同期連携によるデータベースエンジン高速化

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 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の増加傾向は明らかと言えよう．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．このように，計算機システムの有する処理帯域は着実に向上する一方，単位オペレーションあたりの実行遅延の改善率は芳しいとは言えない状況にある．プロセッサ動作周波数向上によるクロック遅延短縮は 2005 年前後から急速に停滞しており [2]，磁気ディスクドライブのアクセス遅延削減は 2000 年以降年率 5% 程度である [3]．またメモリモジュールやネットワークアクセス遅延の改善においても同様の傾向がみられる [4]．すなわち，多重的にオペレーションを駆動することによって，ハードウェアの有する帯域性能を積極的に活用するソフトウェアアーキテクチャがより一層重要となることを意味している．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 [5]，計算機システムのデータ管理の中核を担うデータベースシステムにおいては，とりわけハードウェア性能の活用は重要な課題である．

このような観点から，データベースシステムにおけるクエリ処理の中心的コンポーネントであるデータベースエンジンのアーキテクチャについて検討したい．Ingres [6] や

System-R [7] 等の初期のエンジン実装以来，多くのデータベースエンジンは木構造で表現されるクエリ実行プランを生成し，クエリ実行プランの各演算子ノードを逐次的にたどりながら実行し，クエリ実行をするという実行方式を採用してきた．演算子実行に際してデータベースからデータを取得する必要が生じると，データが格納されているストレージに対して同期的に入出力要求を発行し，入出力の完了を待ってから当該演算を実行する．このようなデータベースエンジンを，本論文では**インオーダ型データベースエンジン**と称することとする．

インオーダ型データベースエンジンにおいては，原則として入出力・演算処理は同期的に行われるため，各々の処理が時間的に重複することはないか，あったとしてもその機会は限定的である．初期の関係データベースエンジン実装が行われた 1970 年代には，計算機システムにおける CPU の並列演算性能や入出力帯域は限られており，また実行状態管理に利用可能な主記憶領域は極めて小容量であったため，簡潔に動作するインオーダ型データベースエンジンは妥当な設計であったといえよう．しかし，今日の計算機システムを構成する CPU は多数のプロセッサコアを有するに至り，ストレージは高密度化・広帯域化し，主記憶も大容量化を続けている．これらの処理性能を十分に活用するためには，処理の多重的実行による帯域活用が必須であるが，インオーダ型データベースエンジンの実行方式は多重性を持たないため，その性能活用は本質的に困難である．パーティション分割や演算のパイプライン化による並列性に着目して性能向上をはかる並列クエリ処理 [8–10] や，演算のベクトル化によるデータ並列性の活用 [11–14] などの取り組みはみられるものの，これらはクエリ処理における特定処理を予め定められた範囲で並列化する技術であり，それぞれの処理単位においては従来通りインオーダ型クエリ実行がなされるものである．

これに対し，クエリ実行方式を根本的に非同期化するアプローチとして，喜連川らは**アウトオブオーダ型データベースエンジン (OoODE)** [15] なるデータベースエンジンアーキテクチャを提案している．アウトオブオーダ型データベースエンジンはクエリ実行を動的にタスク分解することで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当該データに対する並列演算実行を駆動するという，アウトオブオーダ型なるクエリ実行方式によってクエリを実行する．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アドホックな分析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 [16, 17].

ここで，実際のデータベースシステムをアウトオブオーダ型データベースエンジンに基

づいて構築し，その高速性を利用可能とすることに関し，データベースシステムという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な挙動を前提として構築されている．データベースエンジンの非同期化を実現する 1 つのアプローチには，データベースエンジン全体の再設計があげられる．このアプローチはアウトオブオーダ型クエリ実行の高速性を最大限に活用可能である一方で，その実現は必ずしも容易ではない [18]．これに対して，本論文ではデータベースエンジン全体の再設計によらない，より簡便なアプローチを模索したい．従前のデータベースエンジンおよび周辺ソフトウェアの大幅な変更を伴うこと無く，アウトオブオーダ型データベースエンジンの高速性を取り込むことができれば，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，インオーダ型データベースエンジンにおける同期的なクエリ処理のうち，性能利得が大きい部分のみを非同期化し，同期的クエリ処理と非同期的クエリ処理が連携するという，同期・非同期連携クエリ処理を提案する．これにより，従前のインオーダ型データベースエンジンを構成する大部分のコンポーネントや，これに依拠する既存のソフトウェア資産を置き換えることなく，アウトオブオーダ型データベースエンジンの高速性を活用したクエリ処理を実現する．この際，クエリ処理の非同期化された範囲においては，同期的なクエリ処理とは異なる順序で入出力や演算実行が行われる可能性があるため，同期的処理と非同期的処理の連携界面においては，クエリ処理結果の整合性を取るために同期・非同期の協調制御を行う必要がある．本論文では，同期・非同期連携クエリ処理を実現するアプローチとして，（1）入出力処理を非同期化することで入出力帯域を活用するアプローチ，（2）入出力処理およびデータベース演算の任意部分を非同期化することで，入出力帯域と複数プロセッサコアの並列演算能力を活用するアプローチを提示し，それぞれについて同期・非同期協調制御を行う手法を示すことで，クエリ処理における非同期化領域を柔軟に設定し，データベースエンジン設計を行うことが可能であることを示す．

1.1.2　データベースエンジン加速機構

クエリ処理における入出力を非同期化するアプローチとして，本論文ではデータベースエンジン加速機構なるモジュールとして，アウトオブオーダ型データベースエンジンと同期・非同期協調制御機構をパッケージングし，既存のインオーダ型データベースエンジン

に組み込む手法を提案する．データベースエンジン加速機構は，データベースエンジンが入出力キャッシュ領域として有することが一般的なバッファプール領域に着目し，インオーダ型データベースエンジンが同期的入出力を行うデータを，先行的な非同期入出力発行によりバッファプールへと読み込む．これによりインオーダ型データベースエンジンは，その実行論理を一切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からのデータ供給を，バッファヒット率向上という形で享受することができ，入出力遅延が大幅に削減されることが期待される．とりわけ選択性を有する分析クエリ実行のように，入出力遅延によってインオーダ型データベースエンジンの実行速度が極度に律速されるようなクエリに関して，飛躍的な性能向上が期待される．提案するデータベースエンジン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，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．

提案手法の実現性，およびその性能向上効果を確認するべく，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 を対象として開発したデータベースエンジン加速機構の試作実装 PgBooster を示し，160 台のディスクドライブを有するミッドレンジ級データベースサーバを用いて評価実験を行い，その有効性を明らかにする．また，入出力性能の向上のみならず，プロセッサの並列演算性能活用を可能とするデータベースエンジン加速機構の設計についても検討を行い，初期性能評価実験を通してその潜在的な性能利得を示すことを試みる．

さらに，データベースシステムにおけるクエリ最適化に関して，データベースエンジン加速機構を用いたクエリ実行のコストモデルを提案し，評価実験によりクエリ実行プラン選択の精度を評価する．

1.1.3　演算処理の並列化機能を有するデータベースエンジン加速機構

クエリ処理の入出力および部分的なデータベース演算の非同期化を行うアプローチとして，本論文ではデータベースエンジン加速機構を更に拡張し，データベース演算の並列実行結果を供給することで，複数プロセッサコア活用を可能とするデータベースエンジン加速機構を提案する．当該加速機構は，アウトオブオーダ型データベースエンジンにおけるクエリ処理において，データベース演算の結果得られたタプルデータをインオーダ型デー

タベースエンジンへと供給することで，インオーダ型データベースエンジンにおける逐次演算実行の負荷を削減する．これにより，システム全体としてのプロセッサ利用効率が向上し，更なる高速化が可能であることを示す．本論文では，複数プロセッサコア活用型のデータベースエンジン加速機構の試作実装 PgBoosterMC を開発し，TPC-H データセットおよび人流データセットを用いた評価実験によってその有効性を明らかにする．

1.1.4 アウトオブオーダ型データベースエンジンにおける複数クエリ実行間の動的資源調停

アウトオブオーダ型データベースエンジンは，クエリ処理の動的なタスク分解によって実行並列性を最大限に抽出することにより，高多重の非同期入出力発行と演算用スレッド生成を行うことで，入出力帯域と複数プロセッサコアの利用効率を高める，特に選択性を有するクエリ処理を大幅に高速化する．インオーダ型データベースエンジンと比べて入出力帯域と複数プロセッサコアを積極的に活用することから，利用可能なこれらの資源量がクエリ処理性能に与える影響はより大きくなることが想定される．

本論文では，アウトオブオーダ型データベースエンジンにおいて，複数のクエリ処理を同時に実行している際に，各クエリに割り当て可能な資源量を実行時に調整する動的資源調停手法を提案する．本論文では，特にシステムの入出力帯域がボトルネックとなる状況を想定し，実行中のクエリがそれぞれ優先度に応じた入出力性能を確保可能となるよう，資源調停を行う手法を示す．これにより，豊富な入出力帯域を有するシステムにおいて，優先度に応じた柔軟な性能要求に応えることが可能となる．本論文では PgBooster において提案手法の試作実装を行い，TPC-H データセットを用いた評価実験を行うことで提案手法の有効性を示す．

1.2 本論文の構成

本論文の構成は次の通りである．

第 2 章では，本論文に関連する研究についてまとめる．まず，クエリ処理の多重化によるハードウェア性能活用に関する研究として，並列クエリ処理分野における取り組みと，アウトオブオーダ型データベースエンジンについて述べる．次に，入出力の先読みによるクエリ実行性能向上に関する研究をまとめ，その後に複数クエリ最適化に関する研究をまとめる．

第 3 章では，既存のデータベースエンジン実装と連携することで，そのクエリ実行を大

幅に高速化するデータベースエンジン加速機構を提案する．まず，既存のデータベースエンジンにおいて広く採用される同期的実行と，アウトオブオーダ型データベースエンジンの非同期的実行の連携によるクエリ実行高速化の可能性について論じ，これを実現するための指針として主記憶のバッファプールを介してアウトオブオーダ型データベースエンジンが高速にデータ供給を実施するという方策を示す．当該指針に基づき，同期・非同期連携に必要なコンポーネントとアウトオブオーダ型データベースエンジンを組込み可能なモジュールとしてパッケージングするデータベースエンジン加速機構を提案する．その後に，データベースエンジン加速機構が有効にインオーダ型データベースエンジンと連携するためのアウトオブオーダ型クエリ実行協調制御アルゴリズムについて，その詳細を論じる．また，データベースエンジン加速機構はバッファプールを介することでクエリ実行高速化を実現するため，バッファ管理におけるページ置換ポリシについて考察する．

第 4 章では，データベースエンジン加速機構の試作実装を示し，その性能評価を行う．まず，データベースエンジン加速機構の試作機として，オープンソースデータベース管理システム PostgreSQL を対象として開発を行った PgBooster の設計と実装を示す．そして，TPC-H データセットを用いた性能評価試験，および協調制御アルゴリズムの挙動検証により，データベースエンジン加速機構の有効性と性能向上効果を示す．さらに GPS データセットを用いた性能評価実験を行い，実データに対してもデータベースエンジン加速機構が有効に機能し，性能向上をもたらすことを示す．

第 5 章では，アウトオブオーダ型データベースエンジンにおいて複数のクエリを同時実行する際の実行時資源割り当てについて議論する．まずアウトオブオーダ型データベースエンジンにおけるクエリ実行モデルを構築することで，クエリ実行のスループット性能に関して定量的に考察し，複数クエリ実行間で優先度に基づく動的資源調停を行う手法を提案する．そして，TPC-H データセットを用いた評価実験により，提案方式により動的資源調停を行うことで，各クエリ実行が優先度に応じたスループット性能を確保することが可能であることを示す．

第 6 章では，データベースエンジン加速機構において複数プロセッサコアの並列演算性能を活用するための設計について論じ，試作実装によってその初期性能評価を行う．第 3 章において示したデータベースエンジン加速機構は入出力処理のみを高速化するものであるため，まず CPU 演算の逐次実行における性能制約について考察する．その後にインオーダ型データベースエンジンに対して演算結果のタプルを供給することで，複数プロセッサコアの並列演算性能を活用可能とする設計を提案する．そして当該設計に基づく試作実装 PgBoosterMC を示し，TPC-H データセットおよび人流データセットを用いた評価実験により，CPU 演算負荷が高いクエリ実行における提案手法の有効性を示す．

第 7 章では，データベースエンジン加速機構を有効に活用するためのクエリ最適化方式を提案する．クエリ最適化においては，適切なプラン選択を行うために各クエリ実行プランのコストモデルが必要とされるため，まずデータベースエンジン加速機構を用いる場合のコストモデルを提案する．当該コストモデルは実行環境の入出力性能特性に依存するパラメタを持つため，コストモデルを自動較正する手法を提案する．そして，TPC-H データセットを用いてコストモデル自動較正を実施し，複数のクエリにおいてクエリ実行プラン選択の精度評価を行う．

最後に，第 8 章で本論文をまとめるとともに，今後の研究課題について述べる．

第 2 章

関連研究

2.1 クエリ処理の多重化によるハードウェア性能活用に関する研究

計算機プログラムにおいて，目的とする出力結果を得るための処理の実行方式は，大きく同期的実行と非同期的実行の 2 つに分類される．同期的とは，ある処理単位の実行を開始すると，その処理の完了を待ってから後続の処理に制御を移す実行方式である．一方，非同期的実行とは，ある処理の実行を開始した後に，当該処理の完了を待たずに他の処理の実行を開始する実行方式である．

同期実行はその動作モデルが簡潔で理解しやすく，また処理の実行状態の遷移が直線的で管理が容易であることから，幅広い場面において標準的な実行方式として用いられている．これはデータベースシステムにおいてクエリ実行の中核を担うデータベースエンジンにおいてもあてはまる．Ingres [6] や System-R [7] に端を発する初期の関係データベースエンジンの実装から始まり，今日の商用実装やオープンソース実装に至るまで，多くのデータベースエンジンでは同期的に入出力を行うことでストレージに格納されるデータを取得し，当該データに対する計算処理を実行する，という手順を逐次繰り返すインオーダ型クエリ実行方式を基本方式として採用している [19–24].

同期的実行においては，原則として複数の処理が時間的にオーバラップして実行されることがないため，計算機システムの有する演算・入出力資源を高効率に活用するという観点からは望ましくない．そのため，インオーダ型クエリ実行方式を基調としながらも，クエリ実行におけるハードウェア資源活用効率を向上させるため，従前より様々な取り組みがなされてきた．

なかでも並列クエリ処理は最も広範な取り組みがなされてきた分野の 1 つである．こ

れはデータベースの分割によるパーティション並列性や，クエリ処理を構成する演算子間のパイプライン並列性に着目し，これらから抽出される並列実行可能な処理を多重的に展開するものである [8–10]．並列クエリ処理においては，データベースのパーティショニング方式 [25–33] や，並列結合方式 [31, 34–41] など，実に多くの取り組みが行われてきた．またクエリ実行における演算処理をベクトル化することで，データレベル並列性を活用しクエリ実行の高速化を図る取り組みもみられる [11–14]．

一方，非同期入出力を用いたクエリ実行の多重化によるハードウェア性能活用に関する取組みは多くなく，これまでは線形走査や先読みの非同期入出力を用いた効率化 [42–44] や，チェックポイント時のダーティページ書き出し [44] に留まっていた．これに対し，喜連川らは従来のインオーダ型データベースエンジンにおける実行方式を根本的に転換し，クエリ実行を動的タスク分解することで高多重な非同期化入出力発行・演算実行を行い，ストレージの入出力帯域およびマルチコアプロセッサ性能を高効率に活用可能とするアウトオブオーダ型データベースエンジンを提案している [15]．アウトオブオーダ型データベースエンジンは，特に選択性を有するクエリ実行において従来のインオーダ型クエリ実行に対して大幅な高速性を発揮することが確認されている [45]．また山田らは多数のノードから構成される並列データ処理系に対してアウトオブオーダ型の実行方式を適用し，ストレージ入出力ならびにネットワーク入出力の非同期化を行うことで，大幅な高速化が実現可能であることを示した [46]．

2.1.1 本研究との関連

本研究は，インオーダ型データベースエンジンにおける同期的クエリ実行と，アウトオブオーダ型データベースエンジンにおける非同期的クエリ実行が連携することで，既存のインオーダ型データベースエンジン実装の抜本的な変更を行うことなく，そのクエリ実行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ことを狙う取り組みである．すなわち，その本質は異なる実行方式で等価な処理を行う 2 つのデータベースエンジンが協調的に動作するための制御手法にある．その意味において，データベースエンジン自体のクエリ処理に関する方法を扱う並列クエリ処理や，あるいはアウトオブオーダ型データベースエンジンに関する研究と本研究は直交するものである．

2.2　入出力の先読みに関する研究

入出力の先読みは入出力遅延を隠ぺいし，入出力帯域の活用効率を向上させる有効な手段として，計算機システムの歴史において初期の頃から今日に至るまで活発な研究が続けられている分野である．最も基本的なシーケンシャル先読み [47–51] をはじめとして，入出力アクセスの規則性に基づく先読み [52–57]．アプリケーションからのヒント情報を用いた先読みや [58–61]，データブロック間の相関性マイニングを用いる先読み [62] など，様々な方法が提案されてきた．先読みが誤っていた場合には性能に対するペナルティが非常に大きいため [55, 63] 多くの先読み手法においては不要な入出力を発行しないよう保守的なアプローチがとられてきたが，ストレージやプロセッサ技術潮流によって計算機システムにおける性能バランスは大きく変化し，より積極的な先読みを行うことによる利得が増加する傾向にあるとの考察もみられる [64]．

データベースシステム特有の先読みに関する取り組みもみられ，例えば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 [65] や，索引検索をストレージコントローラにおいて多段展開する方法 [66, 67] などがある．またアクセスパターンに対してデータ配置を最適化する取り組み [68, 69] なども行われてきた．

2.2.1　本研究との関連

本研究において提案するデータベースエンジン加速機構は，インオーダ型データベースエンジンが必要とするデータに対して事前に入出力を発行し，バッファプールへと読み込みを行い，バッファヒット率を大幅に向上させる手法である．投機的に入出力を行うことでバッファヒット率を向上させるという点においては既存の先読み手法と本研究の提案手法は共通する．その一方で，提案するデータベースエンジン加速機構はそれ単体として完備なデータベースエンジンの機能性を有しており，クエリ実行の過程において必要とされるデータのみに入出力の対象を限定することが可能である．すなわち，入出力対象をあくまで予測する先読み手法とは本質的に異なる．

2.3　複数クエリ最適化に関する研究

複数クエリ実行における最適化は，クエリ実行間で重複して実行される処理を集約して共通化するアプローチを基本とする取り組みが多い．特にデータベースシステムにおいて

具体的な処理方式はクエリ実行プランによって決定されるため，共通する結合演算に着目してグローバルなクエリ最適化を行う手法 [70–73] が提案されている．またクエリ実行時に共通する処理を動的に検出し集約する取り組みとして，Candea らはスター型スキーマにおけるファクト表を常に走査し続ける CJoin を提案した [74]．また Harizopoulos らはクエリ実行を複数のパイプラインステージへと分割し，各パイプラインステージで共通する処理を検出し集約可能な Qpipe なるアーキテクチャを提案した [75].

複数クエリ実行時のスケジューリングに関してもいくつかの取り組みがみられる．締め切り要求が異なるクエリ実行のスケジューリングに関して，Carey らは要求レベルに応じてディスクアクセスのスケジューリングを行うことで，高優先度トランザクションの応答時間が低優先度のトランザクションから受ける影響を抑制可能であることを示した [76]. また Pang らはリアルタイム性の要求されるデータベースシステムにおいて，クエリ実行に要求されるメモリ消費量を複数のクラスに分類することで，クエリ実行が締め切り要求に遅れる割合を低減する手法を提案した [77]．また商用データベースシステムにおける QoS を実現する取り組みとして，Oracle DRM [78] や DB2 Query Patroller [79] といった実装がみられるが，これらは基本的に CPU 演算資源の動的なスケーリングを行うにとどまっている．

2.3.1 本研究との関連

本研究では，アウトオブオーダ型データベースエンジンにおいて複数のクエリが同時実行されている場合に，各クエリに設定された連続値の優先度に従って弾力的に各クエリのスループットを調整する手法を提案する．

共通処理を集約することによる複数クエリ実行の最適化に関しては，本提案手法と直交する技術である．

またクエリの要求レベルに応じてスケジューリングを行うという点においては [76, 77] と共通するものの，これらの取り組みはオンライントランザクション処理におけるクエリを主な対象としており，また 4 クラス程度の優先度にクエリを分類し，締め切り要求が満たされるようスケジューリングを行う方法であるのに対し，本提案手法は大規模なデータアクセスを伴う分析的クエリを対象としており，クエリに設定された連続値の優先度に比例する形でスループットを割り当てるという点において異なる．また本提案手法では入出力スループットにおいても優先度に従ったスケーリングが可能であるのに対し，[78, 79] は CPU 演算資源における QoS を提供するに留まっている．

第 3 章

同期・非同期協調制御機構を備えたアウトオブオーダ型データベースエンジン

データベースエンジンはその中心的機能のみでも 100 万行から 1000 万行規模のコードから構成されることが珍しくない．また大量のデータを管理し，適切に利用可能するというその役割から，データを利用する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．今日におけるデータベースエンジン実装の多くはインオーダ型による同期的なクエリ実行方式を基本として構築されているが，これを抜本的に非同期化し，アウトオブオーダ型へと転換することは必ずしも容易ではない．そこで本論文では，データベースエンジンの中でも非同期化による性能利得が大きい範囲のみを非同期化し，同期的クエリ処理と非同期的クエリ処理を連携させる，同期・非同期連携クエリ処理を提唱する．本章では，同期・非同期連携に際して（1）入出力の非同期化を行うアプローチ，および（2）入出力と部分的なデータベース演算の非同期化を行うアプローチを提示し，その得失について議論する．そして，アウトオブオーダ型データベースエンジンを既存のインオーダ型データベースエンジンへプラグインすることで，インオーダ型データベースエンジンをほとんど変更することなく（1）の入出力非同期化を実現可能であるデータベースエンジン加速機構を提案し，その要となる同期・非同期協調制御アルゴリズムについて議論する．

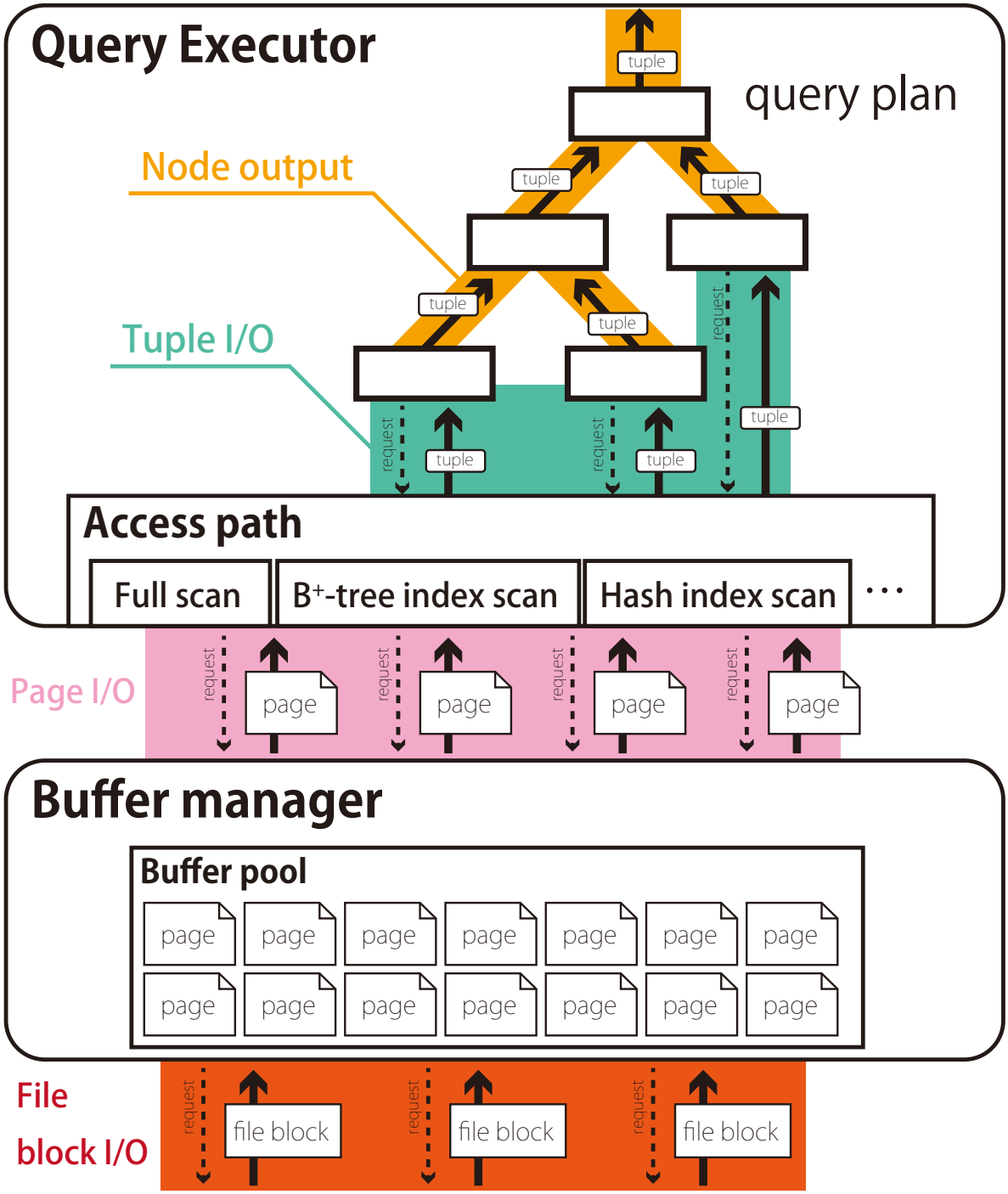


図 3.1　データベースエンジンコンポーネントとデータ処理単位

3.1　データベースエンジンにおける同期・非同期連携

3.1.1　同期・非同期連携の基本的着想

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（**図 3.1**）.

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り，クエリ最適化器によって生成されたクエリ実行プランに従って動作する．クエリ実行プランは木構造により表現され，各プランノードは実行すべき処理（リレーション走査，結合，ソート等）を表す．クエリ実行器におけるデータ処理単位は**タプル**であり，プランノードは下位ノードの処理結果であるタプルを入力として受け取り，その処理結果を上位ノードへと出力する．そして，最上位のプランノードが出力するタプル列がクエリ実行結果としてクエリ要求発行元へと出力される．処理の実行過程においてストレージ上に格納されるタプルにアクセスする必要が生じると，クエリ実行器はアクセスパスを通してタプルが格納されるページ番号を特定し，当該ページに対するアクセス要求をバッファマネージャへと発行することでタプル入出力を行う．

バッファマネージャはクエリ実行器からページ要求を受付けると，ページ番号から当該ページが格納されているファイルとファイル内ブロックアドレスを特定し，ストレージに対して当該ブロックに対する入出力要求を発行する．当該ブロックの入出力が完了すると，バッファマネージャはクエリ実行器に対して要求されたページデータを返すことでその処理を完了する．また一般にバッファマネージャは主記憶上にキャッシュ空間を保持することで，ページ要求の度にストレージに対して入出力要求を発行することを抑制する．このキャッシュ領域はバッファプールと呼ばれ，クエリ実行器が求めるページがバッファプールに格納されているときには，ページ要求は**バッファヒット**するといい，ストレージに対する入出力を伴うことなく処理することができる．逆にページがバッファプールに格納されていない場合，ページ要求は**バッファミス**するといい，ストレージに対して入出力を発行することで当該ページを取得し，バッファプールに格納した上でクエリ実行器へとページデータが返される．ストレージ入出力は極めて高価なオペレーションであるため [4]，バッファヒット率はクエリの実行速度に顕著な影響を与える．データベースエンジンにおけるバッファヒット率を向上に関しては，バッファプールの管理ポリシ [48,80–82] や先読み [49,50,58,83] 等，様々な取り組みが行われている [84].

インオーダ型データベースエンジンにおいて，クエリ実行は同期的に進行する（**図 3.2**）.

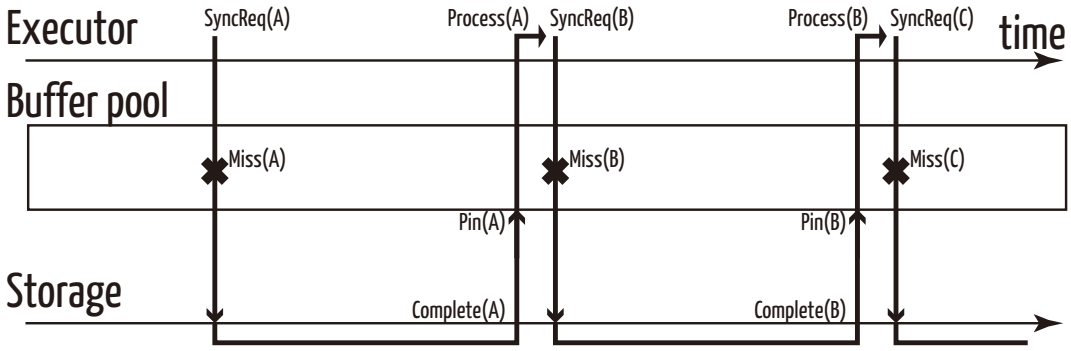


図 3.2　インオーダ型クエリ実行

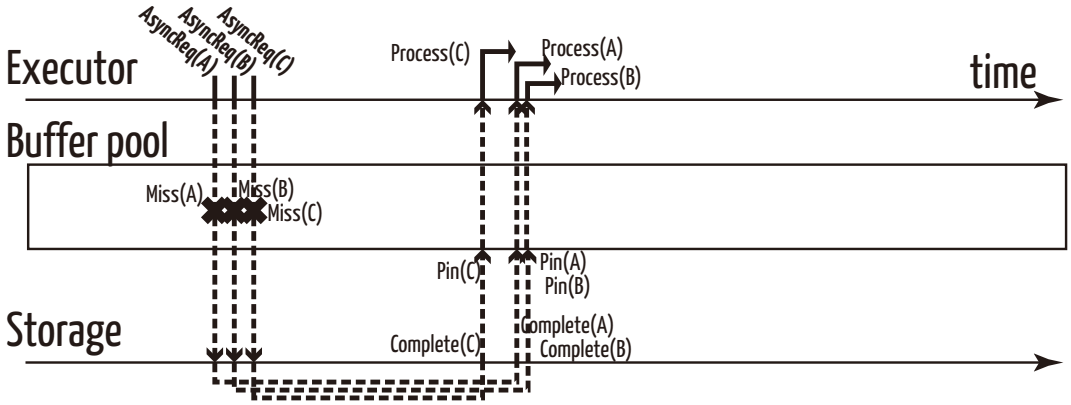


図 3.3　アウトオブオーダ型クエリ実行

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，その取得完了を待ってからページに対する演算処理を実行する．こうして 1 つの演算が完了すると，次なる演算へと制御が移る．同期的にクエリ実行をする場合，各々のページに関する入出力処理や，CPU 演算処理は時間的にオーバラップすることが無いか，あったとしても極めてその機会は限定的である．即ち，インオーダ型クエリ実行においては，本質的にシステムの有する入出力・演算資源を多重的に利用することが困難である．

これに対し，アウトオブオーダ型データベースエンジンにおけるクエリ実行は非同期的に進行する（**図 3.3**）．クエリ実行器は実行することが判明している演算に関して，当該演算に必要なページに対し非同期にページ要求を多重的に発行する．即ち，複数の演算が各々のページ要求の完了を待つ状態となり，バッファマネージャはページ要求を受けて非同期入出力をストレージに対して多重的に発行する．そして，ページ要求完了を契機として演算の実行が随時駆動される．そのため，クエリ実行中は複数の入出力処理・CPU 演算処理がオーバラップして多重的に実行され，システムの有する入出力帯域および並列演

算性能を高効率に活用することが可能となる．特に中程度の選択性を有するクエリ実行においては，インオーダ型データベースエンジンに対して高い性能を示すことが知られている [45].

現存するデータベースエンジン実装は，70 年代から開発が続けられているものも多く，そのコード規模は極めて大規模である [85–87]．多くはインオーダ型データベースエンジンとして実装されているが，根幹を成すクエリ実行方式をアウトオブオーダ型へと転換し再実装を行うこと，あるいは十分な互換性や機能性を有する新規実装を行うことは必ずしも容易ではない [88]．そこで，本論文ではより簡便なアプローチを模索したい．即ち，既存のインオーダ型データベースエンジン実装に対して，アウトオブオーダ型データベースエンジンの高速性を外部機能として取り込み，クエリ実行の高速化を実現する方策について検討を行う．

同期的なインオーダ型データベースエンジンと非同期的なアウトオブオーダ型データベースエンジンは根本的に実行方式を異にする．しかし，クエリ実行の結果として最終的に生成するタプル集合は等価であり，またクエリ実行プランに示される演算に従って途中結果のタプル生成やページ入出力を行うという構造は共通する．つまり，アウトオブオーダ型データベースエンジンが先行して非同期的にクエリ実行を進め，その過程で取得したページや，生成し得たタプルをインオーダ型データベースエンジンへと有効に注入することができれば，インオーダ型データベースエンジンはクエリ実行に要する入出力や CPU 演算を省略することが可能となる．アウトオブオーダ型データベースエンジンからのデータ注入が行われない場合，あるいは処理の省略に間に合わない場合には，インオーダ型データベースエンジンは単に自身の実行方式に則ってクエリ実行すればよい．このような**同期・非同期連携**により，既存のインオーダ型データベースエンジン実装を大きく変更することなく，アウトオブオーダ型データベースエンジンの高速性を取り込むことが可能となる．

同期・非同期連携を実現するためには，両データベースエンジンを接続するために次の 2 つのコンポーネントが必要である．

データ配送バッファ　アウトオブオーダ型データベースエンジンから注入されるデータを受け取り，インオーダ型データベースエンジンへと供給するためのバッファ領域．

タイミング制御モジュール　インオーダ型データベースエンジンにおける処理の進捗を把握し，直近にインオーダ型データベースエンジンが必要とするデータのみをデータ配送バッファへ投入するようアウトオブオーダ型データベースエンジンの動作を制御するモジュール．

アウトオブオーダ型データベースエンジンは非同期的にクエリ実行をするため，その内部における演算の実行順序は本質的に非決定的である．そのため，インオーダ型データベースエンジンがデータを処理する順序とは異なる順序でデータを注入することになるため，その順序の違いを吸収するためにデータを格納するためのバッファ領域が必要である．またアウトオブオーダ型データベースエンジンの高速性を活かすために，当該バッファ領域は高速にアクセスできる必要があるため，主記憶に十分収まる程度の有限容量バッファであることが妥当である．クエリ実行におけるデータフットプリントがデータ配送バッファ容量を大幅に上回る場合，インオーダ型データベースエンジンにおけるデータ処理の順序を全く考慮せずにアウトオブオーダ型データベースエンジンがデータを配送バッファへと投入すると，いたずらに配送バッファ領域を消費し，有効にインオーダ型データベースエンジンの高速化に寄与しない可能性が生じる．そこで，インオーダ型データベースエンジンにおけるクエリ実行の進捗を把握し，直近に必要とされるデータが優先的に投入されるよう，アウトオブオーダ型データベースエンジンの動作を制御するモジュールが必要となる．

3.1.2　同期・非同期連携の実現レイヤと得失

データベースエンジンにけるデータ処理の単位は図 3.1 に示すように，クエリ実行器層においてはタプルであり，バッファマネージャ層においてはページである．すなわち，同期・非同期連携によるインオーダ型データベースエンジンの高速化を実現する基本的なアプローチとして，タプル単位でデータを注入するアプローチと，ページ単位でデータを注入するアプローチの 2 つが考えられる．

タプル単位でのデータ注入を行うアプローチでは，インオーダ型データベースエンジンが 1 つタプルを受け取る度に，より下位層のクエリ実行プランにおける処理や入出力処理を全て省略することが可能である．すなわち，アウトオブオーダ型データベースエンジンにおける CPU 演算・入出力資源活用の利益を享受することが可能である．これを実現するためには，クエリ実行器におけるタプルデータの受け渡しが行われるクエリ実行プランノード間や，クエリ実行プランノードとアクセスパス間などの界面において，これから生成しようとするタプルがすでにデータ配送バッファに到着していないかを確認し，到着している場合にはそちらを選択するという論理を組み込む必要がある．また多くのインオーダ型データベースエンジンではイテレータモデル [89] を採用しており，クエリ実行プランノードやアクセスパスは内部に状態変数を有している．そのため，データ配送バッファからタプルを受け取る場合には，当該タプルの処理が完了したものとしてこれらの状態変

数を適切に更新する必要がある．通常のインオーダ型データベースエンジンにおいては，状態変数の更新とデータに対する演算，結果タプルの生成は一連の手続きとして記述されるものであるため，この中から状態変数の更新に関する手続きのみを分離して実行できるよう整理することが必要となる．このように，タプルを単位として同期・非同期連携を行う場合，データ配送バッファが機能するためにクエリ実行器における論理の一部追加・再構成が必要となる．これに加えて，有限のデータ配送バッファの下で有効にタプルを注入するためのタイミング制御モジュールを構築しなければならない．

一方，ページ単位でデータを注入するアプローチでは，バッファマネージャにより既にデータ配送バッファの機能は実現されている．ページ取得は，ページ識別子を引数としてバッファマネージャにページ要求を発行するというインターフェースを介して行われる．アウトオブオーダ型データベースエンジンが予め当該ページに対してページ要求を発行し，入出力を完了していれば，インオーダ型データベースエンジンのページ要求はバッファヒットするため，透過的にアウトオブオーダ型データベースエンジンからのデータ注入を受付けることができる．そのため，アウトオブオーダ型データベースエンジンにおけるページ要求のタイミング制御モジュールを構築可能であれば，それは即ち当該アプローチが実現可能であることを意味し，実現容易性においてタプル注入によるアプローチに対して優れている．ただし，ページ単位でデータ注入を行う場合には，クエリ実行器レイヤにおける CPU 演算は従前と同様逐次的に行われる．つまり，アウトオブオーダ型データベースエンジンの演算並列性は活用することができず，性能向上の理論的上限に関してはタプル注入によるアプローチには劣る．

このように，ページ単位でデータ注入を行うアプローチはソフトウェア設計における論理的簡潔性，実現容易性に優れているため，以降は当該アプローチを基軸としてその具体的実現方法を論じることとする．当該アプローチはアウトオブオーダ型データベースエンジンの演算並列性による性能向上という利益を享受することはできないものの，入出力が支配的なボトルネックであるクエリ実行においてはバッファヒット率の向上による大幅な性能向上が期待される．また，当該アプローチ実現に必要とされるアウトオブオーダ型データベースエンジンのタイミング制御モジュールの構築にあたっては，2 つのデータベースエンジンにおけるデータ処理順序を構造化し，協調動作させる方法論を確立する必要がある．これは同期・非同期連携において必須のものであるため，タプル単位のデータ注入によるアプローチに取り組む際にもその基盤として貢献するといえよう．

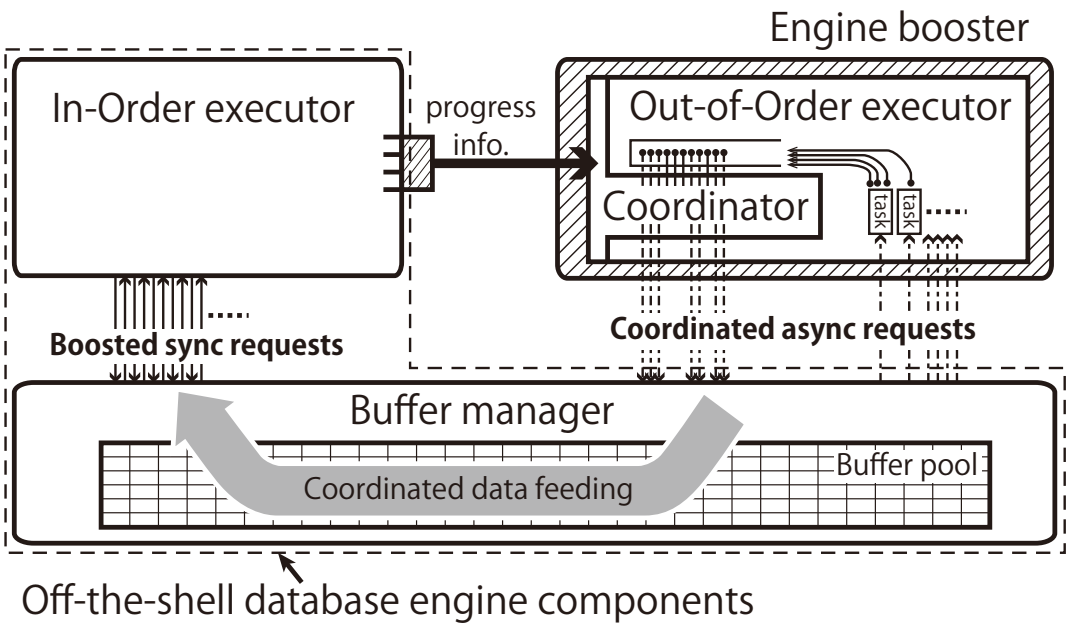


図 3.4　既存データベースエンジンと加速機構の概要

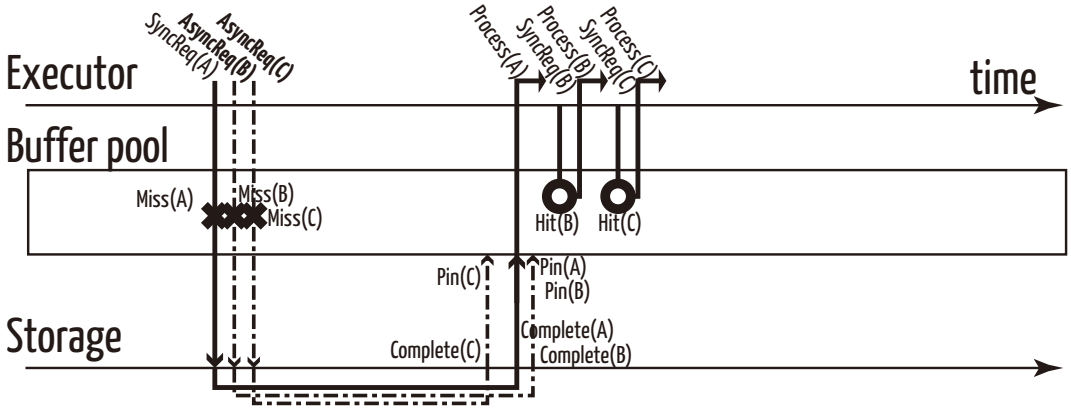


図 3.5　アウトオブオーダ型クエリ実行を伴うインオーダ型クエリ実行

3.2　データベースエンジン加速機構

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データベースエンジン加速機構を構成する手法を提案することで，ページ単位のデータ注入による同期・非同期連携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 3.4** 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．これにより，**図 3.5** に示すようにアウトオブオーダ型クエリ実行器によって先行的にページがバッファプールへと読み込まれ，インオーダ型クエリ実行器のバッファヒット率が大幅に向上し，その実行が高速化されることが期待される．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出来，アウトオブオーダ型クエリ実行器による更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，従来通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，第 3.3 節において議論する．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．第 3.4 節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．

3.3　同期・非同期協調制御アルゴリズム

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，当該ページはバッファプールから追い出されてしまう可能性が高い．つまり，アウトオブオーダ型クエリ実行器は，ある一定ステップ内にインオーダ型クエリ実行器が必要とするページに対象を限って先行的なページ読み込みを行うよう，アウトオブオーダ型クエリ実行コーディネータによる制御がなされる必要がある．この制御を実施するため，まずクエリ実行過程の構造を規定し，これに基づいてアウトオブオーダ型クエリ実行の制御方式を議論する．

3.3.1　クエリ実行過程の構造

一般にクエリ実行器は，クエリ最適化器によって生成された木構造状のクエリ実行プランに基づいてクエリ実行をする．各プランノードはプランノードの処理結果を入力として

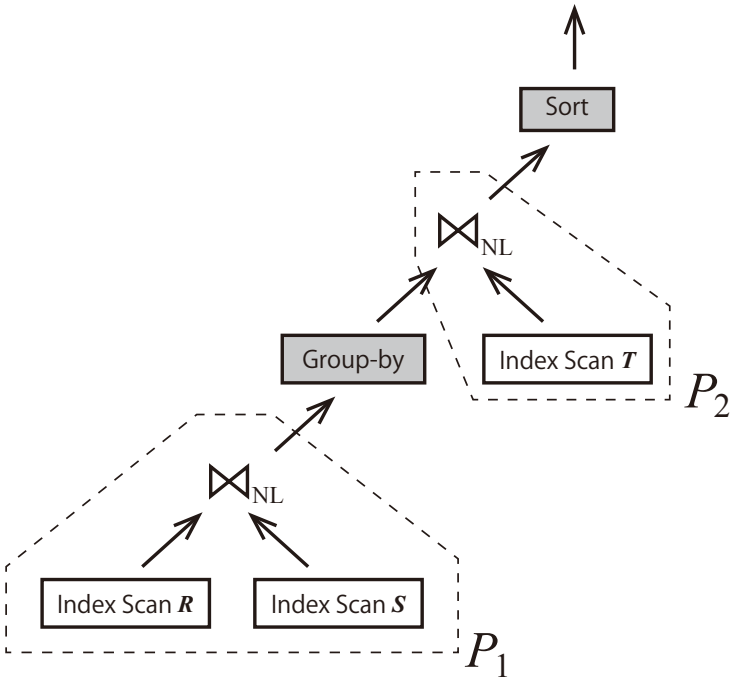


図 3.6　クエリ実行プランとパイプライン

受け取り，その処理結果を親ノードへと出力する．プランノードは処理の種類を表しており，テーブルの順次全件走査 (Seq Scan)，索引走査 (Index Scan)，フィルタ (Filter)，ネステッドループ結合 (NL Join)，ハッシュ結合 (Hash Join)，ソートマージ結合 (Sort-Merge Join)，グループ化 (Group-by)，ソート (Sort)，スカラ値計算 (Scalar) などが一般的に用いられるプランノードである．プランノードの中でも，ハッシュ結合やソートマージ結合，グループ化，ソート，スカラ値計算のように，少なくとも 1 つの入力を全て処理しないかぎり出力を生成しないノードを指して**ブロッキングノード**と称する．クエリ実行プランはブロッキングノードを境界として複数の**パイプライン**へと分割される．ただし，パイプラインはブロッキングノードを含まないものとする．**図 3.6** にクエリ実行プランと，ブロッキングノードによって分割されたパイプラインの例を示す．

インオーダ型クエリ実行器において，異なるパイプラインにあるプランノード同士は，最初の入力を受け取ってから最後の出力を生成するまでの期間が互いに重なり合うことはない．すなわち，ブロッキングオペレータの出力を仮想的なテーブルとみなせば，パイプライン $P_1, P_2, \cdots, P_N$ から構成されるクエリ実行プランを実行することは，$P_1, P_2, \cdots, P_N$ に相当する N 個のクエリ実行プランの直列化された実行と見なすことができる．つまり，単一パイプラインからなるクエリ実行プランに関してアウトオブオーダ

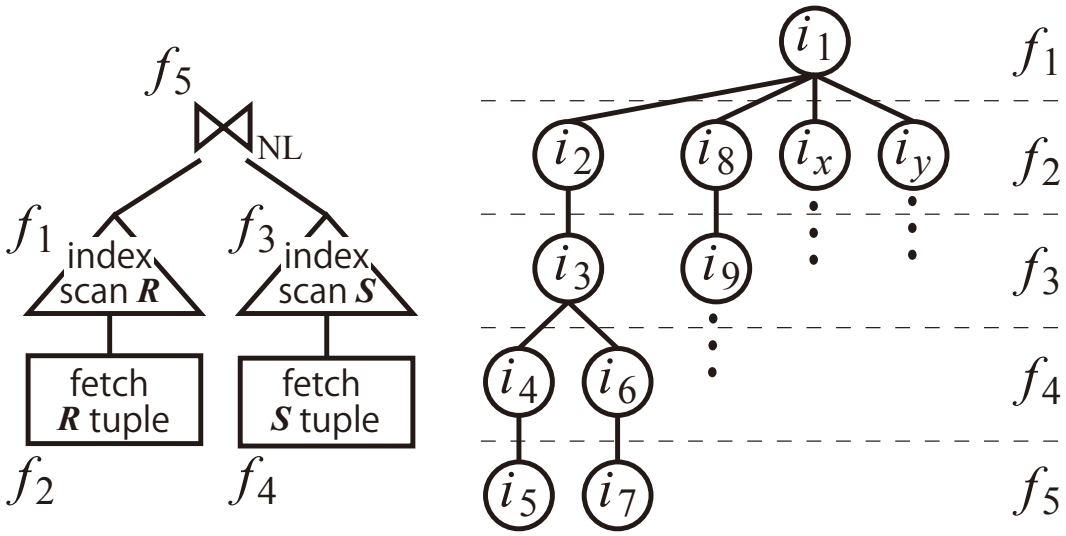


図 3.7　2 表の結合を行うクエリ実行プランと演算インスタンス木

型クエリ実行器の協調制御が実現可能であれば，多段パイプラインからなるクエリ実行プランに関しても同様であるといえる．以降は，簡単化のためクエリ実行プランは単一パイプラインによって構成されるもののみを対象として議論を進める．

クエリ実行器による実行過程において，要求されるページを決定するのは**データベース演算** f と，その演算 f を実行する**コンテクスト** c である．ただし，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．例えば B^+ 木索引から索引エントリを取得するというデータベース演算を考えると，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** $i = \langle f, c \rangle$ と呼ぶこととする．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．つまり，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 3.7** 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．当該プランは R 表の索引検索を行い，その検索結果を用いて S 表の索引検索を駆動し，R 表と S 表のタプル結合を行うため，まず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．この例で

は 4 件のエントリが該当するので，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成すものと考えることができる[*1]．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．例えば図 3.7 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．例えば図 3.7 においては，演算インスタンス i_6 は，$i_1 \to i_2$(1 番目の子) $\to i_3$(1 番目の子) $\to i_6$(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時には $\mathrm{route}(i) = \mathrm{route}(j)$ 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

[*1] 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．

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．

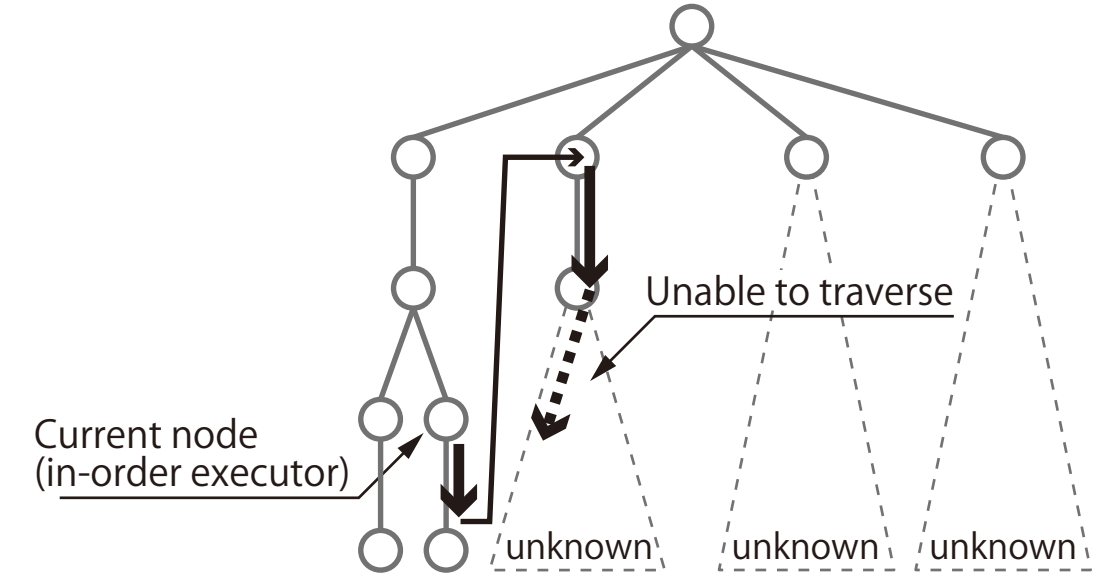
- $\mathrm{route}(i), \mathrm{route}(j)$ の各分岐で選択する子番号を根から順次照合してゆき，異なる子番号を選ぶ最初の分岐において，$\mathrm{route}(i)$ のほうが子番号の小さい子を選択する
- $\mathrm{route}(i) \neq \mathrm{route}(j)$ かつ，$\mathrm{route}(i)$ が $\mathrm{route}(j)$ に含まれる

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．

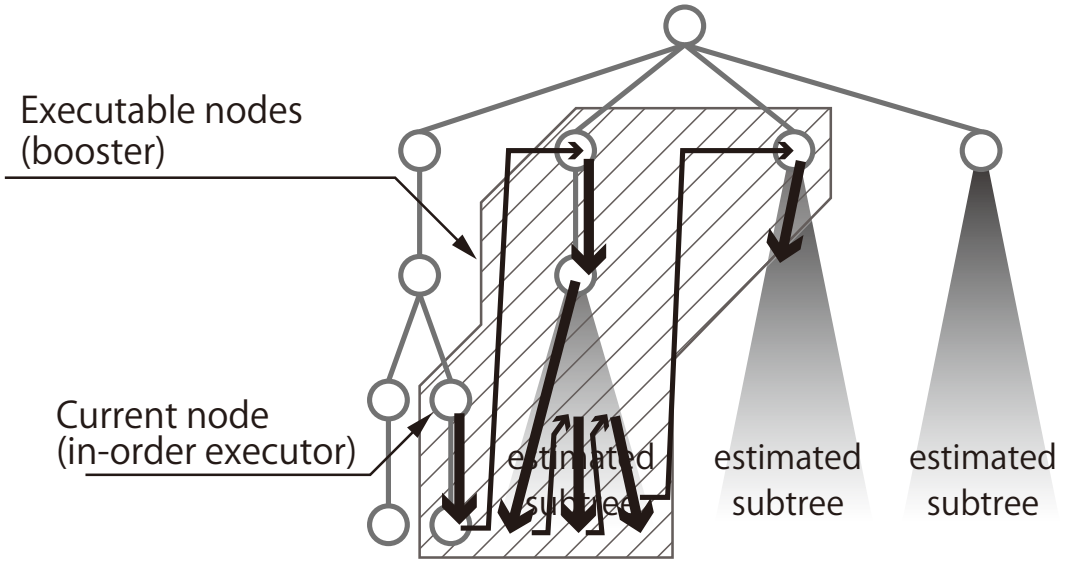
なお，ここに示したデータベース演算粒度の設定は一例に過ぎず，ここまでの議論は任意のデータベース演算粒度設定に対して適用可能である．例えば図 3.7 に示す例では，索引検索を該当索引エントリ取得 (f_1) とタプル実体取得 (f_2) の 2 つのデータベース演算に分割しているが，これらを 1 つのデータベース演算としてまとめることも可能であり，また該当索引エントリ取得を索引ページ取得単位で更に細分化することも可能である．データベース演算の粒度がより細かいほど，クエリ実行の有するデータ並列性を精緻に抽出し，入出力を非同期化する機会を拡大することができる．一方で，演算インスタンス生成数の増加や演算インスタンス経路の情報量増加など，クエリ実行管理に要する一定の演算資源・記憶資源のオーバヘッド増大の要因ともなるため，必ずしも細分化が適切であるとは限らない．最適なデータベース演算の粒度設定は，ハードウェアアーキテクチャやデータベースエンジンの具体的設計に依る部分が大きいため，ここではその詳細に立ち入らないこととする．

3.3.2 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全て把握できる．即ち，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明らか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ものを正確に把握することができるため，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，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，クエリ実行完了を以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，事前に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知ることは困難であるため，神託アルゴリズムに対す

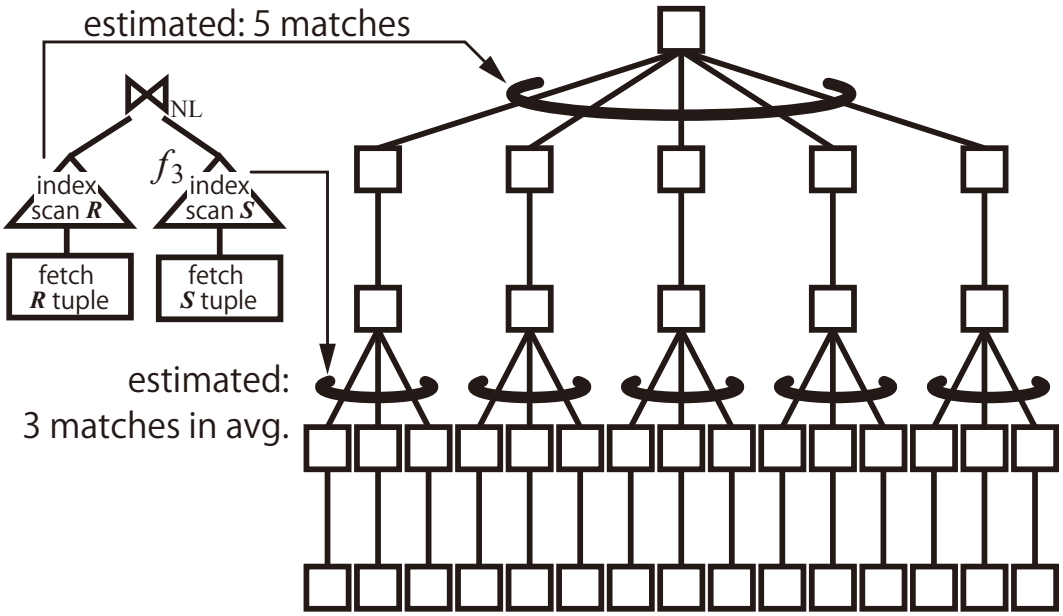


(a) クエリ実行中の演算インスタンス木

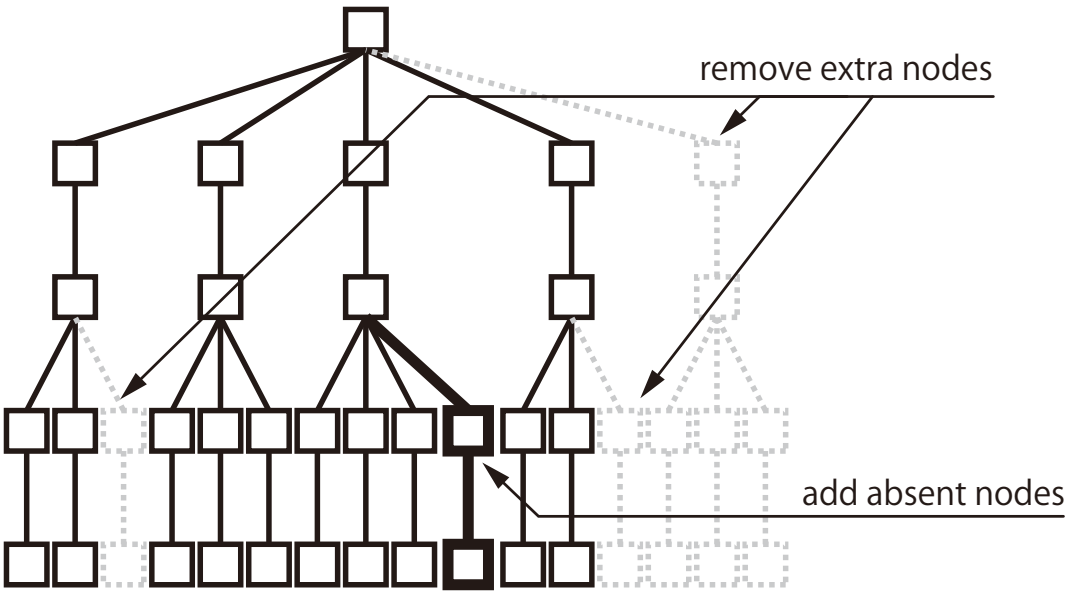


(b) クエリ実行中の演算インスタンス木と未知部分木予測

図 3.8　演算インスタンス木と未知部分木予測による先行実行可能範囲の算出



(a) クエリ実行前の予測演算インスタンス木



(b) クエリ実行中の予測演算インスタンス木と動的補正

図 3.9　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

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，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，よりインオーダ型クエリ実行器に近い順序でページアクセスがなされるため，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 3.8**(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出来ない．

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 3.8(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，完全に正確では無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 3.8(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た先行実行可能範囲を求める手順を説明する．

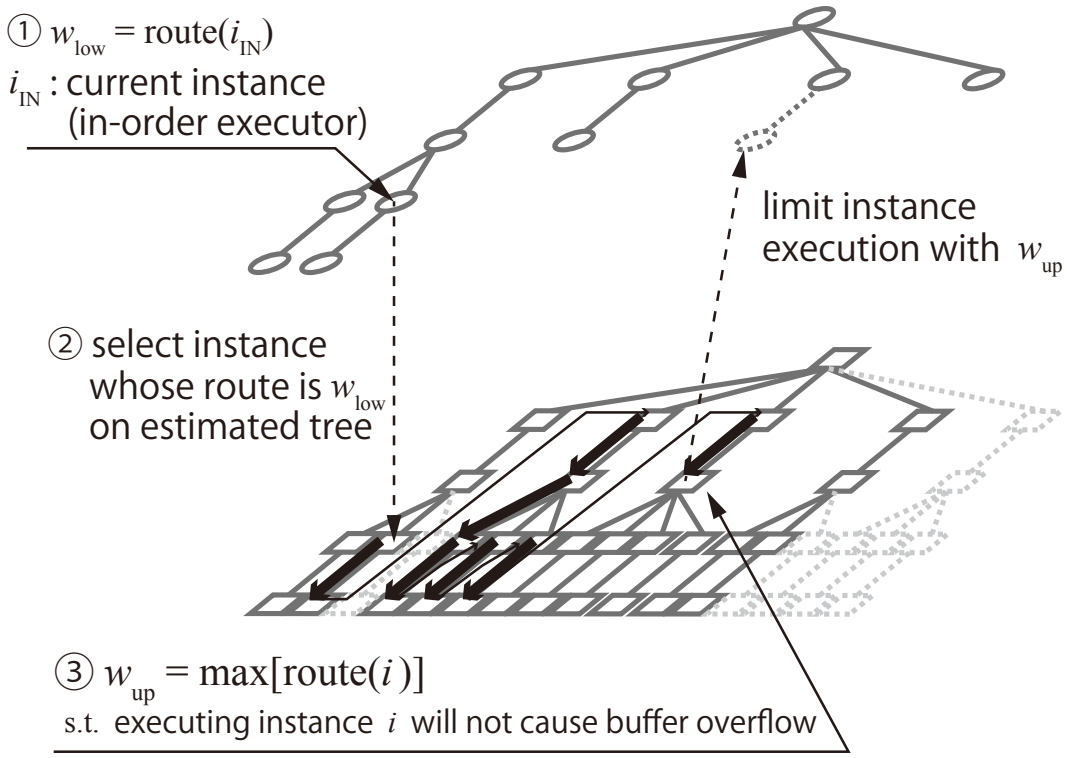


図 3.10　演算インスタンスウィンドウの更新手順

まず予測演算インスタンス木の生成は，データベースが有する各種統計情報[*2]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．例えば図 3.7 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．この時には，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 3.9**(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．

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，即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

[*2] 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等の統計情報を有している．

なるので，その都度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．例えばクエリ実行前に図 3.9(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 3.9(a) に示すよう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 3.9(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みなすことで，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 3.10** 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上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度に行い，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．

3.4 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，即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け付けると，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．即ち，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 3.11** 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 ページは既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

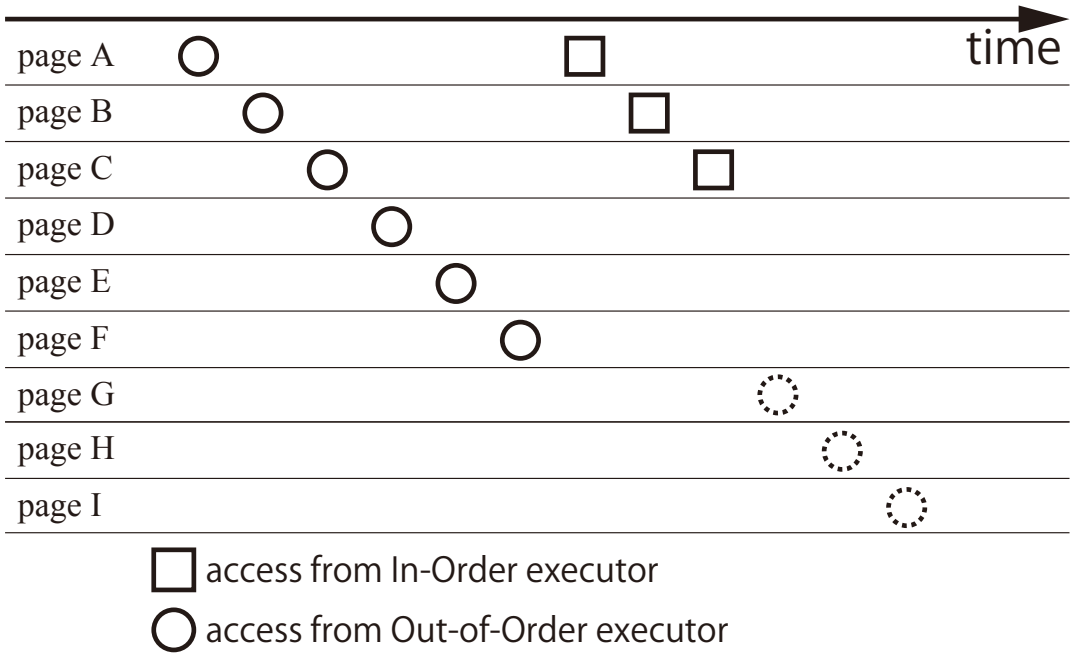


図 3.11　未消費ページ追い出しの発生する一例

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一通りに限らないが，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，まず消費された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殆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．

第 4 章

データベースエンジン加速機構の試作実装と評価

本論文の提案するデータベースエンジン加速機構は，既存のデータベースシステムや，データベースアプリケーションをはじめとする膨大なソフトウェアをほとんど変更することなく，アウトオブオーダ型データベースエンジンの高いクエリ実行性能を活用出来るという，ソフトウェア工学的な費用対効果が極めて高い点にその特徴を有する．すなわち，データベースエンジン加速機構の有効性を評価する上で，実際のデータベースシステムにおいて動作する加速機構の開発を実施し，その挙動検証・性能評価を行うことは不可欠であるといえよう．本章では，データベースエンジン加速機構の試作実装の設計と実装について詳解し，その後にクエリ実行性能，および挙動検証に関する実験の結果を示すことで，データベースエンジン加速機構の有効性を明らかにする．

4.1 PgBooster: PostgreSQL に基づく試作実装

本論文で示すデータベースエンジン加速機構の試作実装である PgBooster は，データベース管理システム PostgreSQL [21] をベースシステムとし，これにプラグインとして組込む形で開発された．PostgreSQL は，オープンソースのデータベース管理システムの中でも広範なユーザベースを持つ代表格であり [90]，ソフトウェア規模はコアコンポーネントのみでも C 言語で約 230 万行に達する．また，PostgreSQL の前身である POSTGRES [91] の設計指針の 1 つである拡張性の高さを引き継ぎ，データ型，オペレータ，アクセスメソッドをユーザレベルで拡張可能な枠組みを有することから，オープンソースコミュニティによる拡張機能の開発も活発に行われている．このような豊富な周

辺ソフトウェアを含め，PostgreSQL の機能性を一切損なうこと無くアウトオブオーダ型データベースエンジンの高速性を取り込むことを目的とし，PgBooster の設計および実装を行った．

4.1.1 PgBooster の設計と実装

PgBooster 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を，PostgreSQL のクエリ実行エンジン部にプラグインする形態をとる．アウトオブオーダ型クエリ実行が高速性を発揮するクエリ実行に際しては，PgBooster のアウトオブオーダ型クエリ実行器による先行的データ読込みにより，PostgreSQL の既存クエリ実行器におけるバッファヒット率が大幅に向上し，クエリ実行が高速化されることが期待される．また PgBooster の機能が無効化された場合には，クエリ実行の性能を含め振る舞いの一切が通常版の 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，次に挙げる点に関しては 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．(C) バッファマネージャにおける GCLOCK ページ置換ポリシに第 3.4 節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一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．

PostgreSQL のコアコンポーネントは，約 230 万行の C 言語のソースコードから構成されている．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 2193 行，PgBooster 自体の新規コード 4371 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．

4.1.2 拡張データ型に対する問合せ

PostgreSQL は拡張データ型をユーザレベルで定義することが可能であるだけでなく，Hellerstein らによって提案された汎用の木構造索引 GiST（Generalized Search Tree）[92] を実装しており，拡張データ型に対するアクセスパスを提供している．GiST の索引生成および検索アルゴリズムは，Consistent，Union，Compress，Decompress，Penalty，PackSplit という 6 種類の基本メソッドによって記述されており，データ型ごとにメソッド実装を与えることで，任意のデータ型に対する索引生成と索引検索に対応することができる．

PgBooster では PostgreSQL 標準の B^+ 木索引に加えて，GiST 索引を用いた索引検索の実行をサポートする．すなわち，GiST 索引が利用可能な任意の拡張データ型に関する問合せに関して，アウトオブオーダ型クエリ実行の高速性を活用することができる．

4.2 評価実験環境

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．実験システムの構成諸元を**表 4.1** に示す．サーバは 4 プロセッサ 24 物理コアと 32 GB の主記憶を搭載し，64bit 版 RedHat Enterprise Linux Server 5.8[*1]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

[*1] OS カーネルは Linux 2.6.18

表 4.1　実験システム諸元

Server: IBM x3850 M2	
Processor	4x Intel Xeon X7460 6-core processor (2.66GHz)
Memory	32GB DDR2 DIMMs
HBA	8x Zephyr-X LightPulse Fibre (for DS5300)
OS	RedHat Enterprise Linux 5.8 (Linux kernel 2.6.18-308.el5)
Storage: IBM DS5300	
RAID controller	Dual active
Cache memory	2x 4GB
Host interface	8x 4Gbps Fibre Channel
Disk drive	160x 450GB 15,000rpm FC HDD
RAID volume	20x RAID5(7D+1P) volumes

IBMDS5300 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 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．

4.2.1　ディスクストレージの基本入出力性能測定

PgBooster の評価実験に先立ち，評価実験環境におけるストレージの基本入出力性能測定を測定する実験を行った．PostgreSQL では 8KB のページ単位で入出力が発行され，特に本論文の対象とするワークロードにおいては典型的にランダム読み込みが行われるため，ランダム読み込みの入出力性能がクエリ実行性能に大きく影響する．そこで本実験においては，8KB ランダム読み込みの入出力性能を測定した．

実験にあたっては，Linux カーネルの提供する非同期入出力機能を利用し，入出力対象デバイスに非同期読込み要求を発行し続けるテストプログラムを実装した．テストプログラムは `open(2)` システムコールを `O_DIRECT` フラグを設定して呼び出すことで，Linux カーネルのページキャッシュを迂回し，直接デバイスへと入出力要求を発行する．入出力

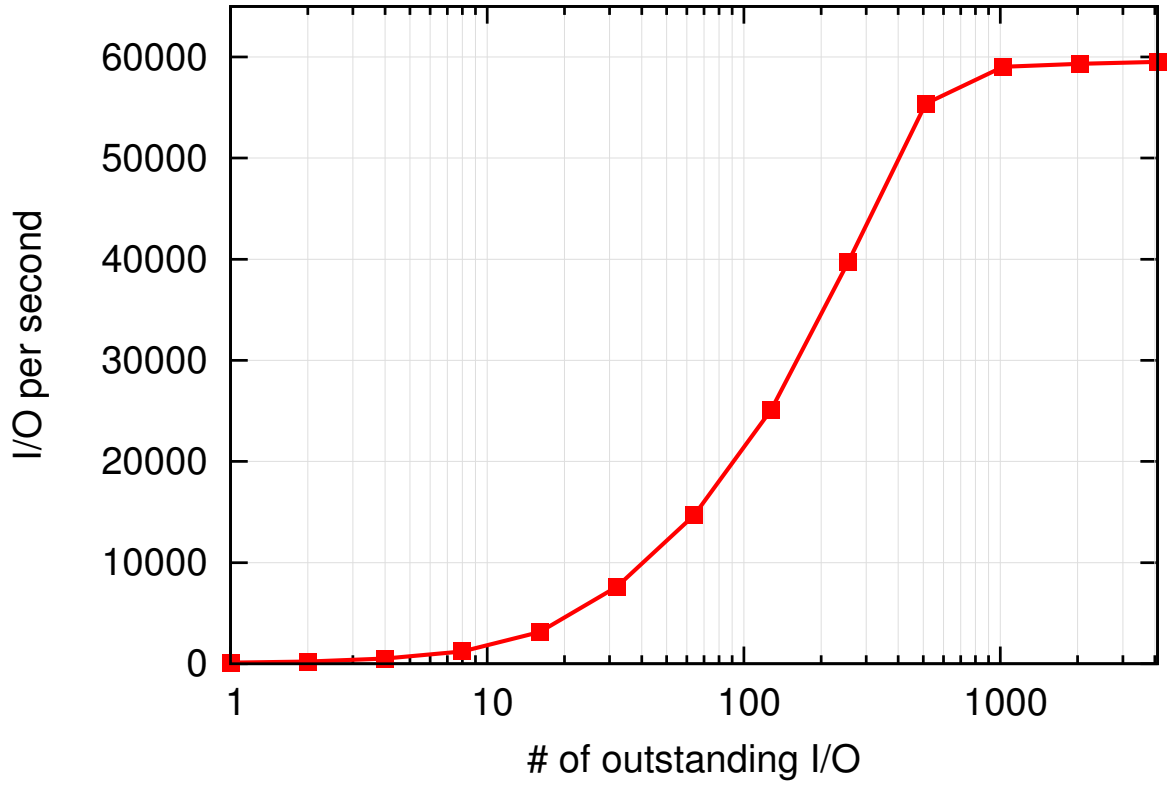


図 4.1　8KB ランダム読み込みにおけるアウトスタンディング入出力数と入出力性能

性能測定の方法としては，当該プログラムによって 60 秒間非同期入出力の発行を実施し，実行開始 5 秒後から実行終了 5 秒前までの入出力処理数を計測することで，1 秒あたりに処理された入出力数（IOPS）を算出することとした．テストプログラムが発行する非同期入出力の多重度を 1 から 4096 まで変化させ，それぞれについて入出力性能の測定を実施した．

結果を**図 4.1** に示す．横軸は非同期入出力の多重度を示し，縦軸は入出力性能を示す．多重度 1 のときには 101 IOPS であり，多重度 1 から 1024 まではその増加に応じて IOPS が増加し，多重度 1024 のときには 59000 IOPS であった．また，多重度が 1024 以上の場合には，入出力性能にはほぼ変化はみられなかった．実験環境におけるストレージシステムが同時受付可能な入出力要求数は 1024 であるため，非同期入出力の多重度が 1024 以下では，多重度に応じて入出力性能が高くなることが期待され，これは実験結果と一致する．以上より，非同期入出力を高多重に発行することでストレージの入出力性能を高効率に活用可能であることが確認できる．すなわち，データベースエンジン加速機構

による非同期的な入出力発行によって，クエリ実行の大幅な高速化が期待される．

4.3 TPC-H データセットを用いた有効性評価実験

データベースエンジン加速機構は，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．分析的ワークロードにおける評価を行うため，意思決定支援系ベンチマークの業界標準である TPC-H [93] のデータセットを用いた．TPC-H データセットは公式提供される `dbgen` コマンドによって Scale Factor = 1000 で生成し，PostgreSQL にロードを実行した後に，クエリ発行対象となるカラムに対して索引を生成した．テーブルおよび索引の総容量は約 2.5TB であった．PostgreSQL のバッファプール容量は 256MB として実験を行った．以降で示す実験結果は，特に説明のない限り各クエリ実行の測定前には 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した上で，PostgreSQL を再起動してバッファプールの消去を実施したものである．

4.3.1 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 規定クエリのうち本論文執筆時点で 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[*2] ．本実験では，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上で，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 4.2** に，また PG に対する PG+Booster の加速率を**図 4.3** に示す．このグラフより，複数の種類のクエリについて PG+Booster は 100 倍以上の性能向上を実現しており，またいずれのクエリ実行についても PG+Booster は PG の処理性能を下回ることはないことが確認できる．

性能向上が顕著である Q3 に着目すると，これらのクエリ実行における入出力パターンは索引走査に起因してランダム性が高く，PG によるインオーダ型クエリ実行ではストレージの入出力帯域の利用率が極めて低いために，PG+Booster において 350 倍の性能向上が確認された．

Q3 に次いで性能向上率の高い Q8，Q11 においても同様の傾向が確認された．一方で

[*2] 評価実験に用いた PgBooster では，left-deep なネステッドループ結合および索引検索からなるクエリ実行プランと，このようなプランの出力結果に対するソート，集約演算を含むクエリ実行プランをサポートする．

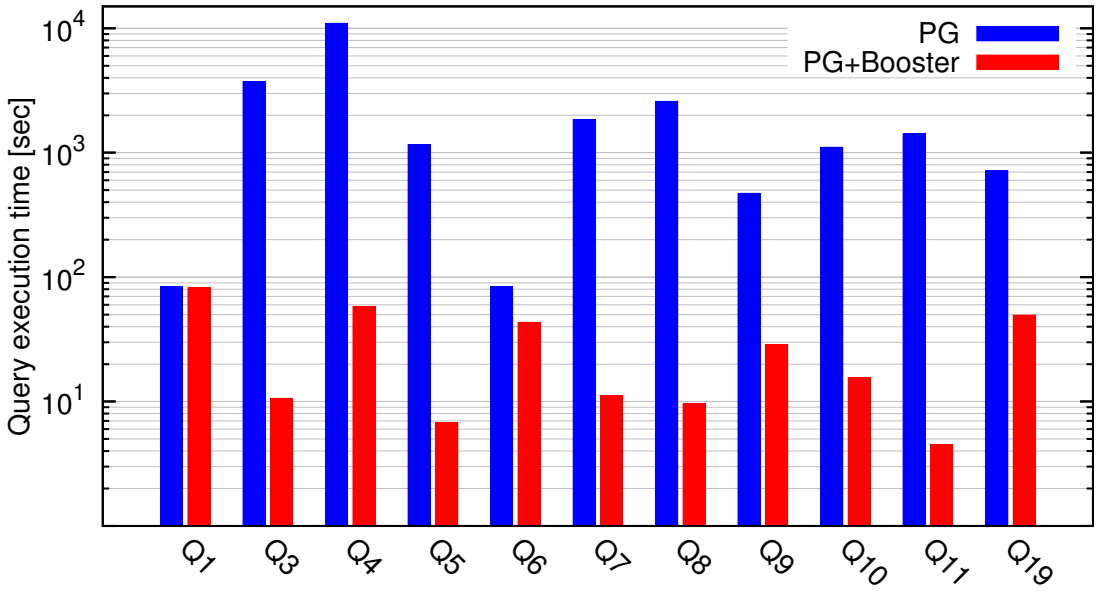


図 4.2　TPC-H 類似クエリ実行時間

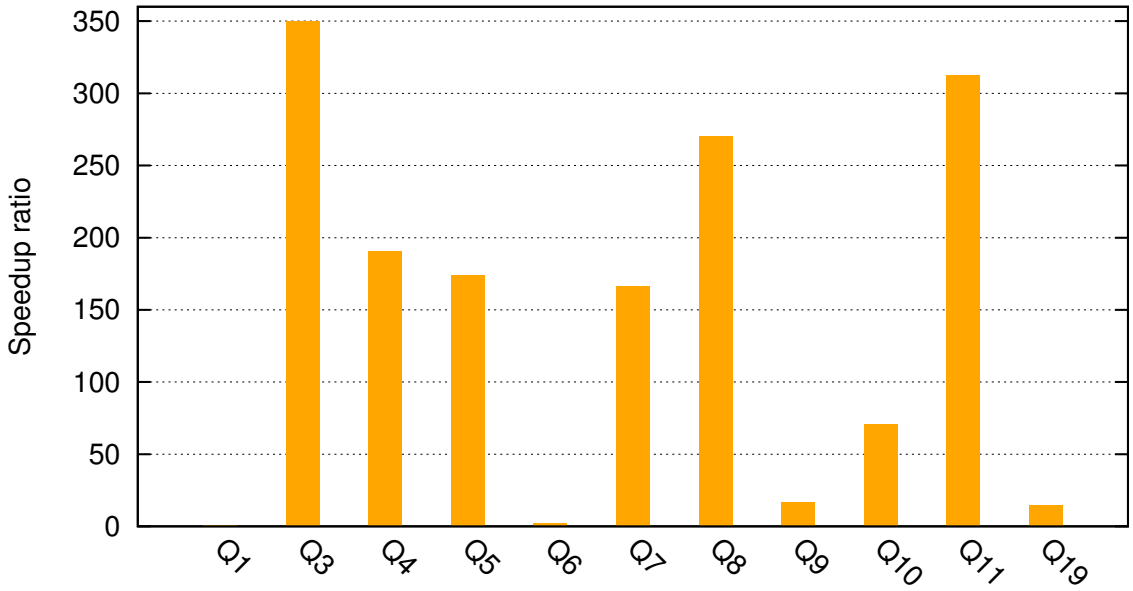


図 4.3　PG+Booster による TPC-H 類似クエリ高速化率

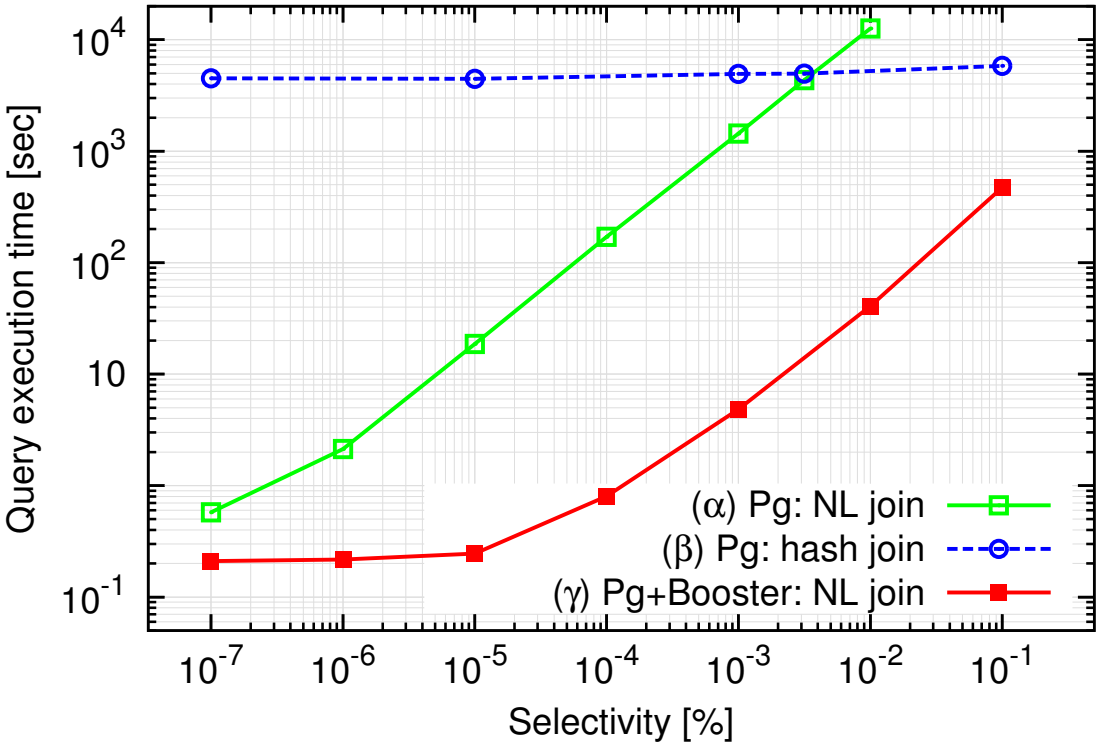


図 4.4　各実行方法における選択率とクエリ実行時間の関係

Q1 や Q6 のように性能向上率が低いクエリは，インオーダ型クエリ実行によって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 でもある程度ストレージ入出力帯域を活用できるものであった．

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高速化可能であることが確認された．

4.3.2　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[*3]．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 4.4** 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．

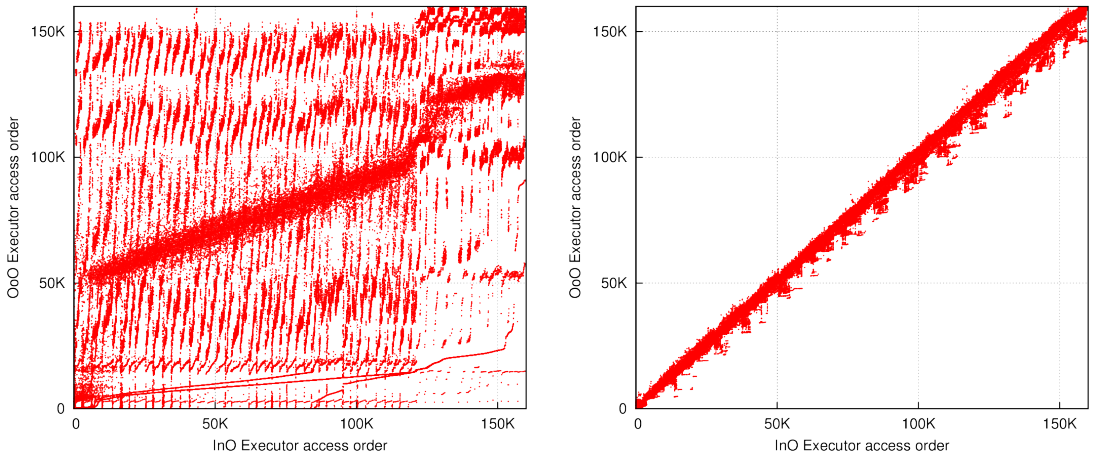
4.3.3 アウトオブオーダ型クエリ実行の協調的制御アルゴリズム評価

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 4.5** に示す．図 4.5 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．

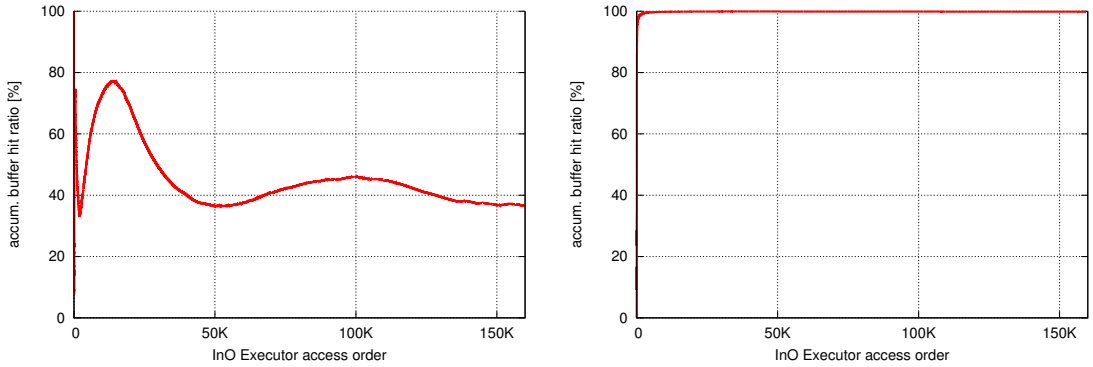
まず図 4.5(a) より，コーディネータを無効化した場合はインオーダ型クエリ実行器とアウトオブオーダ型クエリ実行器のページアクセス系列は，殆ど相関性が見いだせないことがわかる．一方でコーディネータを有効化した場合には，図 4.5(b) に示すようにペー

[*3] 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 4.5　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



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

図 4.6　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

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 4.6** 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，

```
1  CREATE TABLE gps (
2      "id"       varchar,
3      "time"     timestamp,
4      "lat"      double precision,
5      "lon"      double precision,
6      "altitude" double precision,
7      "the_geom" geometry(Point,4326)
8  );
```

図 4.7　GPS データ格納用テーブル定義

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 (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．

4.4　GPS データセットを用いた拡張データ型に対する問合せ性能評価

GiST 索引を用いた拡張データ型に対する索引検索性能を評価するため，空間データを扱う GIS データ処理に関するクエリ実行性能を測定する実験を行った．PostgreSQL における GIS データ処理には空間データベース拡張 PostGIS [94] を利用した．PostGIS が提供する `geometry` 型は二次元ユークリッド空間上のベクトルオブジェクトを表すデータ型であり，点（Point），直線（Line），曲線（Curve），ポリゴン（Polygon），複数点（MultiPoint），複数ポリゴン（MultiPolygon）等のオブジェクトを表現することが可能である．また GiST 索引機構を利用して `geometry` 型に対する空間索引が提供されており，そのアルゴリズムとしては R-tree [95–98] が用いられている．本実験では，`geometry` 型を用いて空間データを PostgreSQL のテーブルに格納し，`geometry` 型に対する GiST 索引走査を行うクエリ実行をすることで，PgBooster の性能を評価した．

実験用データセットとしては，東京大学空間情報科学研究センターより提供された移動体 GPS データセットを用いた．GPS データセットには時刻，移動体識別子，GPS 位置情報（緯度，経度，高度）からなるレコードが 1 年分，総じて 92 億レコードが含まれる．

```
1  SELECT * FROM gps
2  WHERE ST_Within(gps.the_geom, [region]);
```

図 4.8　空間検索クエリ：the_geom の指す位置が指定領域内に含まれるレコードを取得

当該データセットの全レコードを**図 4.7** に示すテーブルにロードした後，移動体識別子カラムに対して B^+-tree 索引を生成し，緯度・経度カラムから点を表す geometry(Point) 型データを生成して the_geom カラムに格納し，当該カラムに対して GiST 索引を生成することで実験用データベースを作成した．作成後のテーブル容量は 1,003GB，時刻カラム索引の容量は 192GB，移動体識別子カラム索引の容量は 270GB，位置情報カラム索引の容量は 474GB であった．

4.4.1　GiST 索引を用いた空間検索クエリによる性能評価

図 4.8 に示す空間検索クエリを実行し，PG および PG+Booster におけるクエリ実行時間の測定を行った．図 4.8 は the_geom カラムの指す点が指定領域内に存在するレコードを取得するクエリであり，the_geom カラムに対して生成された GiST 索引を用いた索引検索によって実行される．検索対象とする領域を次に示す．ただし，いずれの領域も 100 メートル四方の矩形領域である．

- 領域 A
  - 中心座標：北緯 35°21′34.32″，東経 138°43′51.32″
  - 検索該当数：1,895 件（選択率 2.06×10^{-5}%）
- 領域 B
  - 中心座標：北緯 35°29′38.18″，東経 134°13′33.52″
  - 検索該当数：20,160 件（選択率 2.19×10^{-4}%）
- 領域 C
  - 中心座標：北緯 34°58′18.16″，東経 138°23′19.82″
  - 検索該当数：129,943 件（選択率 0.00141%）
- 領域 D
  - 中心座標：北緯 43°4′7.11″，東経 141°21′2.03″
  - 検索該当数：213,524 件（選択率 0.00232%）
- 領域 E

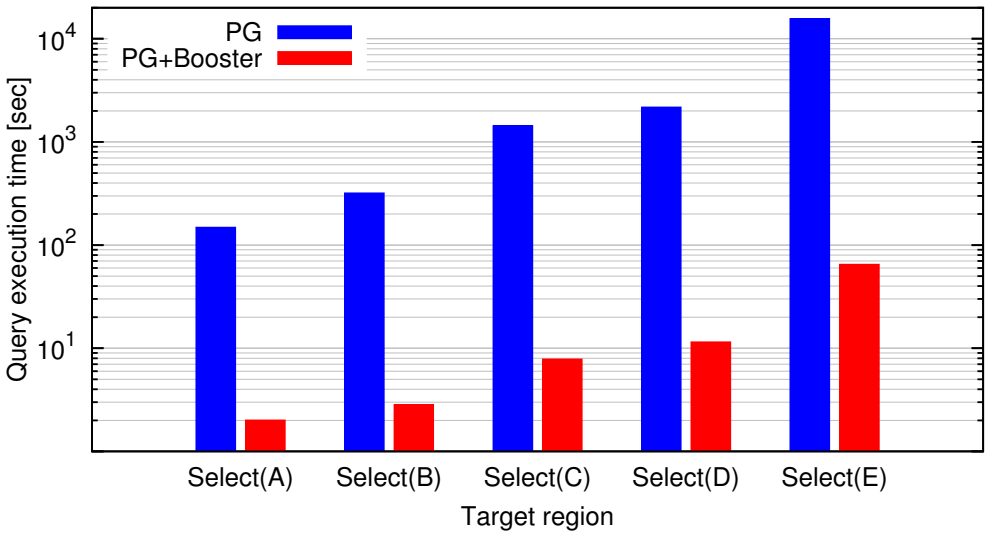


図 4.9　各検索領域に対するクエリ実行時間

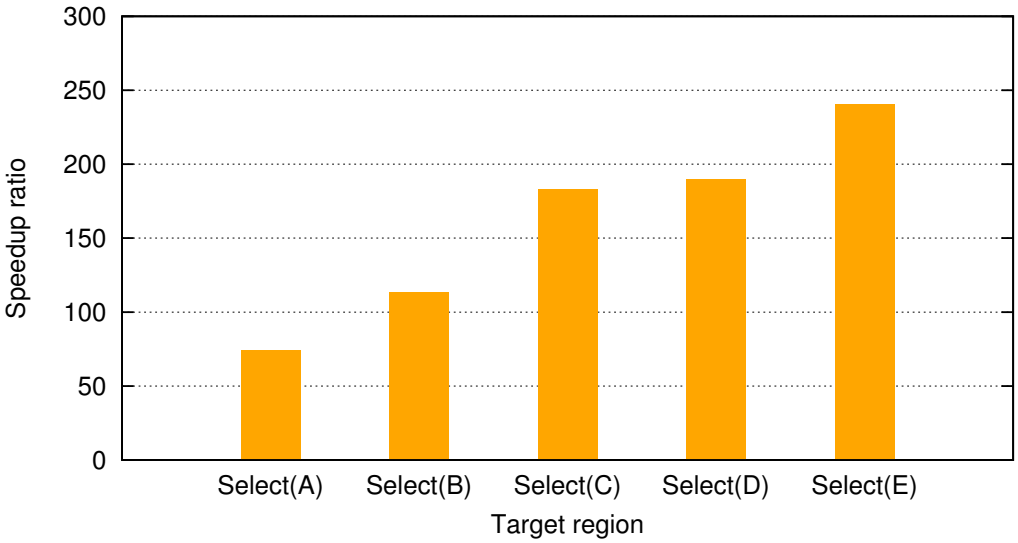


図 4.10　PG+Booster による各検索領域に対するクエリ実行の性能向上率

– 中心座標：北緯 35°41′25.68″，東経 139°42′0.55″
– 検索該当数：1,652,563 件（選択率 0.0180%）

各領域に対するクエリ実行時間の測定結果を**図 4.13** に示す．いずれの領域に対するクエリ実行においても，PG に対して PG+Booster は大幅に短い時間でクエリ実行を完了したことが確認できる．各領域に対するクエリにおける PG+Booster の性能向上率を

```
1  SELECT count(*) FROM gps g1, gps g2
2  WHERE
3    ST_Within(g1.the_geom, [region])
4    AND g1.uid = g2.uid
5    AND g1.time >= '2011-03-11 14:45'
6    AND g1.time < '2011-03-11 15:00'
7    AND g2.time >= '2011-03-11 15:00'
8    AND g2.time < '2011-03-12 15:00';
```

図 4.11　結合を伴う空間検索クエリ

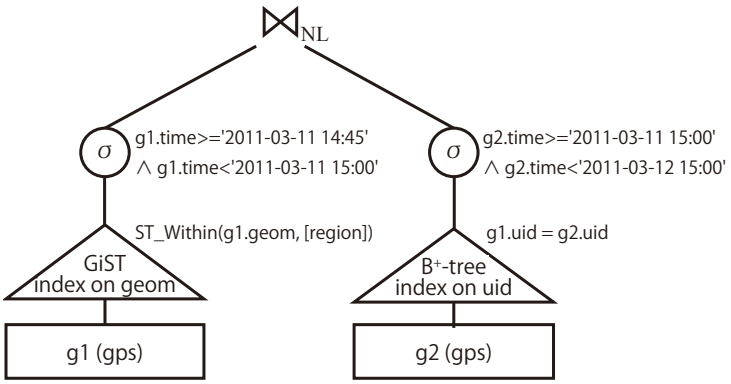


図 4.12　結合を伴う空間検索クエリの実行プラン

図 4.14 に示す．検索該当件数の最も少ない領域 A に対するクエリにおいて，性能向上率は 74.1 倍であり，該当件数が多いクエリほど性能向上率は高く，最大で領域 E における 240 倍であった．一般に索引検索では該当数が多くなるほど入出力量やデータ処理量は増加し，クエリ実行時間は増加する傾向にある．図 4.13 の実験結果はこの傾向に一致する．入出力量が多いほどデータベースエンジン加速機構によるストレージ帯域活用の余地が大きくなるため，該当件数が多いほど性能向上率が高くなることがこの結果より確認できる．

以上の結果より，実データに対する GiST 索引を用いた空間検索においても，PgBooster を用いることで PostgreSQL のクエリ実行が大幅に高速化されるといえる．

4.4.2 結合を伴う空間検索クエリによる性能評価

単なる GiST 索引検索に加えて，**図 4.11** に示す他テーブルとの結合を伴う空間検索クエリを用いて PgBooster の性能を評価する実験を行った．当該クエリは，**図 4.12** に示すように GiST 索引検索と B^+ 木索引検索をネステッドループ結合するクエリ実行プランを，前節に示した領域 A，B，C，D，E に対してそれぞれ実行し，クエリ実行時間の測定を行った．

PG および PG+Booster によるクエリ実行時間を図 4.13 に示す．GiST 索引検索クエリの結果と同様に，いずれの領域に対する移動解析クエリ実行においても PG+Booster は PG よりも大幅に短い時間でクエリ実行を完了したことが確認できる．また検索該当件数が多く，入出力量が大きい領域に対するクエリほど性能向上率が高く，最大で領域 E のときの 262 倍であった．単純な GiST 索引検索と比べて，移動解析クエリでは GiST 索引検索で取得したレコードについて，自己結合により更にレコードを取得するため，入出力量が増加するためデータベースエンジン加速機構による高速化の余地は大きくなる．図 4.14，図 4.14 に示すように，実際に各領域について GiST 索引クエリよりも移動解析クエリのほうが性能向上率が高いことが確認できる．

以上の結果より，実データに対する GiST 索引検索に加えて結合を伴うクエリ実行においても，PgBooster を用いることで PostgreSQL を大幅に高速化可能であることが確認できた．

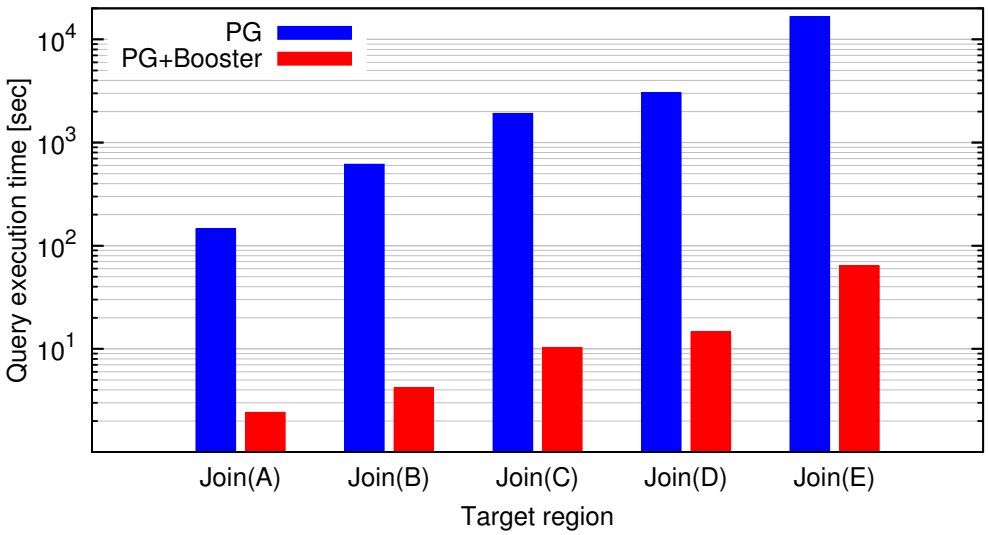


図 4.13　各検索領域に対する移動解析クエリ実行時間

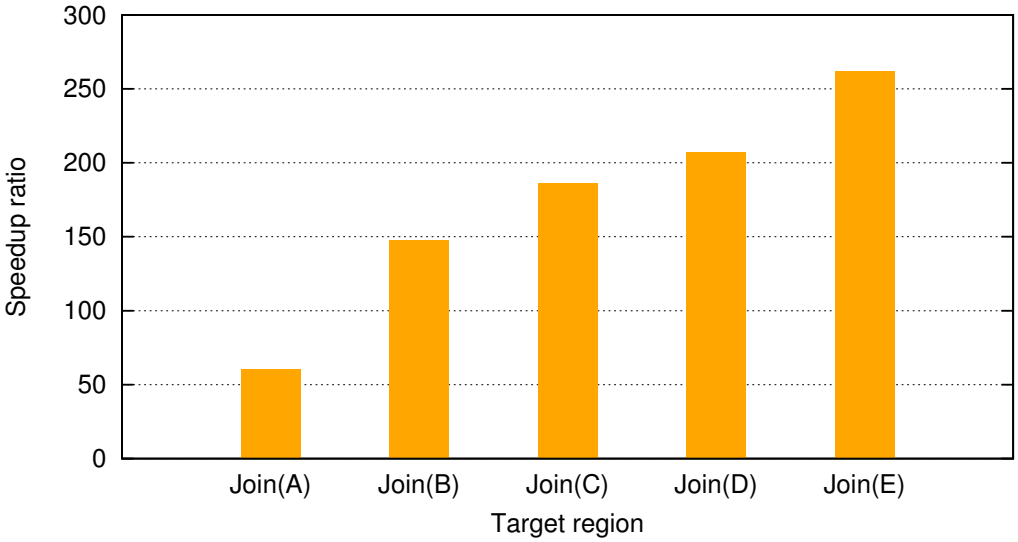


図 4.14　PG+Booster による移動解析クエリ性能向上率

第 5 章

アウトオブオーダ型データベースエンジンにおける複数クエリ実行間の動的資源調停

ここまでのアウトオブオーダ型データベースエンジンに関する議論は，単独のクエリ実行についてのものであった．本章では，アウトオブオーダ型データベースエンジンにおいて，複数クエリを同時並列実行する際の資源配分方式について議論する．

データウェアハウス（DWH）や意思決定支援系（DSS）と呼ばれるデータ分析基盤では，データベースシステムに発行されるクエリはバッチクエリとアドホッククエリの大きく 2 つに分けることができる．バッチクエリは大量の定型データ処理を行うクエリであり，定められた期限までに全処理を完了すればよいため応答性は求められない．アドホッククエリはユーザから応答的に発行される 1 回から数回程度しか実行されない非定型的なクエリであり，短時間で実行を完了することが求められる．データ分析基盤においては単一ユーザが同時に数十から数百の分析的クエリを同時実行することももはや珍しくなく [99]，バッチクエリとアドホッククエリが同時に多数実行されるという状況が想定される．特にデータベース規模が大きくなるほどデータベース全体を走査するようなクエリ実行には長時間を要するため，分析基盤のユーザが関心を持つ一部のデータを選択するアドホッククエリと，大量のデータ走査を伴うようなバッチクエリが多数混在する状況が予想される．

応答性要求の異なる複数のクエリ実行が混在する場合，その要求レベルの高いものほど優先的に演算・入出力資源を割り当て，クエリ実行のスループットを高めることにより優先的に処理することが望ましい．しかしながら，選択性を有するアドホッククエリ実行について，従前のインオーダ型データベースエンジンにおいては入出力遅延が主要なな律速要因であった．そのため，仮に多量の資源を割り当てられようともこれらを有効に活用す

ることが難しく，資源割り当てによるクエリ実行スループット向上の余地は極めて限定的である．一方，アウトオブオーダ型データベースエンジンは高多重に演算・入出力資源を活用可能であるため，利用可能な資源量はクエリ処理性能へと大きく影響することが考えられる．すなわち，優先度に基いて資源を割り当てることができれば，クエリ要求に応じた処理性能を実現することができる可能性がある．

本章では，アウトオブオーダ型データベースエンジンにおける実行モデルをクエリスループットの観点から整理し，クエリ優先度に基づく複数クエリ実行間のスループット調整方式について論ずる．そして，TPC-H データセットを用いた実験により当該方式が有効に機能することを検証する．

5.1 アウトオブオーダ型データベースエンジンの実行モデル

従前のインオーダ型データベースエンジンでは，クエリ実行プランに示された実行論理に従い逐次的にクエリ処理が行われる．一般にクエリ実行プランは木構造で表され，各ノードはクエリ処理に必要な演算を表す．演算を実行する際に，ストレージに格納されるデータが必要とされる場合には，インオーダ型データベースエンジンは当該データを取得するために入出力を要求し，その完了を待ってから取得されたデータに対して演算を実行する．ここで，入出力とその入出力の結果のデータに対する演算の実行との対を**演算インスタンス**と称する．インオーダ型データベースエンジンにおけるクエリ実行は，演算インスタンスの逐次的な実行の繰り返しとして捉えることができる．

これに対して，アウトオブオーダ型データベースエンジンは動的なタスク分解によって，クエリ処理における演算インスタンスの実行並列性を抽出することにより，システムの資源が許す範囲において多数の演算インスタンスを重層的に実行する．演算インスタンスの実行は，簡単には，**図 5.1** に示すように入出力を行うフェーズと演算を実行するフェーズから構成され，演算インスタンスの実行においては，まず入出力フェーズにおいて演算の対象となるデータを取得するための入出力を行う．この際，入出力サブシステムが同時に受付可能な入出力要求の数は有限であり，よって，この受付可能な入出力数を D とすると，入出力サブシステム内の滞留入出力要求数が D を下回った場合にのみ，アウトオブオーダ型データベースエンジンは新たな演算インスタンスの実行を開始することができる．入出力が完了すると，次に取得されたデータに対する演算フェーズへと進む．演算フェーズにおいては，演算インスタンスにかかるデータを以って演算を実行するためのスレッドが生成され，当該スレッドがオペレーティングシステムのタスクスケジューラによりプロセッサコアに割り当てられることにより実行が駆動される．演算フェーズにおい

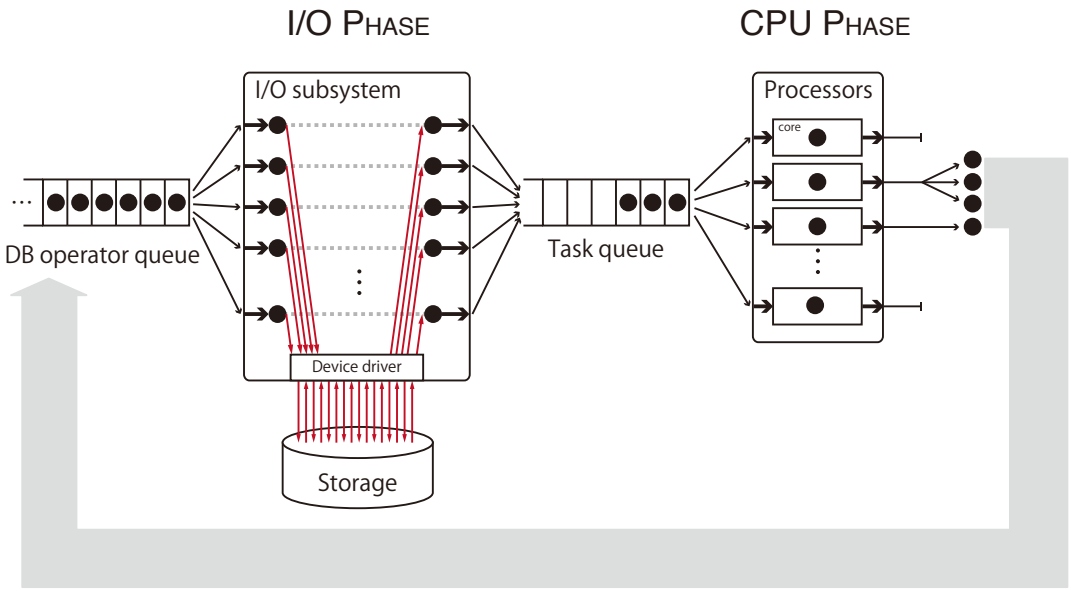


図 5.1　アウトオブオーダ型データベースエンジンの実行モデル

てタプル選択，結合，集約演算等の処理が行われた結果，新たに実行すべき演算インスタンスが生じた場合にはこれらは演算インスタンスキューへと格納され，上述と同様に，随時実行される．このように，アウトオブオーダ型データベースエンジンにおける演算インスタンス実行は図 5.1 に示す図面によってモデル化することができる．

1 つの演算インスタンスの入出力要求処理に要する平均時間を $1/\mu_{\mathrm{io}}$，演算に要する平均時間を $1/\mu_{\mathrm{cpu}}$，またプロセッサコア数を M とすると，各フェーズにおける最大の演算インスタンス処理スループットは次のようになる．

- 入出力フェーズスループット：$\mu_{\mathrm{io}}D$
- 演算フェーズスループット：$\mu_{\mathrm{cpu}}M$

よってアウトオブオーダ型データベースエンジンにおける最大スループット $T_{\max}$ は $\min\left(\mu_{\mathrm{io}}D, \mu_{\mathrm{cpu}}M\right)$ である．

ブロックストレージデバイスにタプルを格納する際には，4KB から 64KB 程度のページに複数のタプルをまとめ，ページ単位で入出力を行うことが一般的である．アウトオブオーダ型データベースエンジンが主に対象とする選択性を有する分析的クエリにおいて，データベース全体に格納されるタプルのうちわずかな割合を取得するものである．そのため，多くの入出力においてページあたりに処理するべきタプルが格納されている数は極小数であることが想定される．今日の計算機システムにおいては，1 プロセッサコアのみで

も毎秒 1000 万タプル以上の処理が実現されるに至っている一方 [100–102]，ストレージ帯域はエンタープライズ級のものであっても最大で 100 万 IOPS 程度であり [103–106]，ストレージ入出力帯域の向上率に対し，プロセッサ処理性能の向上率が今後も上回り続ける見通しであることから [3,4]，アウトオブオーダ型データベースエンジンにおいては，多くの場合において入出力が最大スループットを律速する状況にあるといえよう．実際に第 4 章の評価実験に用いたデータベースサーバにおいて TPC-H Q.3 類似クエリを用いて測定を行ったところ，$\mu_{\mathrm{io}}D = 5.9 \times 10^4 [1/\mathrm{sec}]$，$\mu_{\mathrm{cpu}}M = 6.0 \times 10^6 [1/\mathrm{sec}]$ 程度であった．

入出力フェーズによってスループットが律速される場合，クエリ実行における演算インスタンスのスループットは $\mu_{\mathrm{io}}D$ である．また定常状態においては入出力サブシステムを飽和させるに足る数の演算インスタンスが演算インスタンスキューに充填された状態となり，入出力完了を契機として即座に次なる演算インスタンスの実行が駆動され，入出力要求が発行される．つまり，入出力サブシステム内にはほとんど常に D 個の入出力要求が滞留した状態となる．

5.2 複数クエリ実行間のスループット調整

複数クエリ実行間のスループット調整問題は次のように定義することができる．

> クエリ $Q_1, Q_2, \cdots, Q_N$ に優先度 $p_1, p_2, \cdots, p_N (p_k > 0)$ が付与されている時，
> クエリ Q_k の演算インスタンス処理スループット T_k が次の条件を満たすよう，
> 各クエリの実行を制御する．

$$\forall k (1 \leq k \leq N). \left(T_k = \mu_{\mathrm{io}} D \frac{p_k}{\sum_{i=1}^{N} p_i} \right) \tag{5.1}$$

前節で述べたように，アウトオブオーダ型データベースエンジンにおいて入出力が全体を律速することを仮定した場合，この問題は次のように還元することができる．

クエリ $Q_1, Q_2, \cdots, Q_N$ に優先度 $p_1, p_2, \cdots, p_N (p_k > 0)$ が付与されている時，クエリ Q_k の入出力スループット U_k が次の条件を満たすよう，各クエリの実行を制御する．

$$\forall k (1 \leq k \leq N). \left(U_k = \mu_{\mathrm{io}} D \frac{p_k}{\sum_{i=1}^{N} p_i} \right) \tag{5.2}$$

クエリ Q_k 実行における入出力スループット U_k は，クエリ Q_k に関して入出力サブシステム内に滞留する入出力要求を D_k とすると

$$U_k = \mu_{\mathrm{io}} D_k \tag{5.3}$$

であるため

$$\forall k (1 \leq k \leq N). \left(D_k = D \frac{p_k}{\sum_{i=1}^{N} p_i} \right) \tag{5.4}$$

を満たすよう各クエリ実行を制御すればよい．

式 (5.2) の条件を満たすようクエリ実行を制御する**アルゴリズム 1** を示す．ここでは，同時実行可能なクエリ数は最大 N_{max} であり，N_{max} 個のクエリ実行スロット Q_k が存在するものと仮定している．p_k は Q_k に設定された優先度である．ただし実行中でないクエリ実行スロット Q_k に関しては $p_k = 0$ とする．アウトオブオーダ型データベースエンジンは，入出力サブシステムが新たに入出力要求を受付可能となる度に[*1]，各クエリ実行スロット Q_k に対して QueryTryIO を呼び出し，入出力要求の発行を試みる．この際，Q_k が既に発行中で未だ完了に至っていない入出力数は D_k によって捕捉されており，当該アルゴリズムでは D_k が $D \times p_k / \sum_{i=1}^{N} p_i$ を下回る場合のみ演算インスタンス o に関する入出力要求を発行する．入出力要求 o の完了が検出されると，アウトオブオーダ型データベースエンジンは QueryFinishIO を呼び出し，D_k を更新した上で演算インスタンス o の演算を行うためのスレッドを生成する．

実行中の全てのクエリが定常状態にあり，各々の演算インスタンスキューが十分多くの演算インスタンスを格納している場合，アルゴリズム 1 が式 () に示す条件を満たすよう動作することは自明である．一方，クエリ Q_k のみが実行開始直後や終了直前などの過渡

[*1] このイベントは，オペレーティングシステムのシグナルによる入出力完了通知や，入出力管理用のポーリングスレッドを用いるなどの手段によって検出可能である．

Algorithm 1 優先度に応じた入出力スループット調整アルゴリズム

Initialization
for $i \leftarrow 1, \cdots, N_{max}$ **do**
  $D_i \leftarrow 0$
end for

k : an index of a target query slot Q_k
procedure QUERYTRYIO(k)
  if ISQUEUEEMPTY() **or** $D_k \geq D \times p_k / \sum_{i=1}^{N_{max}} p_i$ **then**
    return
  else
    $o \leftarrow$ POPOPERATORINSTANCE(k)
    ISSUEASYNCIO(o) ▷ Issue I/O request of o
    $D_k \leftarrow D_k + 1$
  end if
end procedure

k : an index of a target query slot Q_k
o : an operator instance whose I/O is finished
procedure QUERYFINISHIO(k, o)
  $D_k \leftarrow D_k - 1$
  STARTTHREAD(o)
end procedure

状態にあるときには，演算インスタンスキューに十分な数の演算インスタンスが格納されておらず，$D_k < \frac{p_k}{\sum_{i=1}^{N} p_i}$ となる可能性がある．即ち，$\sum_i D_i < D$ となり一時的にシステム全体の入出力帯域の利用率低下が発生する．クエリ Q_k 以外は定常状態にあり入出力帯域を最大まで利用できていたとすると，時刻 t_0 から t_1 に至る間の損失である入出力量

$L(t_0, t_1)$ は次のようになる．

$$
\begin{aligned}
L(p_k|t_0, t_1) &= \mu_{\mathrm{io}} \int_{t_0}^{t_1} \left(D \frac{p_k}{\sum_{i=1}^{N} p_i} - D_k(t) \right) \mathrm{d}t \\
&= \mu_{\mathrm{io}} D \frac{p_k}{p_k + \overline{p_k}} \Delta t - \int_{t_0}^{t_1} D_k(t) \mathrm{d}t \qquad (5.5) \\
&\left(\text{ただし } \overline{p_k} \equiv \sum_{i \neq k}^{N} p_i, \Delta t = t_1 - t_0 \right)
\end{aligned}
$$

p_k が大きいほど $L(p_k|t_0, t_1)$ の第一項は大きくなるため，優先度の高いクエリほどその過渡状態における損失が大きくなることがわかる．

前述の考察は，システムの入出力スループットを最大限に活用するという観点からすると，クエリ Q_k が過渡状態にあるときにはクエリ $Q_l (l \neq k)$ は $D \frac{p_l}{\sum_{i=1}^{N} p_i}$ より多くの入出力を発行したほうが良いことを示唆する．しかしながら，実際に計算機システムで用いられるストレージの多くは系内に滞留する入出力要求数が多いほど平均応答時間が長くなる傾向にあるため，このような戦略はクエリ Q_k が定常状態に至るまでに要する時間を増大させることが懸念される．特に本方式では応答性要求の高い (優先度の高い) クエリほどその実行時間を優先的に短縮することを旨とする．そのため，クエリ Q_k が定常状態に至るための時間が，他のクエリ Q_l の干渉によって長くなることを抑制するための代償として損失 $L(p_k|t_0, t_1)$ が生じると考えることができる．その意味において，当該アルゴリズムにおいて損失が生じることは妥当であるといえよう．

本論文では，各クエリの優先度と確保可能なスループット性能の比率が同一となるような，基礎的な動的資源調停を示すことを目的とする．すなわち，サービスレベル要求等に応じて優先度設定ポリシを設計し，本手法を適用することで，より高度な資源調停によるクエリ処理性能調整を行うシステムを構築することが可能である．

5.3 評価実験

本論文の提案するアウトオブオーダ型データベースエンジンの複数クエリ間スループット調整方式の評価実験を実施した．実験では，第 4 章において述べた PgBooster のアウトオブオーダ型クエリ実行器に対し，提案手法を試作実装したものを利用した．データベースエンジン加速機構としてアウトオブオーダ型データベースエンジンがクエリを実行する際には，これに並行してインオーダ型のクエリ実行も行われるが，アウトオブオーダ型データベースエンジンの部分のみに着目すると，第 5.1 節，第 5.2 節における議論をそ

表 5.1　経時的振舞いの検証に用いたクエリ一覧.

QueryID	start time [sec]	priority
Q_1	0	100
Q_2	0	100
Q_3	10	100
Q_4	10	100
Q_5	20	4000

のまま当てはめることが可能である.

評価実験環境は第 4 章 4.2 節に示されたものを利用した．またデータセットは第 4 章 4.3 節と同様に TPC-H データセットを Scale Factor=1000 で生成し，PostgreSQL へとロードしたものを利用した．事前の基本入出力性能測定より本実験環境では $D = 480$ として入出力発行数の抑制を行った.

5.3.1　経時的振舞いの検証

本実験では，複数のクエリを異なるタイミングで実行開始してアウトオブオーダ型データベースエンジンの振る舞いを観察することで，提案手法が期待通り動作することを検証した．実験に用いたクエリの一覧を，優先度，および実行開始時刻とともに**表 5.1** に示す．本実験ではいずれのクエリも TPC-H Q.3 の選択条件を調整したものとなっており，各々のクエリが選択するタプルは互いに重複が発生しないよう選択条件の設定を行った．またいずれのクエリも同一の選択率 $1.8 \times 10^{-3}\%$ となるよう調整されており，単独で実行した際にはいずれも実行時間 11.5 秒であることを予備実験により確認した.

各クエリ実行の入出力スループット経時変化を**図 5.2** 時刻 $t = 0$ においてクエリ Q_1, Q_2（共に優先度 100）の実行が開始されると，$t = 2.2$ まで両クエリ実行の入出力スループットは増加し，以降は両クエリの入出力スループットは約 21,600 IOPS，総じて約 43,200 IOPS を定常的に保ちながら推移し，同一優先度のクエリはほぼ同一スループットとなるよう制御できていることが確認できた.

$t = 10$ にクエリ Q_3, Q_4（共に優先度 100）の実行が開始されると，クエリ Q_1, Q_2 は実行中クエリの優先度にもとづいて入出力発行数を $0.25D$ まで絞るため，システム全体の入出力スループットは 22,400 IOPS 程度まで一旦低下した．これは式 (5.5) に示した過渡状態における損失であり，$t = 11.6$ にクエリ Q_3, Q_4 の演算インスタンス不足が解消す

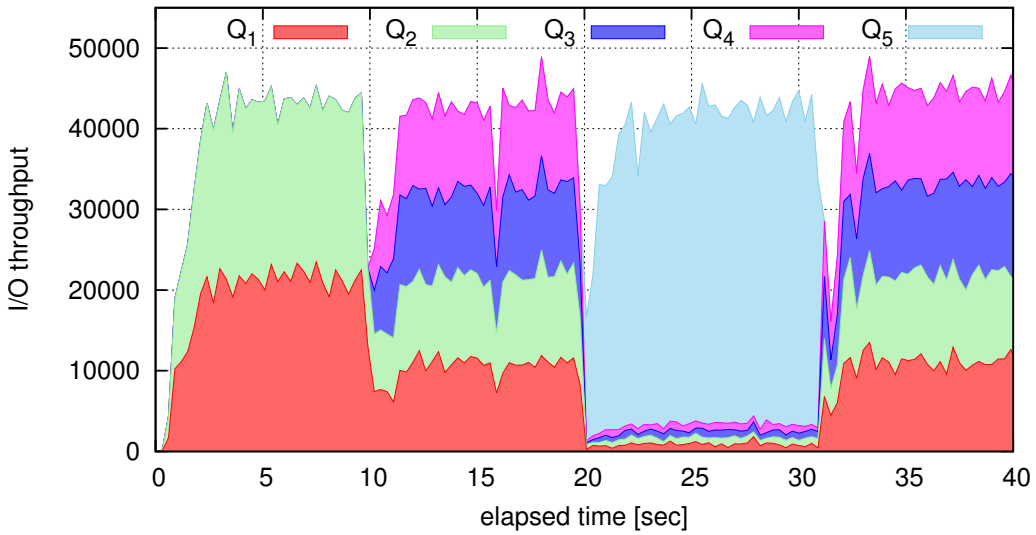


図 5.2　$Q_1, \cdots, Q_5$ の実行における入出力スループット

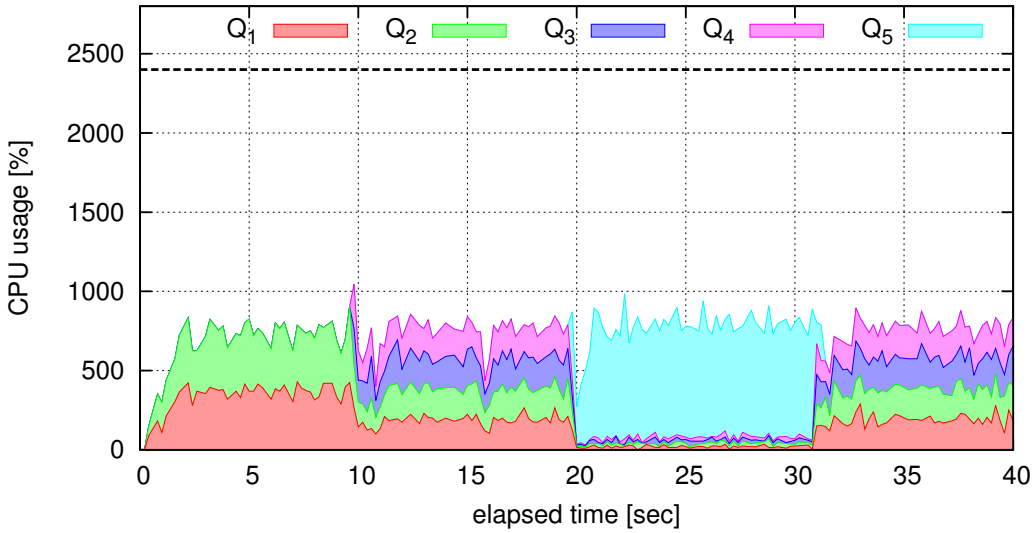


図 5.3　$Q_1, \cdots, Q_5$ の実行における CPU 利用率 (最大 2,400%)

るまで入出力スループットの低下がみられた．それ以降はクエリ Q_5 が開始されるまで各クエリ約 10,600IOPS，総じて約 42,400 IOPS を定常的に保ちながら推移した．ただし $t = 16$ で一旦入出力スループットの落ち込みが見られるが，これは PostgreSQL のバッファプールが満たされたタイミングであり，バッファプールからのページ追い出し論理が起動以降初めて駆動されることに起因するものである．この結果から，同一優先度のクエ

リが複数実行されている場合には，実行開始のタイミングによらず均等に入出力スループットが割り振られることが確認できた．

$t = 20$ においてクエリ Q_5（優先度 4,000）が開始されると，クエリ $Q_1, \cdots, Q_4$ の入出力発行数はそれぞれ $0.023D$ まで絞られるため入出力スループットは 17,000 IOPS 程度まで一旦低下した．この落ち込みは Q_3, Q_4 開始時の落ち込みに比べて大きく，実行が開始されるクエリの優先度が高いほどその過渡期の入出力スループット低下が大きいという式 (5.5) からの予測と一致する．その後クエリ Q_5 の入出力スループットが増加し，$t = 22$ 以降では総じて約 42,200 IOPS を定常的に保ちながら推移した．この際，クエリ Q_5 の入出力スループットは平均して約 38,800 IOPS であり，全体の入出力スループットの 91.9% であった．$\frac{p_5}{p_5+\overline{p_5}} = 90\%$ であるため，概ね目標通りの入出力スループット割り当てが出来ていることがわかる．またこの間クエリ $Q_1, \cdots, Q_4$ はそれぞれ約 860 IOPS と同程度の入出力スループットであった．

クエリ Q_5 の実行が終盤に近づく $t = 31$ 以降，演算インスタンス数の減少により入出力スループットが低下し，$t = 31.2$ に実行を終了すると $Q_1, \cdots, Q_4$ の入出力スループットが再び上昇し，$t = 33$ 以降は約 44,900 IOPS を定常的に推移した．

CPU 利用率を**図 5.3** に示す．各クエリともに概ね入出力スループットに比例する程度の CPU 利用率となっていることが確認できる．実験環境は 24 物理コアを有しており最大の CPU 利用率は 2,400% であるが，本実験では平均 CPU 使用率は 756% 程度であり，入出力によってクエリ実行が律速されいてることがわかる．

以上の結果より，複数クエリの間でスループットを優先度に応じて割り振ることが実現されており，随時クエリ実行数が増減する状況に対応して弾力的にスループット調整が行われることが確認できた．

5.3.2 優先度とクエリ実行時間

各クエリの優先度と同比率で入出力スループットを割り振ることにより，優先度に応じてクエリ実行時間が短縮されることを確認するための実験をおこなった．本実験では，背景クエリとして優先度 100 で TPC-H Q.3 を実行する最中に，対象クエリの実行を開始して，その実行時間の測定を行った．対象クエリとしては背景クエリと同質のワークロードとして TPC-H Q.3（選択率 $1.6 \times 10^{-3}\%$），背景クエリと異質のワークロードとして TPC-H Q.8（選択率 $0.8 \times 10^{-3}\%$）の 2 種類を用いた．TPC-H Q.3 は customer $\bowtie$ orders $\bowtie$ lineitem という 3 表の結合を行うクエリであるのに対し，TPC-H Q.8 は region $\bowtie$ nation $\bowtie$ customer $\bowtie$ orders $\bowtie$ lineitem $\bowtie$ supplier $\bowtie$ nation $\bowtie$ part と 8 表の結合を

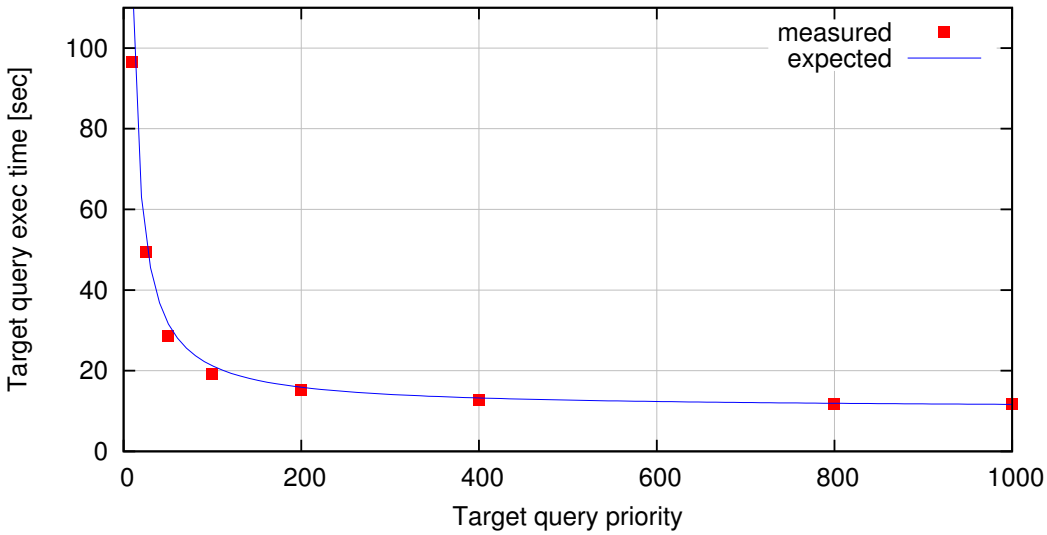


図 5.4　背景クエリ TPC-H Q.3（優先度 100），対象クエリ TPC-H Q.3 のときの対象クエリ実行時間

行うクエリである．背景クエリはその実行開始から，対象クエリの実行終了まで定常状態となるよう TPC-H Q.3 の選択率を適宜調整し，また対象クエリとは選択するタプルが重複しないよう選択条件を設定した．対象クエリの実行時間測定は，その優先度を 10 から 1000 まで変化させて行った．

まず TPC-H Q.3 を対象クエリとした際の，優先度ごとのクエリ実行時間を**図 5.4** に示す．凡例の measured は測定値を，expected は期待されるクエリ実行時間を表す．ただし，期待されるクエリ実行時間は，対象クエリの単独実行時の実行時間を t_{single}，優先度を p としたとき

$$t_{expected} = \frac{p + 100}{p} t_{single} \tag{5.6}$$

とした．TPC-H Q.3 については t_{single} =10.6 秒であった．対象クエリ優先度が優先度 100 以下の場合には，実行時間の実測値が期待される実行時間を 10% 程度下回る結果となった．優先度が 100 より大きい場合には，ほぼ期待した通りの実行時間となることがわかる．測定を実施した各対象クエリ優先度における，実行時間の期待値と実測値の誤差を**図 5.5** に示す．優先度 10 から 800 まではいずれも実測値が期待値を下回る結果となった．背景クエリと対象クエリが選択するタプルは論理的に重複の無いよう選択条件が設定されているが，物理的には異なるタプルが同一ページに含まれている場合が存在することや，両クエリ間で共通する索引ページの取得を行う場合等，入出力対象データには共有領

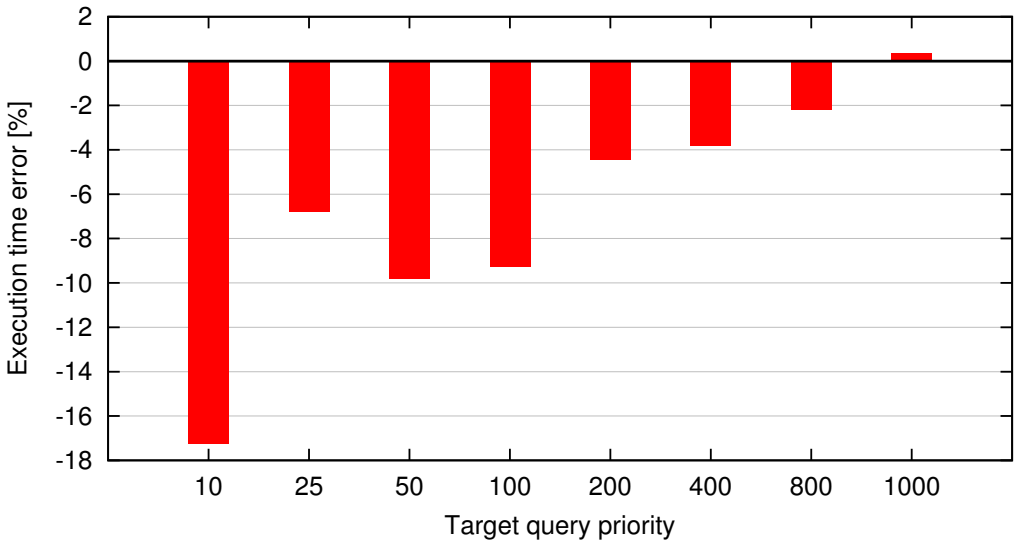


図 5.5　背景クエリ TPC-H Q.3（優先度 100），対象クエリ TPC-H Q.3 のときの期待実行時間と実測実行時間の誤差

域が存在するため，これによって期待されるよりも対象クエリの実行が高速になったものと推察される．また優先度 1000 の場合には実行時間の期待値を実測値が 3.8% 上回る結果となったが，これは対象クエリの選択する表領域に対して，背景クエリによって新たに取得される表領域が入出力対象として加わるため，ディスクストレージにおけるディスクヘッドのシーク時間が増したためであると考えられる．

次に，対象クエリを TPC-H Q.8 としたときの，優先度ごとのクエリ実行時間を**図 5.6** に，実行時間の期待値と実測値の誤差を**図 5.7** に示す．TPC-H Q.3 と比べて対象クエリ優先度 100 以下における予測値と実測値の乖離がより大きくなっていることが確認できる．TPC-H Q.8 は Q.3 と比べて結合段数が多く，入出力全体のうち索引検索が占める割合が高いため，バッファヒット率が高くなる傾向にある．このため，低優先度の場合でも実行時間増加の程度が比較的小さくなったものと思われる．

以上の結果より，対象クエリの優先度が背景クエリに対して大きい場合には，優先度に応じて期待値に近い実行時間が達成されることが確認された．また対象クエリの優先度が背景クエリに対して小さい場合には，論理的に相異なるタプルのみを選択する場合においても一部の入出力対象データが共有されていることから，期待値よりも短い時間で実行を完了することが確認された．

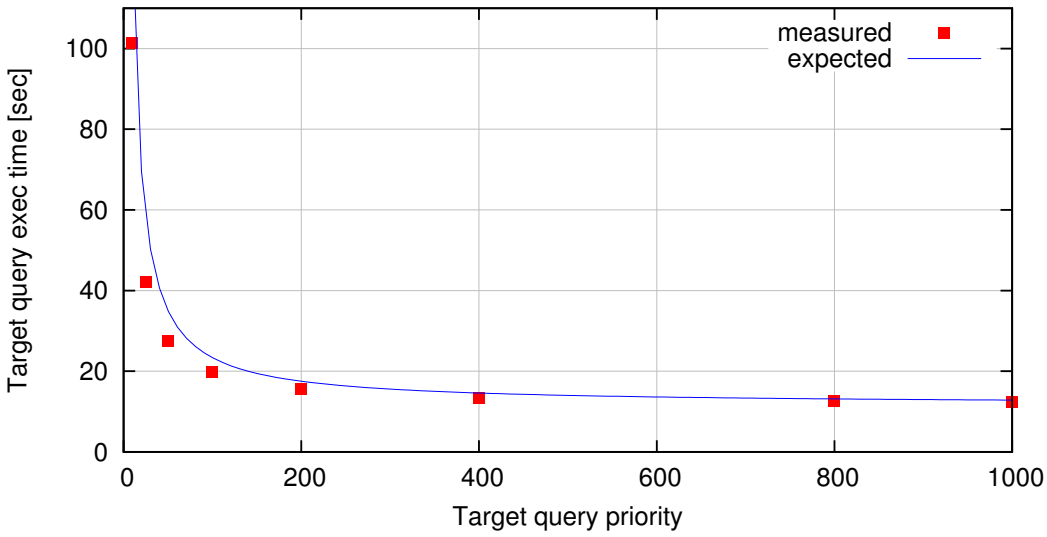


図 5.6　背景クエリ TPC-H Q.8（優先度 100），対象クエリ TPC-H Q.3 のときの対象クエリ実行時間

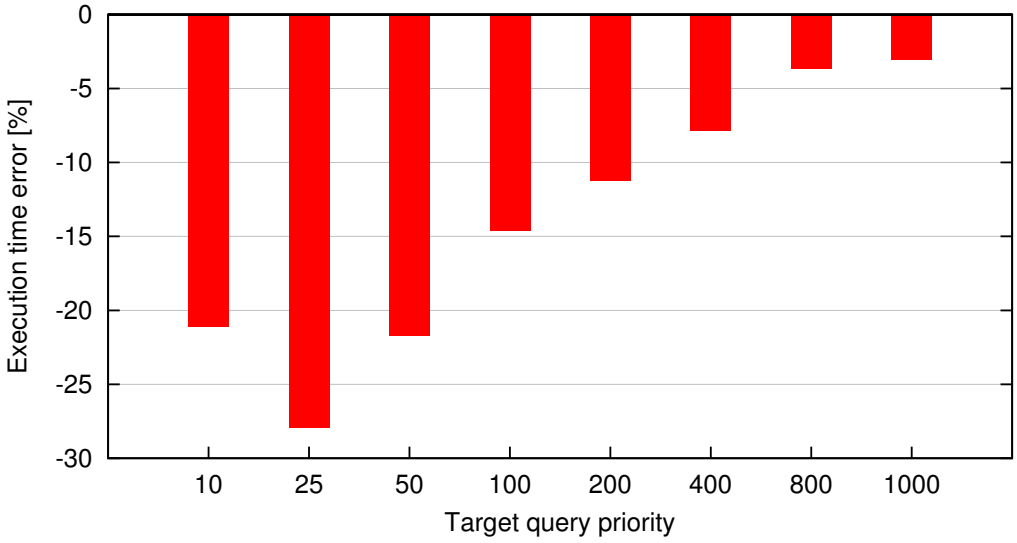


図 5.7　背景クエリ TPC-H Q.8（優先度 100），対象クエリ TPC-H Q.3 のときの期待実行時間と実測実行時間の誤差

第 6 章

演算処理の並列化機能を有するデータベースエンジン加速機構

第 3 章において提案したアウトオブオーダ型クエリ実行方式に基づくデータベースエンジン加速機構は，アウトオブオーダ型クエリ実行器によって先行的にデータをバッファプールに読み込み，インオーダ型クエリ実行器における入出力待ち時間を削減することで，性能向上を実現することを指針として設計された．この設計指針により，既存インオーダ型クエリ実行器の実行論理を一切変更する必要がないため，クエリ実行結果の正当性を担保することが容易であることに優位性を持つ．仮に実装レベルでアウトオブオーダ型クエリ実行器における実行論理に誤りが含まれていたとしても，バッファプール上のページデータを直接破壊するようなことがなければ，単にバッファヒット率の向上が見込まれない結果に終わるだけである．特に本論文が対象としている書き込みを伴わないクエリ実行に関しては，バッファプール上のデータ書き換えを生じるコードパスは論理的に実行されない．

一方，当該設計指針によれば演算処理は従来通りインオーダ型クエリ実行器によって逐次的に実行されるため，ストレージの入出力性能向上に対するクエリ実行性能のスケーラビリティは，プロセッサのシングルスレッド性能によって律速されることが懸念される．プロセッサにおける熱設計の観点から今後もシングルスレッド性能はほぼ向上せず，プロセッサコア数が増加する傾向を踏まえると [107, 108]，将来のデータベースシステムにおける入出力帯域の高効率な活用を考える上で，マルチコア処理性能活用の重要性が増してゆくと予想される．

本章では，マルチコアプロセッサ処理性能の活用を可能とするデータベースエンジン加速機構の設計についてその可能性を検討し，試作実装を用いた初期性能評価によって有効

性を検証する．

6.1 逐次演算実行によるスケーラビリティ制約

既存のインオーダ型クエリ実行器のみでクエリ実行した際の実行時間 T_{IN} のうち，入出力処理に要する時間を T_{io}，CPU 演算処理に要する時間を T_{cpu} とすると

$$T_{IN} = T_{io} + T_{cpu} \tag{6.1}$$

である．データベースシステムにおけるクエリ実行は，基本的にページ単位でデータの入出力を行い，取得データに対して何らかの CPU 演算を行うという形態をとるため，クエリ実行において処理するページ数を N_{io}，1 ページのデータ入出力に要する平均時間を τ_{io}，1 ページのデータに対する CPU 演算に要する平均時間を τ_{cpu}，また入出力・CPU 演算時間比率を $\alpha \equiv \tau_{cpu}/\tau_{io}$ とすると，T_{IN} は次のように書くことができる．

$$T_{IN} = N_{io}\left(\tau_{io} + \tau_{cpu}\right) = N_{io}\tau_{io}\left(1 + \alpha\right) \tag{6.2}$$

システムが広い入出力帯域を有するときには，データベースエンジン加速機構を用いることでその帯域を活用し，クエリ実行を高速化することができる．システムの最大入出力スループットを S_{io}（インオーダ型クエリ実行における入出力スループットを 1 として正規化）とすると，1 回の入出力に要する平均時間は最小で τ_{io}/S_{io} となるため，クエリ実行時間の理論的最小値 T_B は次のようになる．

$$T_B = N_{io}\left(\frac{1}{S_{io}} + \alpha\right) \tag{6.3}$$

よって，データベースエンジン加速機構による性能向上率の最大値 S_B は次のようになる．

$$S_B = \frac{T_{IN}}{T_B} = \frac{1+\alpha}{1+\alpha S_{io}} S_{io} \tag{6.4}$$

式 (6.4) は Amdahl の法則 [109] となっている．

複数の S_{io} について，データベースエンジン加速機構によって得られる性能向上率の最大値の，CPU 演算・入出力時間比率に応じた変化を**図 6.1** 示す．このグラフからわかるように，CPU 演算・入出力時間比率が大きい，すなわちデータあたりの CPU 演算量が多いクエリほど性能向上率が顕著に低下する．またシステムの入出力帯域が大きいほど性能向上率低下の度合いは大きく，入出力帯域の効率的活用が困難であることを示している．データベースシステムの業界標準ベンチマークである TPC [93] の性能上位構成においては数百台のディスクドライブを有する構成は珍しくなく，大規模なものでは単一のサーバ

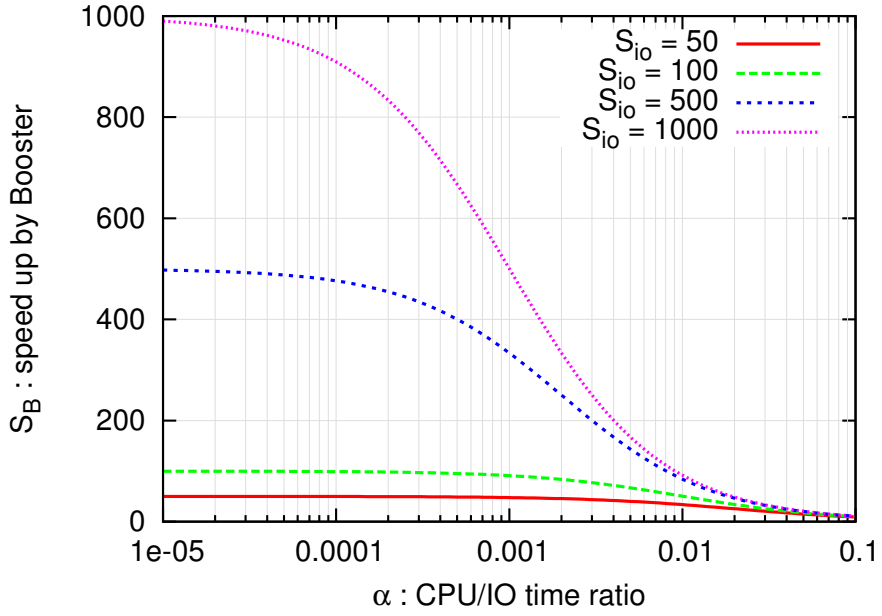


図 6.1　CPU 演算・入出力時間比とデータベースエンジン加速機構による性能向上率最大値 S_B

に一千台以上のドライブが接続される構成もあることから，$S_{io} = 1000$ 程度はエンタープライズ用途においては十分現実的な規模のシステムにおいて生じるうる値である．その性能活用率を考えた時には，$\alpha = 0.001$ という入出力が要する時間が極めて支配的なクエリにおいても，$S_{io} = 1000$ の場合の性能向上率は $S_B = 500$ に留まる．データベースエンジン加速機構はインオーダ型クエリ実行器の結果正当性を損なわないことが論理的に保証され，またデータベースエンジン全体に対して極めて小規模な実装により実現可能であるという利点を有するものの，ここまでの議論はストレージが高性能化するほど逐次演算実行が性能向上率に与える影響が無視できなくなることを示唆する．

そこで，本章ではアウトオブオーダ型データベースエンジンがより積極的にインオーダ型データベースエンジンに関与することで，複数プロセッサコアの並列演算性能を活用し，CPU 演算に要する実質的な時間をも縮減する方法について考えたい．データベースエンジン加速機構の内包するアウトオブオーダ型クエリ実行器そのものは，取得データに対する CPU 演算を並列に実行することで複数プロセッサコアの性能を活用しクエリ実行をする．ただし，その CPU 演算結果はインオーダ型クエリ実行器に与えられることはなく，あくまで先行的入出力によるバッファヒット率の向上を通して，実質的な入出力スループット向上のみを狙うものである．仮に CPU 演算結果をインオーダ型クエリ実行器に適切に注入することが可能であるとすると，アウトオブオーダ型クエリ実行器が複数コ

アを利用して並列実行した CPU 演算結果を受け取ることで本来行うべきであった CPU 演算の一部を省略することが可能となり，結果として複数コアによってもたらされる並列処理性能に応じてクエリ実行性能が向上することが期待される．

複数コア活用によって CPU 演算スループットが M 倍になったとすると，1 ページ分のデータ処理に要する時間は τ_{cpu}/M となるので，この場合のクエリ実行時間 T_{MCB}，およびインオーダ型クエリ実行からの性能向上率 S_{MCB} はそれぞれ次のようになる．

$$T_{MCB} = N_{io}\left(\frac{1}{S_{io}} + \frac{\alpha}{M}\right) \tag{6.5}$$

$$\frac{S_{MCB}}{S_{io}} = \frac{1+\alpha}{1+\alpha\dfrac{S_{io}}{M}} \tag{6.6}$$

ただし，システムのスループット上限は入出力によって律速されることを想定するため，アウトオブオーダ型クエリ実行による入出力スループットの性能向上率 S_{io} を M が超えることは論理的にありえないため $S_{io} \leq M$ である．また M はシステムの有するプロセッサコア数よりも大きくなりうることに注意されたい．入出力によってクエリ実行が律速される状況においては，ベースラインであるインオーダ型クエリ実行をしている際には CPU 利用率は 1 コア分の 100% を大きく下回るため，仮に 1 コアのみを利用する場合でもアウトオブオーダ型クエリ実行による多重的 CPU 演算実行によりスループットは向上し $M > 1$ となる．

この場合のスケーラビリティ上限 S_{MCB}/S_{io} の低下は，S_{io}/M の増加の抑制によって避ける事ができる．即ち，複数プロセッサコアの並列演算性能を有効にインオーダ型クエリ実行器の加速へと利用することができれば，入出力帯域がより高いシステムにおいても有効にその性能を活用することが可能であると期待される．

6.2 並列演算性能を活用可能なデータベースエンジン加速機構

タプル単位でインオーダ型クエリ実行器にデータ供給を行い高速化を行う方式について，まずクエリ実行器においてタプル供給を受付け可能な機会について検討する．第 3 章 3.1 節に示したデータベースエンジンコンポーネントと，データ処理単位の概要を**図 6.2** に再掲する．この図に示すように，クエリ実行器においてタプルデータがやり取りされる流路は次の 2 つに分類できる．

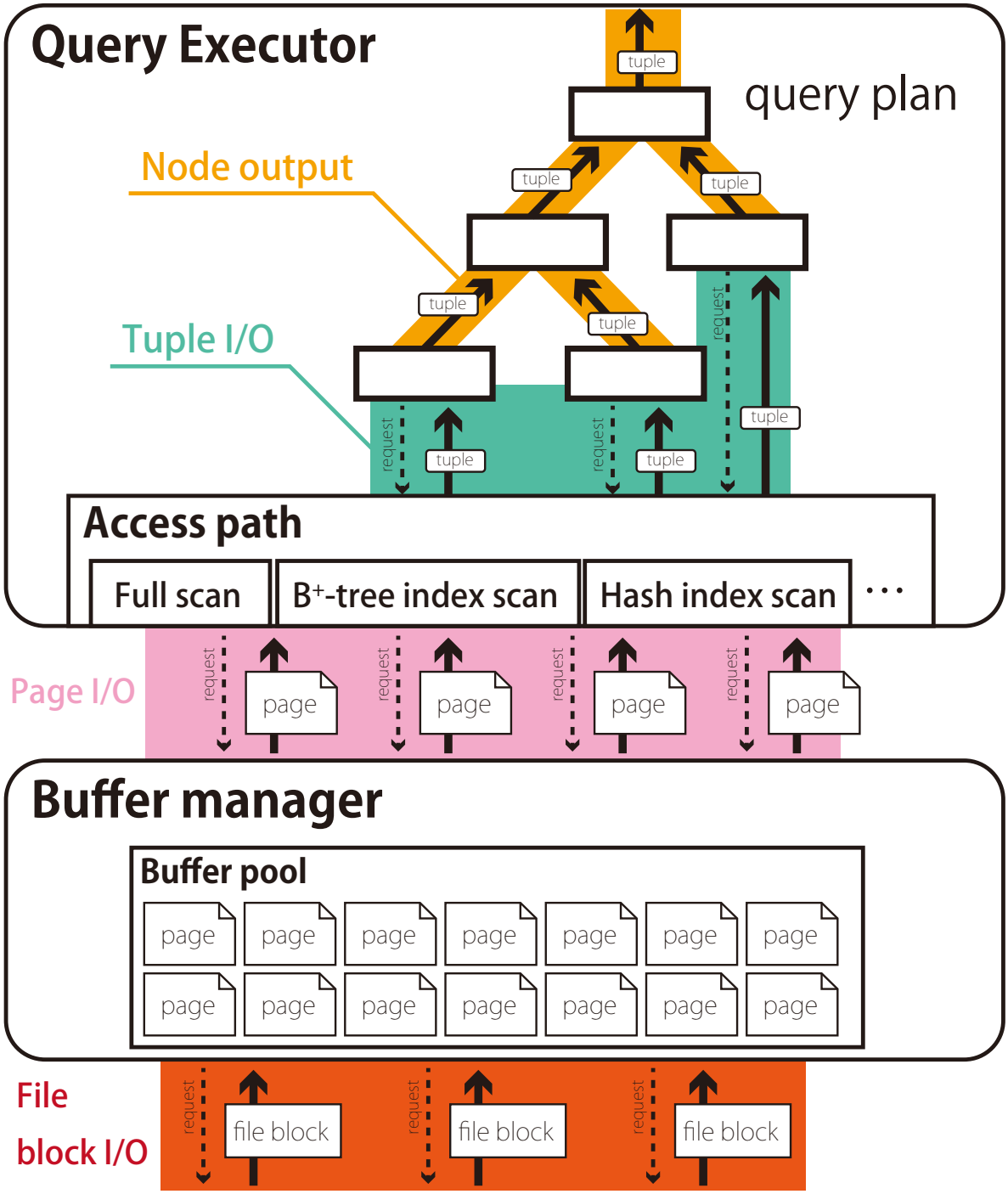


図 6.2　データベースエンジンコンポーネントとデータ処理単位（再掲）

```
1  Tuple GetNext(NestedLoopNode node) {
2    PlanNode outer_node = node.left;
3    PlanNode inner_node = node.right;
4    Tuple outer_tuple, inner_tuple;
5    while (true) {
6      outer_tuple = GetNext(outer_node);
7      if (outer_tuple == NULL) return NULL;
8      // If inner_node need parameters derived from outer_tuple
9      if (inner_node.need_params)
10       InitNodeParams(inner_node, outer_tuple);
11     while (true) {
12       inner_tuple = GetNext(inner_node);
13       if (inner_tuple == NULL) return NULL;
14       return Join(outer_node, inner_node);
15     }
16   }
17 }
```

図 6.3　従前のネステッドループ結合アルゴリズム

プランノード出力

下位のプランノード実行結果として出力されるタプル.

タプル入出力

アクセスパスによって取得されるタプル.

従前のインオーダ型データベースエンジンにおいて広く用いられているイテレータモデル [89] では，プランノードに対して GetNext 関数を呼び，その結果としてプランノードから 1 つタプルを取得する．GetNext が呼び出されたプランノードにおいて，結果タプル生成に下位プランノードの出力が必要な場合，再帰的に GetNext が呼び出される．またストレージ上に格納されたタプルが必要であれば，アクセスパスに対して GetNext を呼び出すことによってこれを取得する.

このようにタプル流路は上位プランノード側から GetNext 呼び出しによって駆動されるため，各プランノードの出力側にタプル供給バッファを設け，GetNext 呼び出しを行うポイントをタプル供給の基点とすることができる（**図 6.4**）．例えば**図 6.3** に示す一般的なネステッドループ結合アルゴリズムでは，アウターノードとインナーノードの 2 つの下位プランノードを持ち，それぞれに対して GetNext を繰り返し呼び出している．この

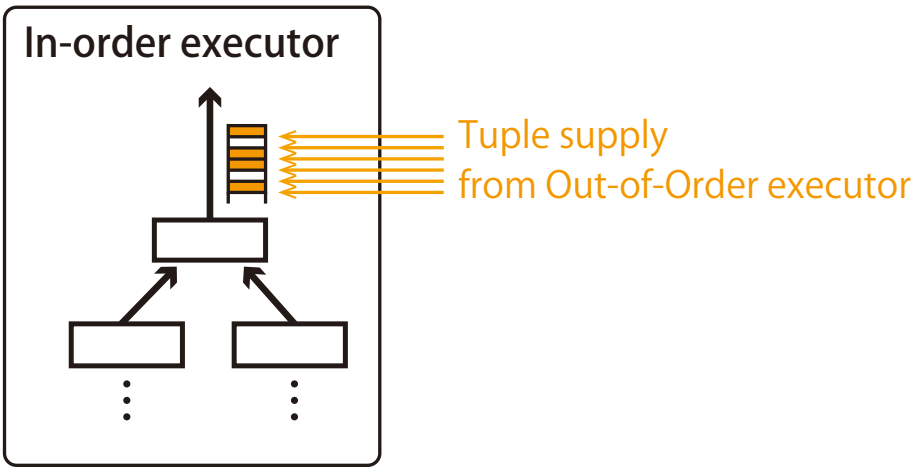


図 6.4　プランノードを基点としたタプル供給

GetNext 呼び出しにおいて，アウトオブオーダ型クエリ実行器から供給されたタプルを取得することで，従来 GetNext において行われていた処理を省略することが可能となる．このような機能を持つ GetNext を **BoostedGetNext 関数**と称することにする．

BoostedGetNext 関数を構成する基本戦略は 2 通り考えられる．

（戦略 1）　アウトオブオーダ型クエリ実行器からの供給タプルが無い場合には，従来通りインオーダ型クエリ実行をして結果タプルを生成する．

（戦略 2）　アウトオブオーダ型クエリ実行器からの供給タプルが無い場合には，タプル供給が行われるまで待つ．

最悪のケースを想定すと，アウトオブオーダ型データベースエンジンから有効にタプルが供給されない場合には，インオーダ型クエリ実行器のみで独立してクエリ実行可能な（戦略 1）がより堅牢である．アウトオブオーダ型データベースエンジンにおけるデータ処理の順序は非決定的であるので，インオーダ型クエリ実行器がある時点から処理していくタプルが $t_k, t_{k+1}, \cdots, t_{k+N-1}$ であるときに，容量 N のタプル供給バッファに対して $t_k + N + i (i \geq 0)$ を先んじて投入することが起こりうる．この場合，後から投入される $t_k, t_{k+1}, \cdots, t_{k+N-1}$ のうちいずれかはタプル供給バッファ容量が一杯で投入に失敗する可能性がある．このとき，（戦略 2）では投入に失敗したタプルを受け取ることができないため，一旦タプル供給バッファからタプルを取り除き，適宜アウトオブオーダ型クエリ実行器において再実行し，改めてタプル供給を行う必要がある．

```
1  // update internal state of node
2  void MoveNext(PlanNode node);
3
4  // pop a tuple from a tuple-supply buffer of node
5  Tuple GetBoostingTuple(PlanNode node);
6
7  Tuple BoostedGetNext(PlanNode node) {
8    Tuple output = GetBoostingTuple(node);
9    if (output != NULL) {
10     MoveNext(node);
11   } else {
12     output = GetNext(node);
13   }
14   return output;
15 }
```

図 6.5　（戦略 1）における BoostedGetNext 実装の例

他方，アウトオブオーダ型データベースエンジンから有効にタプルが供給される場合を考えると，（戦略 2）に対して（戦略 1）はクエリ実行のうち省略可能なコストは限定的である．イテレータモデルでは各プランノードやアクセスパスはスキャンポインタ等の内部状態変数を持ち，GetNext が呼び出されるたびにこれを更新することでクエリの実行状態を管理する．そのため，（戦略 1）ではタプル供給をうけた場合でも内部状態変数の更新を行わなければならない．このときの BoostedGetNext 実装例を**図 6.5** に示す．内部状態変数の更新のみを行う MoveNext は，クエリ実行状態を管理するために下位プランノードおよびアクセスパスに対して再帰的に呼びださなければならず，各プランノードやアクセスパスの種類毎に適切に実装されなければならない．また例えば B^+ 木索引検索における MoveNext の実装を考えると，索引キー以外のタプル選択条件が存在する場合には，索引エントリが指すタプルを順次取得して，選択条件を満たすタプルが存在する索引エントリまでスキャンポインタを進める必要があるため，実質的に GetNext における大部分のコードパスを実行することとなる．一方，（戦略 2）を用いる場合にはアウトオブオーダ型データベースエンジンからのタプル供給に依存するプランノード以下に関して，全てのコードパス実行を省略してタプル供給を受けるため，タプル受け渡しのオーバヘッドが十分小さければアウトオブオーダ型データベースエンジンの性能を高効率に取り込むことが可能である．

ここまでの議論をまとめると，（戦略 1）はタプル供給が失敗した場合のペナルティが小さいものの，成功した場合の利得も限定的であるため，クエリ実行器における実行論理を変更してまで採用するべきかには疑問が残る．（戦略 2）はタプル供給に失敗した場合，アウトオブオーダ型データベースエンジンにおける再実行というペナルティが発生するが，タプル供給に成功した場合，アウトオブオーダ型データベースエンジンの入出力・CPU 演算の結果を直接的に享受することが可能となり，利得は大きい．この際，アウトオブオーダ型データベースエンジンにおいては，第 3 章にて論じた協調制御アルゴリズムを用いることで，インオーダ型クエリ実行器がタプルを必要とする順序にあわせてタプル出力がなされるよう制御を行うことで，再実行ペナルティが発生する可能性は大幅に抑制することが可能であると期待される．よって，本論文では（戦略 2）を採用することとする．

タプル供給の実施にあたっては，GetNext 呼び出しを BoostedGetNext 呼び出しに置き換えるポイントを選択する必要がある．（戦略 2）を採用する場合においては，BoostedGetNext を呼び出すプランノード以下の全ての処理が省略されるため，より上位のプランノードにおいてこれを行うことが望ましい．つまり，クエリ実行プランの中でアウトオブオーダ型データベースエンジンが実行可能な最大のサブツリーの根となるノードにおいて，GetNext 呼び出しを BoostedGetNext 呼び出しに置き換えればよい．

また，タプル供給バッファに投入されるタプルは，インオーダ型クエリ実行器において必要とされる順序が把握可能である必要がある．これは，第 3 章にて示したデータベース演算インスタンスの演算インスタンス木における経路を利用することで表現可能である．即ち，タプルが一意に識別可能な粒度で演算インスタンスを設定し，当該経路をキーとする優先度キューや探索木等によってタプル供給バッファは実装可能である．

6.3 性能評価実験

前節に述べた設計をもとに，タプル供給方式による並列演算性能を活用可能なデータベースエンジン加速機構の試作実装 PgBoosterMC を開発し，その性能評価実験を行った．ただし本実験ではタプル供給方式によって得られる性能利得を確認することを目的とするため，試作実装においてはタプル供給失敗時には一旦処理を中断し，通常のインオーダ型クエリ実行を改めて実行することとした．評価実験に用いた環境は第 4 章 4.2 節に示したものを用いた．またデータセットは第 4 章 4.3 節と同様に TPC-H データセットを Scale Factor=1000 で生成し，PostgreSQL へとロードしたものを利用した．

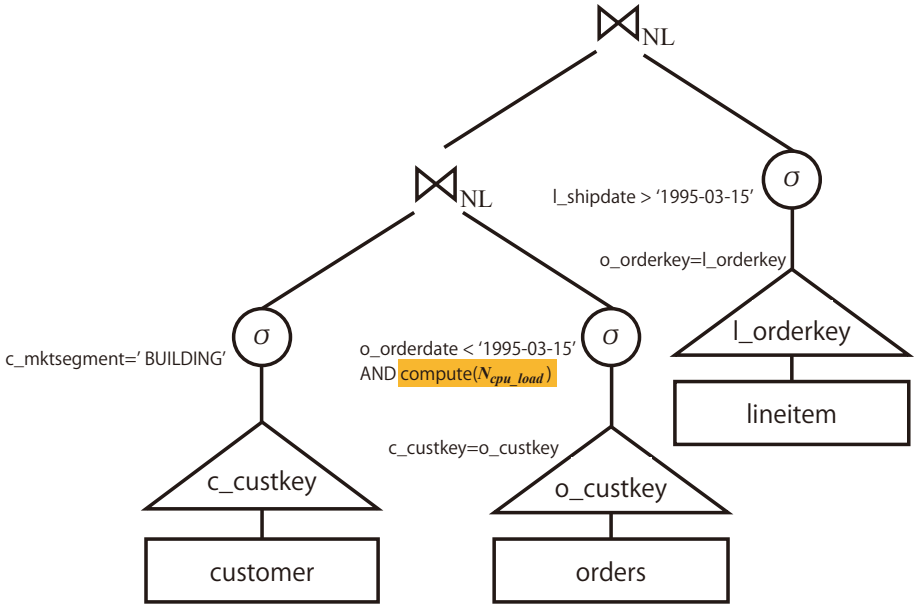


図 6.6　CPU 演算負荷が調整可能な Q.3 類似クエリ実行プラン．orders 表の各レコード選択条件評価において N_{cpu_load} 回の整数演算を行う．

```
1  CREATE FUNCTION compute(n integer)
2  RETURNS boolean AS $$
3  DECLARE
4      c integer := 0;
5  BEGIN
6      FOR i IN 1..n LOOP
7          c := c + 1;
8      END LOOP;
9      RETURN true;
10 END;
11 $$ LANGUAGE plpgsql;
```

図 6.7　CPU 演算負荷調整に用いる compute 関数の定義

6.3.1　TPC-H データセットを用いた性能評価

本実験では，クエリ実行における CPU 演算負荷を変更しながらクエリ実行時間を測定し，PgBoosterMC が有効にアウトオブオーダ型クエリ実行の並列演算性能を活用

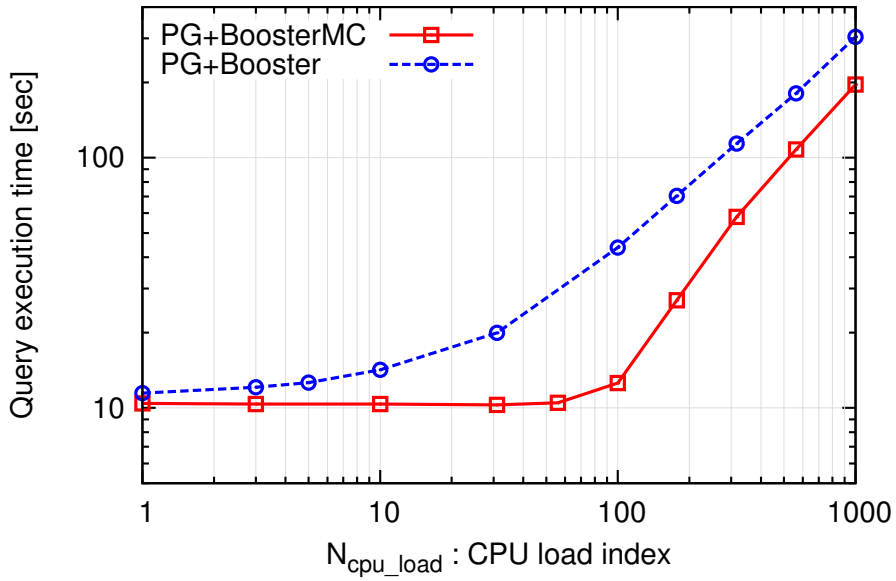


図 6.8　CPU 演算負荷 N_{cpu_load} とクエリ実行時間

できていることを検証した．測定用クエリには TPC-H Q.3 類似クエリを用いることとした．ただし，**図 6.6** のクエリ実行プランに示す orders 表の選択条件においてユーザ定義関数 compute（**図 6.7**）を呼び出し，N_{cpu_load} 回整数演算を繰り返すことで，クエリ実行における CPU 演算負荷の調節を行った．測定対象としては PgBooster を有効化した PostgreSQL（PG+Booster）と，PgBoosterMC を有効化した PostgreSQL（PG+BoosterMC）を利用した．

クエリ実行時間を**図 6.9** に示す．PostgreSQL においてユーザ定義関数呼び出しは比較的高価な処理であるため，$N_{cpu\_load} = 1$ において既に PG+Booster と PG+BoosterMC には 1.0 秒の実行時間の差が生じており，$N_{cpu\_load} = 100$ までは CPU 演算負荷が大きくなるほど実行時間の差が大きくなる．特に PG+BoosterMC では $N_{cpu\_load} = 56$ までは演算負荷の増加に対してほとんど実行時間が増加していないことがわかる．これは，入出力スループットがクエリ実行におけるボトルネックとなっている状況下では，CPU 演算資源に余裕がある限り PG+BoosterMC ではクエリ実行速度が低下しないためである．$N_{cpu\_load} \geq 56$ においては CPU 演算速度がボトルネックとなり始めるため，PG+BoosterMC においても N_{cpu_load} の増加に応じてクエリ実行時間は増加した．

クエリ実行時の平均入出力スループットを図 6.9 に示す．PG+Booster では，CPU 演算負荷が増加するに従い入出力スループットが低下した．これは，CPU 演算負荷が高

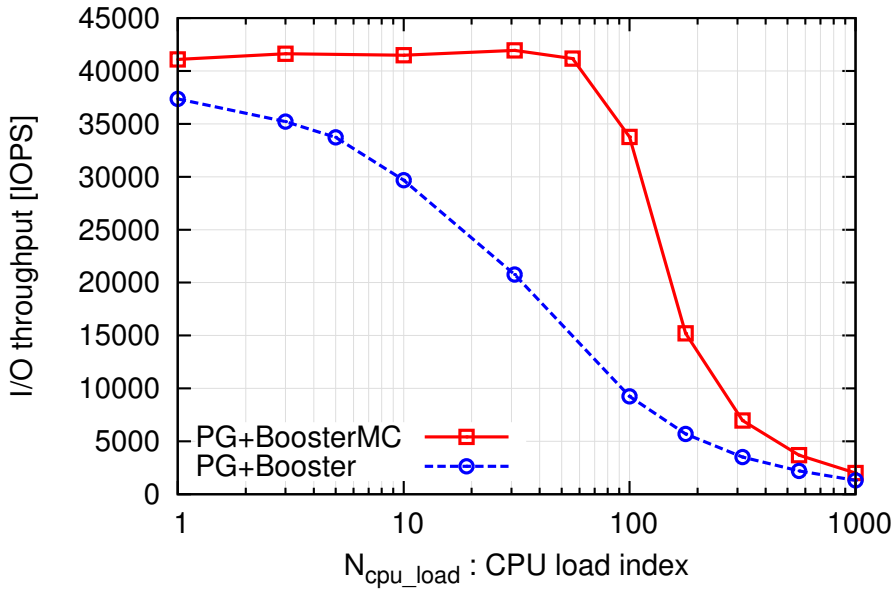


図 6.9　CPU 演算負荷 N_{cpu_load} と入出力スループット

いほどインオーダ型クエリ実行器が単位時間あたりに処理可能なデータ量は減少するため，これに歩調を合わせる形でアウトオブオーダ型クエリ実行における入出力スループットが制限されるためである．一方 PG+BoosterMC では，入出力がボトルネックである $N_{cpu\_load} \leq 56$ ではほぼ入出力スループットが低下せず，高効率に入出力帯域を活用可能であることがわかる．

以上の結果より，インオーダ型データベースエンジンに対してタプル供給を行うデータベースエンジン加速機構を用いることで，インオーダ型クエリ実行における逐次演算実行による性能律速を緩和し，CPU 演算負荷が高いクエリにおいても入出力帯域を高効率に活用可能であることが確認できた．

6.3.2　人の流れデータセットを用いた性能評価実験

実験用データセットとしては，東京大学空間情報科学研究センターより提供された人の流れデータセットおよび建物ポリゴンデータセットを利用した．人の流れデータセットはパーソントリップ調査[*1]結果に基づいて作成されたものであり，当該データセットは個

[*1] 調査対象世帯に調査票を送付して回答を収集することで，移動軌跡に加えて性別・年齢等の属性や，移動手段・移動目的等を含めて調査する方法

人識別子・時刻・位置に加えて，移動目的・移動手段等の属性が付加されたレコードから構成されており，調査対象の 36 万人それぞれについて，データを 1 分単位で補完したレコード 5.18 億件が格納されている．また建物ポリゴンデータセットは，建物の形状ポリゴンに対して建物の名称・区分等の属性が付加されたものであり，298 万件のレコードが格納されている．これらデータセットを PostgreSQL にデータロードし，実験に利用した．データロード後の容量は，人の流れデータセットが 95GB，建物ポリゴンデータセットが 1.0GB であった．

当該データセットにおいて，人の流れデータセットは `pflow` テーブルに格納されており，各レコードの位置に対して PostGIS を用いた R 木索引を生成した．R 木索引は平衡多分探索木の一種であり，索引対象の空間を階層的に分割して索引ノードへと割り当てることで索引付けを行う．各索引エントリは矩形 r とポインタ（下位層索引ノード，あるいはタプルのいずれかを指す）p の対から構成されており，矩形 r は当該エントリから到達可能なタプルは全てを包含する最小外接矩形となっている．同一階層に位置する相異なる索引エントリの矩形 $r1, r2$ は重複部分を持ちうるが，重複が多いほど検索対象領域を絞り込む効率が低下するため，1 回の R 木索引検索においてアクセスする索引ノード数は増加し，検索対象オブジェクトと矩形の交差判定演算回数は増加する．

ある 1 つの建物ポリゴンを検索対象とする `pflow` リレーションの R 木索引検索について，索引ノードへのアクセスを可視化した結果を**図 6.10** に示す．各平面は空間オブジェクトが配置される平面座標空間を表しており，索引ノードの各階層は異なる平面として示されている．平面上の矩形は各階層の索引エントリのうち，検索対象ポリゴンとの交差判定が行われたものを表す．橙色の矩形は交差すると判定され，当該エントリが指す下位の索引ノードあるいはタプルを取得したものであり，水色の矩形は交差しないと判定されたものである．また赤色の矢印は検索対象である建物ポリゴンの中心位置を示す．この図より，各階層において多くの矩形が重複しており，結果として無駄な矩形とポリゴンの交差判定が多数行われていることがわかる．このような R 木の索引検索においては，多数のページに対する入出力発行が行われることに加えて，入出力あたりの相対的な演算負荷が増加する．即ち，インオーダ型クエリ実行におけるプロセッサの演算速度が律速要因となり，先行的な入出力結果供給による行うデータベースエンジン加速機構による性能向上は制限されるため，タプル演算結果供給によるデータベースエンジン加速機構によって更なる性能向上が得られることが期待される．

評価実験に用いたクエリを**図 6.11** に示す．当該クエリは選択条件を満たす建物ポリゴンを取得し，当該建物ポリゴンを検索キーとして `pflow` リレーションの R 木索引検索を駆動するネステッドループ結合からなるクエリ実行プランによって実行される．

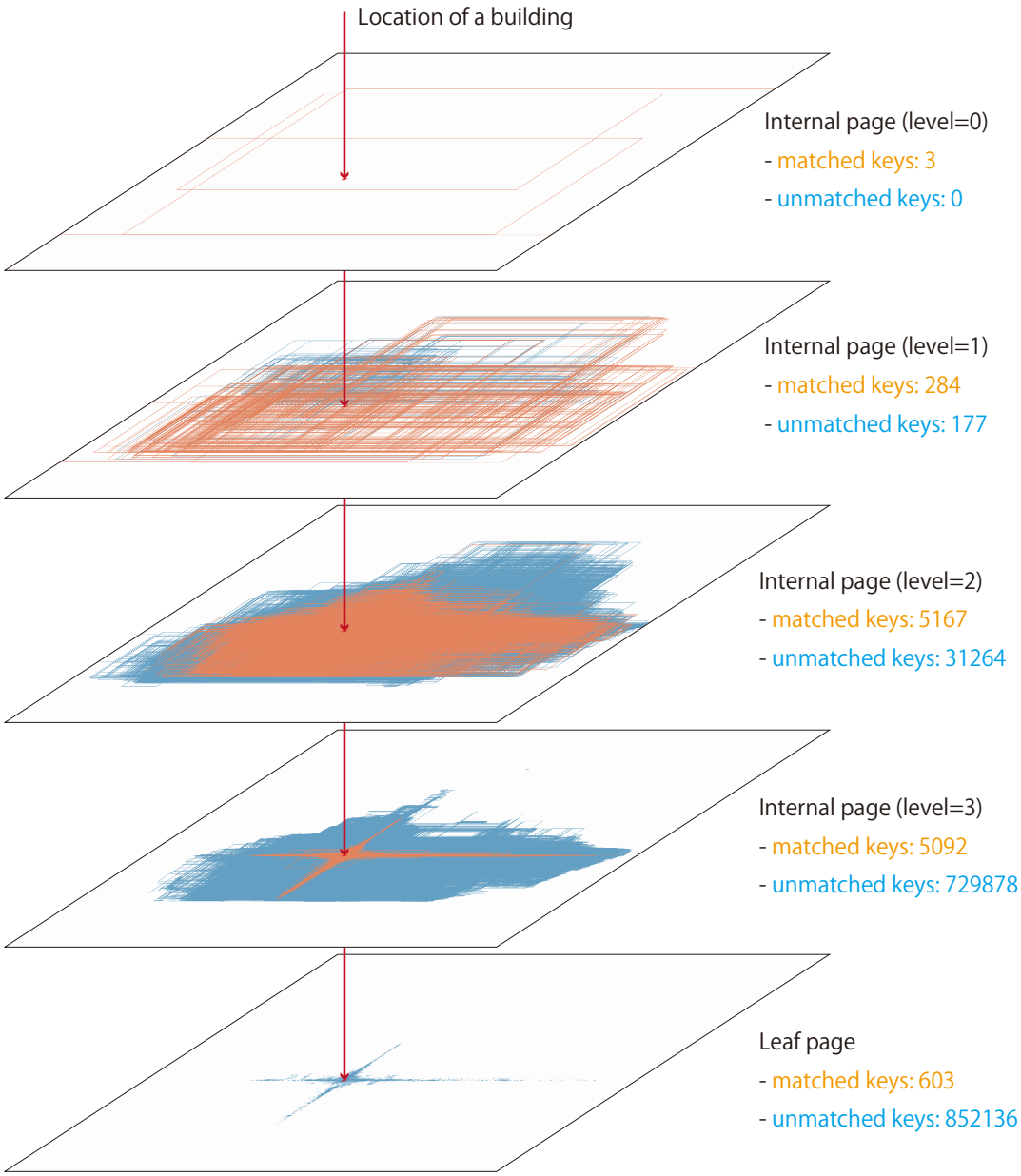


図 6.10　建物ポリゴン 1 件による R 木索引実行時の索引ページアクセスの様子.

通常版 PostgreSQL を用いた場合（PG），PostgreSQL と PgBooster を用いた場合（PG+Booster），PostgreSQL と PgBoosterMC を用いた場合（PG+BoosterMC）のそれぞれについて測定したクエリ実行時間を**図 6.12** に示す．PG の場合にはクエリ実行時間は 30,014 秒であったのに対し，PG+Booster では入出力が高速化されることでクエリ実行時間は 2,699 秒であり，PG に対して 11.1 倍の性能向上がみられた．これは評価

```
1  SELECT DISTINCT pid
2  FROM pflow p, building b
3  WHERE b.atrcode=1200
4    AND ST_Within(p.geom, b.polygon)
5    AND PDATE >= '1998-01-01 09:00:00'
6    AND PDATE < '1998-01-01 10:00:00'
7    AND t.SEX=2 AND t.DATUM=97;
```

図 6.11　性能評価用クエリ

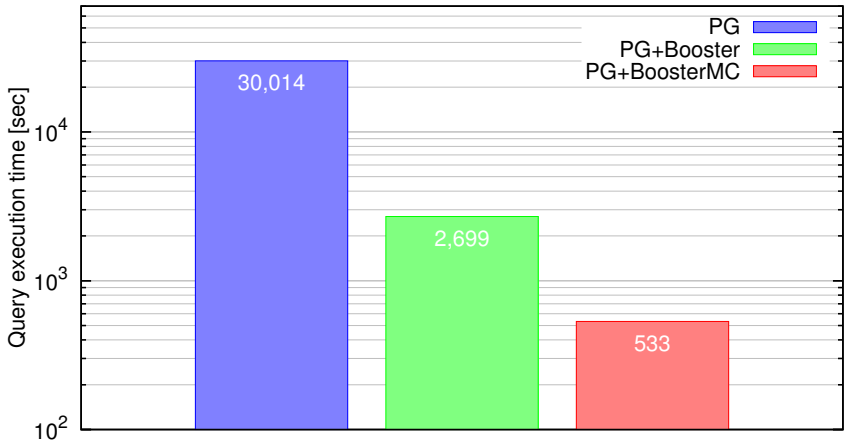


図 6.12　人の流れデータセットにおける評価クエリ実行時間

用クエリにおいては R 木索引探索の CPU 演算負荷が高く，インオーダ型クエリ実行器における逐次演算実行がボトルネックとなっているためである．PG+Booster に対し，PG+BoosterMC ではタプル供給を受けることで複数プロセッサコアの演算性能を活用することで実行時間は 533 秒，PG に対し 56.3 倍，PG+Booster に対し 5.1 倍の性能向上が確認できた．

PgBoosterMC に対して割り当てるプロセッサコア数ごとのクエリ実行時間を**図 6.13** に示す．プロセッサコア数が 8 以下のときには，プロセッサコア数が多いほどクエリ実行時間は短くなったが，プロセッサコア数が 10 でも実行時間はほぼ変わらず，11 以上ではやや実行時間が増加する結果となった．クエリ実行時の平均 IOPS を**図 6.14** に示す．このグラフより，クエリ実行時間が短いほど入出力スループットが高いことがわかる．また CPU 利用率を**図 6.15** に示す．このグラフより，クエリ実行時間が短くなるほどユーザ

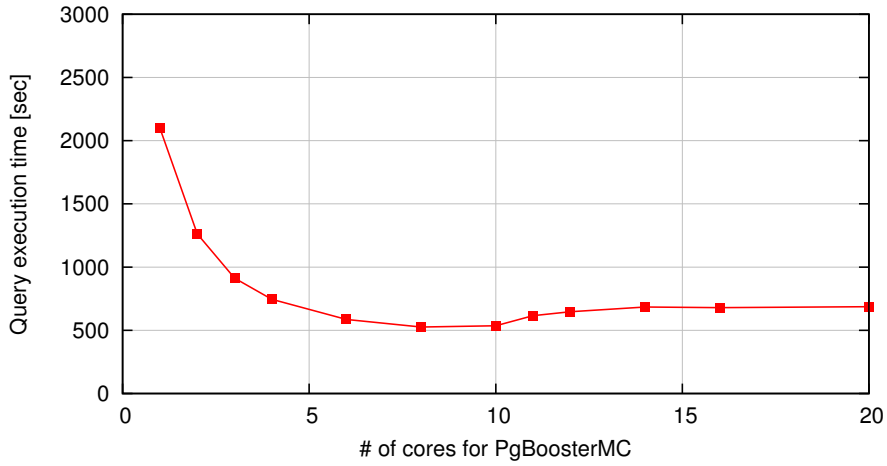


図 6.13　利用プロセッサコア数とクエリ実行時間

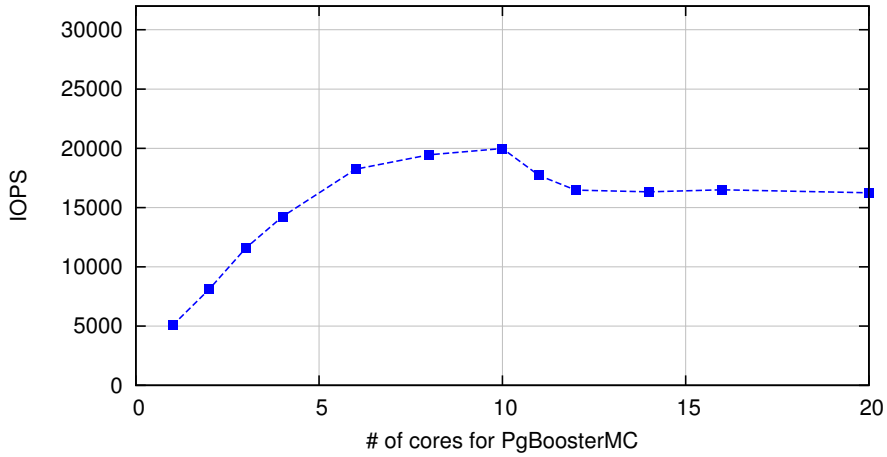


図 6.14　利用プロセッサコア数と入出力スループット

空間の CPU 利用率が高く，またプロセッサコア数を 11 以上の場合にはコア数増加に応じてユーザ空間 CPU 利用率は増加せず，カーネル空間の CPU 利用率が増加していることがわかる．

本実験に用いたデータセットにおいては，図 6.10 に示すように R 木の索引ノード矩形の空間的重複が多く，クエリ実行においては非常に多くのポリゴン交差判定演算が行われる．そのため，本評価クエリの実行は CPU 演算速度が最大の律速要因であり，クエリ実行に利用可能な CPU 演算資源が多いほど，それに応じて入出力帯域を活用可能であることがわかる．ただし，プロセッサコア数が 11 以上の場合においては複数コア間のコヒー

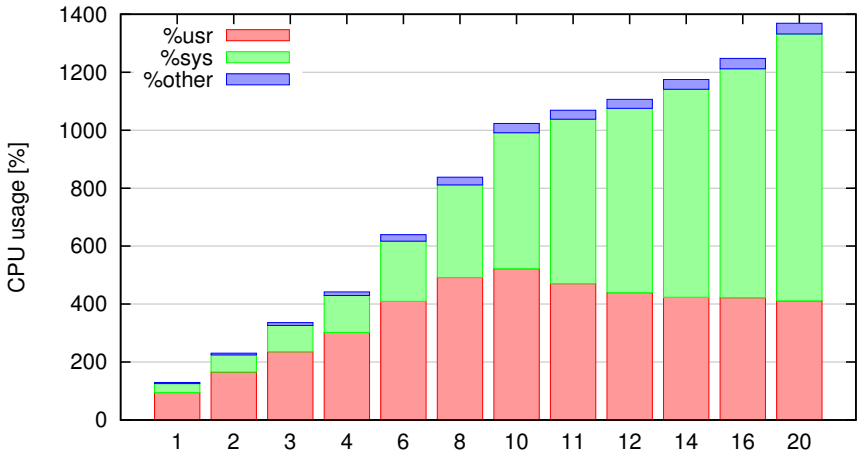


図 6.15　利用プロセッサコア数と CPU 利用率

レンシオーバヘッドが顕在化し，コア数増加がクエリ実行に利用可能な CPU 演算資源増加につながらないため，結果として性能低下を招くことが確認された．

以上の結果より，演算負荷の高い R 木探索を伴うクエリ実行において，複数プロセッサコアの演算性能を活用可能である PgBoosterMC は，入出力の高速化のみを行う PgBooster を更に高速化可能であることが確認された．このように，実データを用いた評価実験においてもタプル供給機能を有するデータベースエンジン加速機構が有効に機能し，クエリ実行を高速化する効果をもたらすことが確認された．

第 7 章

データベースエンジン加速機構を考慮したクエリ最適化

関係データベースシステムの実用性を考える上で，実行コストが最小となるクエリ実行プランを生成するクエリ最適化器は極めて重要な役割を果たす．関係データベースシステムに対するクエリは，その結果データが満たすべき制約条件によって記述される．制約条件を満たす結果を得るための実行手順は必ずしも 1 通りに限定されない．例えばリレーションのデータを取得するアクセスパスとしては，リレーションの大部分を選択する場合には全件走査，リレーションのごく一部を選択する場合には索引走査を利用することが一般的である．また複数リレーションの結合方式に関しては，外表であるリレーションのタプル毎に内表の走査が駆動されるネステッドループ結合や，外表および内表の走査結果をそれぞれハッシュ表に格納し突合するハッシュ結合など，複数の結合方式が広く用いられている．また複数リレーションを結合する際には，結合を実施する順序によってクエリ実行の計算量は大きく左右される．クエリ最適化器は，このような多岐にわたる実行手順の組み合わせから成るクエリ実行プラン候補の中から，コスト最小であるクエリ実行プランを選択しなければならない．

クエリ最適化器におけるクエリ実行プラン生成は，コストモデルに基づいて各アクセスパスや結合に要する実行コストを見積もることで行われる．これまでに様々なクエリ最適化のためのコストモデルが提案されてきているが，これらは基本的にインオーダ型データベースエンジンにおけるクエリ実行のコストをモデル化したものである．インオーダ型データベースエンジンと比べて，アウトオブオーダ型データベースエンジンは選択性を有するクエリ実行において大幅な高速性を示すため，データベースエンジン加速機構を組み込んだインオーダ型データベースエンジンにおいては，異なるクエリ実行プラン同士のコ

表 7.1　索引検索コストモデル諸元

c_r	データベース 1 ページ分のランダム入出力コスト
c_t	1 タプル処理に掛かる CPU コスト
$N(R)$	リレーション R に含まれるタプル数
$P(R)$	リレーション R に含まれるページ数
σ_R	リレーション R の索引検索による選択率
B	バッファプール容量

ストバランスが大きく変化すると考えられる．即ち，従前のコストモデルでは必ずしも最適なクエリ実行プランが選択されるとは限らず，データベースエンジン加速機構を用いた場合のコストモデルを考慮した上でクエリ最適化を行う必要がある．

またクエリ実行のコストモデルは，当該クエリ実行を実施する計算機環境における入出力処理や計算処理の単位オペレーションコストをパラメータとして構築される．最適なクエリ実行プランを選択するためには，このコストパラメータを適切な値に設定することが求められる．コストパラメータの最適値は計算機環境に依存して変化するため [110–112]，実行環境に適応してコストモデルを較正しなければ正確なクエリ実行プラン選択を行うことはできない．

本章では，まずデータベースエンジン加速機構を用いる場合のコストモデルを提案する．また当該コストモデルを利用した正確なプラン選択を可能とするため，コストモデルの自動較正手法を提案する．そして，試験環境において提案手法を用いてコストパラメータの自動較正を実施した上で，提案するコストモデルによるクエリ実行プラン選択の精度を評価する．

7.1　データベースエンジン加速機構を用いる場合のコストモデル

アウトオブオーダ型データベースエンジンは，従来のインオーダ型データベースエンジンと比べて，選択性を有する分析的クエリの実行において大幅な高速性を示す．このような分析クエリの実行においては，典型的に索引走査とネステッドループ結合からなるクエリ実行プランが用いられる．このため，本論文では索引走査・ネステッドループ結合からなるクエリ実行プランにおけるコストモデルについて議論する．

まず，リレーション R の索引検索のインオーダ型データベースエンジンにおけるコストモデルについて考える．コストモデルに関わる性能諸元を**表 7.1** に示す．ここで，リレーション R の索引はクラスタ化されておらず，索引検索における入出力はランダムアクセスであるものとする．索引検索に要する CPU コストは，索引によって選択されるタプル数に 1 タプルあたりの CPU コストを乗じたものであるため

$$C_{\mathrm{IS\_CPU}}(R) = c_t \sigma_R N(R) \tag{7.1}$$

である．また索引検索に要する入出力コストは入出力するデータベースページ数にランダムアクセスのコストパラメータを乗じたものである．索引検索において処理されるタプルのうち一定数は同一ページに含まれうるため，いくばくかのタプル取得の際には必要なページがすでにバッファプール上に存在する場合がある．このような場合を考慮し，索引検索において入出力するページ数の見積りに関しては文献 [113–116] など様々な手法が提案されているが，ここでは LRU バッファ置換ポリシにおいて良い見積りを与えることで知られており，PostgreSQL の索引検索コスト見積りにも用いられる Mackert らによる方法 [117] を採用し，当該見積りを関数 Y によって記す．このとき，索引検索に要する入出力コストは次のようになる．

$$C_{\mathrm{IS\_IO}}(R) = c_r Y(N(R), P(R), \sigma_R, B) \tag{7.2}$$

$$Y(N,P,\sigma,B) \equiv \begin{cases} \min\left(\dfrac{2PN\sigma}{2P+N\sigma}, P\right) & (P \le B) \\ \dfrac{2PN\sigma}{2P+N\sigma} & \left(P > B \wedge \sigma \le \frac{2PB}{N(2P-B)}\right) \\ B + \left(N\sigma - \dfrac{2PB}{2P-B}\right)\dfrac{P-B}{P} & \left(P > B \wedge \sigma > \frac{2PB}{N(2P-B)}\right) \end{cases} \tag{7.3}$$

よって，インオーダ型データベースエンジンにおけるリレーション R の索引検索コストは次のようになる．

$$\begin{aligned} C_{\mathrm{IS-IN}}(R) =& C_{\mathrm{IS\_CPU}}(R) + C_{\mathrm{IS\_IO}}(R) \\ & c_t \sigma_R N(R) + c_r Y(N(R), P(R), \sigma_R, B) \end{aligned} \tag{7.4}$$

リレーション R の索引検索をデータベースエンジン加速機構を用いて実行する場合，ランダム入出力が高多重に非同期発行される．ストレージ帯域を飽和させるに足る多重度で入出力を発行した際の，相対的なランダム入出力スループットの向上率を L としたとき，ランダム入出力に要する時間は実質的に $1/L$ となるため，データベースエンジン加速機構を用いた場合のコストは次のようになる．

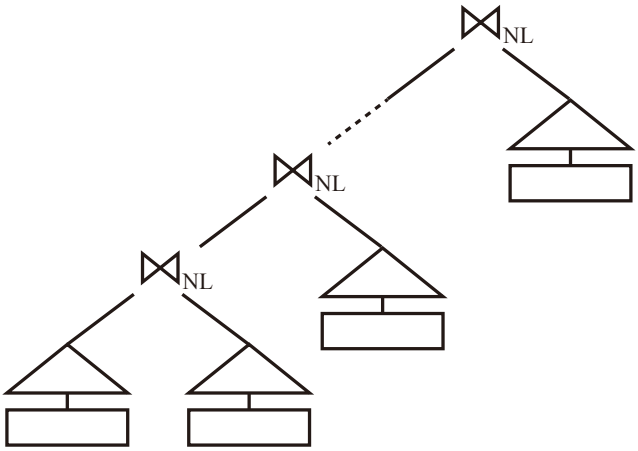


図 7.1　3 つ以上のリレーションをネステッドループ結合する left-deep クエリ実行プラン

表 7.2　ネステッドループ結合 $R \bowtie S$ のコストモデル諸元

$C_{\mathrm{IN}}(R)$	外表 R のインオーダ型データベースエンジンにおける実行コスト
$C_{\mathrm{OoO}}(R)$	データベースエンジン加速機構を用いた場合の外表 R の実行コスト
$n(R)$	外表 R の出力タプル数
σ_S	内表 S の索引検索 1 回における平均選択率

$$C_{\mathrm{IS-OoO}}(R) = C_{\mathrm{IS\_CPU}}(R) + \frac{1}{L} C_{\mathrm{IS\_IO}}(R) \\ c_t \sigma_R N(R) + \frac{c_r}{L} Y(N(R), P(R), \sigma_R, B) \tag{7.5}$$

次に，ネステッドループ結合のコストモデルについて議論する．ネステッドループ結合では，外表であるリレーション R よりタプル r を取得し，タプル r の値に基づいて内表であるリレーション S を検索してタプル s を取得し，タプル r と s を結合したものをその出力とする．タプル r ににについて該当するタプル s が無くなるまで内表の検索を繰り返した後，次なる外表のタプル r を新たに取得し，再び内表検索を実行する，という手順を繰り返す．ネステッドループ結合により 3 つ以上のリレーションの索引検索を結合する場合，**図 7.1** に示すように外表にネステッドループ結合を，内表に索引検索を持つ left-deep なクエリ実行プランを生成することが一般的であるため，ここでは left-deep クエリ実行プランに関するコストモデルについて論じる．

外表 R（リレーションの索引検索あるいはネステッドループ結合）と内表 S（リレー

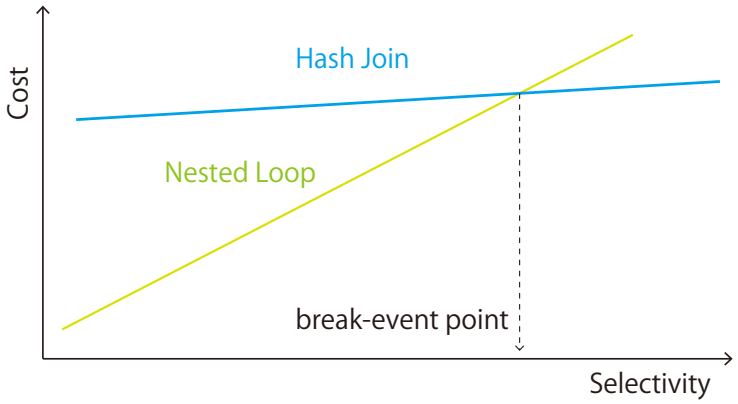


図 7.2　選択率を軸としたハッシュ結合，ネステッドループ結合のコスト曲線概形

ションの索引検索）のネステッドループ結合のコストモデルに関する諸元を**表 7.2** に示す．ネステッドループ結合では，内表検索は $n(R)$ 回だけ繰り返し実行されるため，実質的に選択率 $n(R)\sigma_S$ の索引検索を S に対して 1 回実行する場合と同等となる．よってインオーダ型データベースエンジンにおけるコストは式 (7.4) より次のようになる．

$$C_{\mathrm{NLJ-IN}}(R,S) = C_{\mathrm{IN}}(R) + c_t n(R)\sigma_S N(S) \quad + c_r Y(N(S), P(S), n(R)\sigma_S, B) \tag{7.6}$$

また，データベースエンジン加速機構を用いる場合には次のようになる．

$$\begin{aligned} C_{\mathrm{NLJ-OoO}}(R,S) = & C_{\mathrm{OoO}}(R) + c_t n(R)\sigma_S N(S) \\ & + \frac{c_r}{L} Y(N(S), P(S), n(R)\sigma_S, B) \end{aligned} \tag{7.7}$$

7.2　コストパラメータの自動較正

前節のコストモデルにおいて用いられたコストパラメータ c_r, c_t や L などは，計算機環境の有する処理性能にあわせて最適な値を設定しなければ，精度の良いクエリプラン選択を行うことはできない．特に選択性を有する分析的クエリに関しては，典型的にはネステッドループ結合・索引検索からなるクエリ実行プランと，ハッシュ結合・全件走査からなるクエリ実行プランによって実行される場合が多い．ネステッドループ結合・索引検索ではクエリ実行コストは選択率に概ね比例する形であるのに対し，ハッシュ結合・全件走査では全件走査を行う入出力コストが一定量を占め，それに選択率に応じたタプル演算の

コストが加わる形となる．すなわち，両者の実行コストは**図 7.2** のように示すことができる．このようなコストモデルからわかるように，選択率が一定数以上の場合にはハッシュ結合を選択したほうがよく，選択率一定以下の場合にはネステッドループ結合を用いたほうが良いということである．両者のコストバランスが入れ替わる損益分岐点が，実際のクエリ実行時間における選択率の損益分岐点と一致していることで，選択率が小さい領域における最適なコスト選択が可能となる．

このような損益分岐点が一致するよう．コストモデルにおけるコストパラメータを毎回手動で実施することは難しい．そこで本論文では，コストパラメータ 1 変数を実測を用いて自動較正を行う手法を提案する．この方式を既存のインオーダ型データベースエンジンにおけるコストモデル較正に利用し，クエリプラン選択の損益分岐点予測精度向上を図る．また当該手法を用いてデータベースエンジン加速機構による入出力スループットの相対的向上率 L の値を求めることで，データベースエンジン加速機構を用いた場合も含めて最適なプラン選択がなされるか評価試験を行う．

7.3 ランダム入出力コストパラメータの自動較正

本論文において対象とする入出力インテンシブなデータ分析クエリに関しては，CPU 処理コストに比して入出力コストが極めて大きいという特徴を有する．例えば第 4 章において性能評価に用いた TPC-H Q.3 に関しては，ネステッドループ結合を用いる場合にはランダム入出力コストが大部分を占め，ハッシュ結合を用いる場合にはシーケンシャル入出力コストが大部分を占める．そのため，クエリ選択の精度向上においては，入出力コストパラメータの較正を行うことが最も効果が大きいといえる．入出力コストパラメータに関しては，シーケンシャル入出力のコスト c_s とランダム入出力のコスト c_r の 2 つのパラメータが用いられるが，これらはあくまで 1 ページ入出力に要するコストの相対値であるため，片側のコストパラメータを固定し，もう片側のコストパラメータを適切な比率の値に設定すればよい．ハッシュ結合とネステッドループ結合のクエリ実行プラン選択を考える場合，ハッシュ結合のコストは主にシーケンシャル入出力コスト c_s によって決定され，ネステッドループ結合のコストは主にランダム入出力のコスト c_r によって決定される．ここでは，ランダム入出力のコスト c_r を自動較正することにより，クエリ実行プラン選択の精度向上を図ることとする．

7.3.1 問題設定

選択率が σ の時のネステッドループ結合によるクエリ実行プランのコストモデルを $C_{\mathrm{NLJ}}(c_r|\sigma)$，ハッシュ結合によるクエリ実行プランのコストモデルを $C_{\mathrm{HJ}}(c_s|\sigma)$ とする．ここで入出力コストに関して，$C_{\mathrm{NLJ}}(c_r|\sigma)$ はランダム入出力のコストのみを含み，$C_{\mathrm{HJ}}(c_s|\sigma)$ はシーケンシャル入出力のコストのみを含むものとする．σ が十分小さい時には $C_{\mathrm{NLJ}}(c_r|\sigma) < C_{\mathrm{HJ}}(c_s|\sigma)$ が成り立ち，σ が十分大きい時には $C_{\mathrm{NLJ}}(c_r|\sigma) > C_{\mathrm{HJ}}(c_s|\sigma)$ が成り立ち，$C_{\mathrm{NLJ}}(c_r|\sigma) = C_{\mathrm{HJ}}(c_s|\sigma)$ なる解 σ はただひとつのみ存在するものとする．また $T_{\mathrm{NLJ}}(\sigma)$ を選択率 σ のときのネステッドループ結合クエリ実行プランの実際のクエリ実行時間測定値，$T_{\mathrm{HJ}}(\sigma)$ をハッシュ結合クエリ実行プランの実際の実行時間測定値とする．ただし，実行時間測定値はそれぞれ最大で $\delta T_{\mathrm{NLJ}}(\sigma), \delta T_{\mathrm{HJ}}(\sigma)$ だけ測定の度に値がばらつくことを想定する．

7.3.2 コストパラメータの自動較正アルゴリズム

Algorithm 2 ランダム入出力のコストパラメータ c_r 自動較正アルゴリズム

$\sigma_{bep} \leftarrow$ SolveExecTimeBEP()
$c_r \leftarrow$ SolveRandomCost(σ_{bep})

ランダム入出力のコストパラメータ c_r を自動較正するアルゴリズムの概要を**アルゴリズム 2** に示す．このアルゴリズムでは，まずハッシュ結合とネステッドループ結合の損益分岐点（BEP：break-event point）である選択率 σ_{bep} を SolveExecTimeBEP により求める．そして，コストモデルにおける損益分岐点が σ_{bep} となるランダム入出力コストパラメータ c_r を SolveRandomCost(σ_{bep}) により求める．

SolveExecTimeBEP における求解手順をアルゴリズム 3 に示す．当該アルゴリズムでは，損益分岐点 σ_{bep} よりも明らかに小さい選択率 σ_{low} と，明らかに大きい選択率 σ_{up} を選択し，クエリ実行時間を実際に測定しながら領域 $[\sigma_{low}, \sigma_{up}]$ を二分探索によって絞り込む．ただし，クエリ実行時間は実行毎に測定誤差が生じるため，損益分岐点 σ_{bep} 付近ではハッシュ結合とネステッドループ結合のいずれのクエリ実行プランの実行時間が短いかを判別することが不可能となる．そこで，測定誤差を加味しても実行時間が短いプランを判別可能な範囲において領域 $[\sigma_{low}, \sigma_{up}]$ を絞り込んだ上で，ハッシュ結合の実行時間およびネステッドループ結合の実行時間を内挿することにより σ_{bep} を求めることとした．

Algorithm 3 実行時間測定による選択率の損益分岐点求解アルゴリズム

procedure SolveExecTimeBEP(a)

$\sigma_{low} \leftarrow$ pick σ s.t. $T_{\mathrm{NLJ}}(\sigma) \ll T_{\mathrm{HJ}}(\sigma)$

$\sigma_{up} \leftarrow$ pick σ s.t. $T_{\mathrm{NLJ}}(\sigma) \gg T_{\mathrm{HJ}}(\sigma)$

$t_{\mathrm{NLJ\_}low} = T_{\mathrm{NLJ}}(\sigma_{low})$

$t_{\mathrm{NLJ\_}up} = T_{\mathrm{NLJ}}(\sigma_{up})$

$t_{\mathrm{HJ\_}low} = T_{\mathrm{HJ}}(\sigma_{low})$

$t_{\mathrm{HJ\_}up} = T_{\mathrm{HJ}}(\sigma_{up})$

$Loop \leftarrow true$

while $Loop = true$ **do**

$\sigma' \leftarrow (\sigma_{low} + \sigma_{up})/2$

$t_{\mathrm{NLJ}}, \delta t_{\mathrm{NLJ}} \leftarrow T_{\mathrm{NLJ}}(\sigma'), \delta T_{\mathrm{NLJ}}(\sigma')$ ▷ NLJ measurement

$t_{\mathrm{HJ}}, \delta t_{\mathrm{HJ}} \leftarrow T_{\mathrm{HJ}}(\sigma'), \delta T_{\mathrm{HJ}}(\sigma')$ ▷ HJ measurement

if $t_{\mathrm{NLJ}} + \delta t_{\mathrm{NLJ}} < t_{\mathrm{HJ}} - \delta t_{\mathrm{HJ}}$ **then**

$\sigma_{low} \leftarrow \sigma'$

$t_{\mathrm{NLJ\_}low} \leftarrow t_{\mathrm{NLJ}}$

$t_{\mathrm{HJ\_}low} \leftarrow t_{\mathrm{HJ}}$

else if $t_{\mathrm{NLJ}} - \delta t_{\mathrm{NLJ}} > t_{\mathrm{HJ}} + \delta t_{\mathrm{HJ}}$ **then**

$\sigma_{up} \leftarrow \sigma'$

$t_{\mathrm{NLJ\_}up} \leftarrow t_{\mathrm{NLJ}}$

$t_{\mathrm{HJ\_}up} \leftarrow t_{\mathrm{HJ}}$

else

$Loop \leftarrow false$

end if

end while

$a \leftarrow (t_{\mathrm{NLJ\_}up} - t_{\mathrm{NLJ\_}low})/(\sigma_{up} - \sigma_{low})$

$b \leftarrow (t_{\mathrm{HJ\_}up} - t_{\mathrm{HJ\_}low})/(\sigma_{up} - \sigma_{low})$

$\sigma_{bep} \leftarrow \sigma_{low} - (t_{\mathrm{NLJ\_}low} - t_{\mathrm{HJ\_}low})/(a - b)$ ▷ interpolation

return σ_{bep}

end procedure

SolveRandomCost は，コストモデルによる損益分岐点が SolveExecTimeBEP により計算される損益分岐点 σ_{bep} と一致するときのランダム入出力コストパラメータ c_r を求めるアルゴリズムである．コストモデルによる損益分岐点を求めるアルゴリズムを SolveCostModelBEP とすると，これにより損益分岐点の見積り誤差 $\Delta\sigma_{bep}(c_r)$ は次のように与えられる．

$$\Delta\sigma_{bep}(c_r) \leftarrow \text{SolveCostModelBEP}(C_{\text{NLJ}}(c_r|\sigma), C_{\text{NLJ}}(c_s|\sigma)) \tag{7.8}$$

この際，SolveCostModelBEP は非線形方程式の求解アルゴリズムによって構築できる．本提案手法においては，ニュートン法を用いることとした．このように $\Delta\sigma_{bep}(c_r)$ が与えられると，SolveRandomCost は $\Delta\sigma_{bep}(c_r) = 0$ なる方程式の解である c_r を求める問題となるため，非線形方程式の求解アルゴリズムによって求めることができる．これに関しても，本提案手法ではニュートン法を用いることとした．

ここまで，コストパラメータ c_r の較正手法としてその具体的なアルゴリズムを説明してきたが，これは解が一意に定まるコストパラメータに関していずれも同様に適用することができる．そのため，データベースエンジン加速機構を用いた場合のコストモデルにおける，入出力スループットの相対的向上率 L に関しても同様の手順で求めることが可能である．

7.4 コストモデル評価実験

本章において示したコストモデルにより適切なクエリ実行プラン選択がなされるかを評価するために評価実験を行った．実験に際しては，データベースエンジン加速機構を用いた場合のコストモデルを用いてクエリ実行プラン生成を行うコードパスを PostgreSQL へと追加し，PgBooster が有効化されている場合のみ当該コードパスが有効化されるよう実装を行った．

7.4.1 実験環境

評価実験に用いた計算機環境の諸元を**表 7.3** に示す．本実験環境はデータベース用領域として 24 台の磁気ディスクドライブを搭載する 2U サーバから構成される．各ドライブは SAS インターフェースにより内蔵 RAID コントローラに接続され，RAID6 ボリューム（22D+2P）により 1 つの論理ユニットを編成する．当該論理ユニットにおいて xfs フォーマットにてファイルシステムを作成し，データベース格納領域とした．実験用データベースとしては，TPC-H ベンチマークのデータセットを `dbgen` **により** scale

表 7.3　コストモデル評価実験環境の諸元

Processor	2x Intel Xeon E5-2680 2.70GHz (2p16c)
Memory	64GB DDR2 DIMMs
Storage (OS)	2x 15krpm 300GB SAS HDDs
Storage (database)	24x 10krpm 900GB SAS HDDs
RAID controller	Dell PERC H710P
OS	CentOS 5.8 (Linux 2.6.18)

Default c_r	Calibrated c_r
4.0	56.5

表 7.4　自動較正前後のランダム入出力コスト c_r

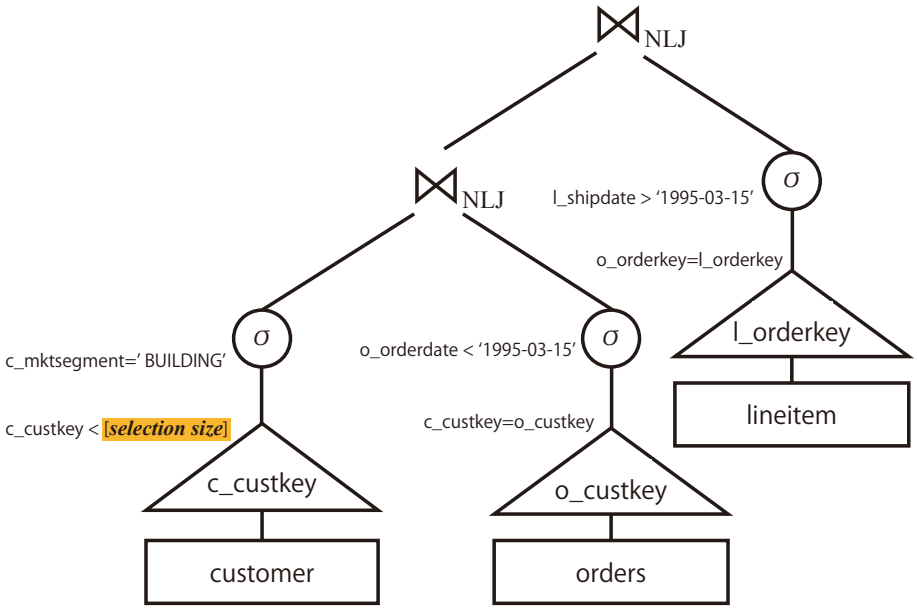
factor=100 で生成したものを PostgreSQL にロードすることで生成した．ロード後のデータベース容量は総じて 246GB であった．

7.4.2　ランダム入出力コストパラメータ自動較正とネステッドループ結合・ハッシュ結合選択精度

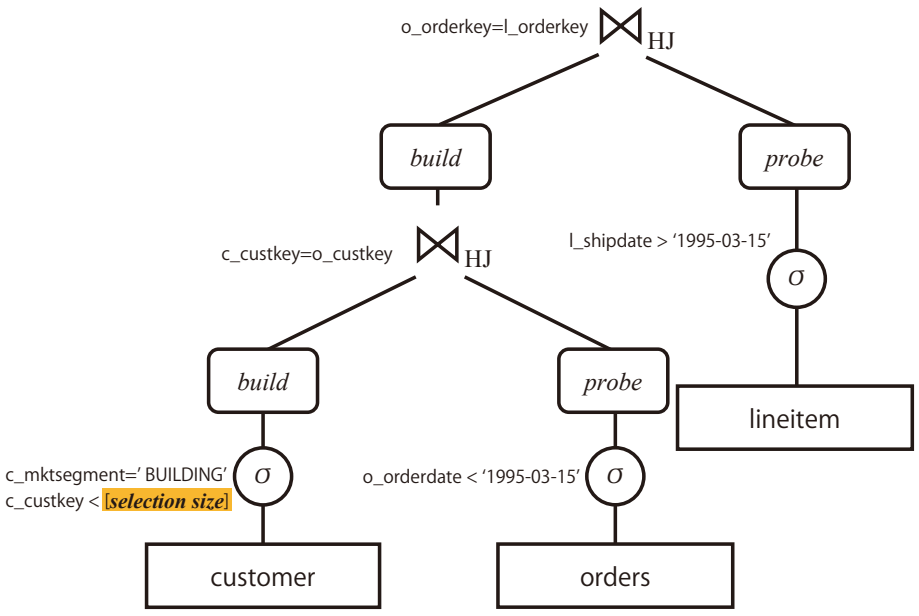
第 7.1 節にて示したように，データベースエンジン加速機構を用いた場合のコストモデルでは，コストパラメータとしてランダム入出力コスト c_r と多重非同期入出力発行による相対的入出力スループット向上率 L の 2 つに依存している．そのため，まずデータベースエンジン加速機構を利用せずに，インオーダ型データベースエンジンにおいてハッシュ結合方式とネステッドループ結合方式を用いたクエリ実行において，ランダム入出力コスト c_r の較正を行った．自動較正における較正用クエリとしては，アウトオブオーダ型データベースエンジンの対象とする典型的なワークロードである TPC-H Q.3 類似クエリ利用することとした．当該クエリは，customer 表における選択数の変更によって選択率を可変とし，**図 7.3** に示すネステッドループ結合とハッシュ結合のクエリ実行プランをそれぞれ実行することで，ランダム入出力コスト c_r の較正を行う．

自動較正の結果を**表 7.6** に示す．PostgreSQL のデフォルト値は $c_r = 4.0$ であったのに対して，自動較正を適用した結果 $c_r = 56.5$ となった．

次いで，自動較正によって得られた $c_r = 56.5$ を用いた際の，ネステッドループ結合と

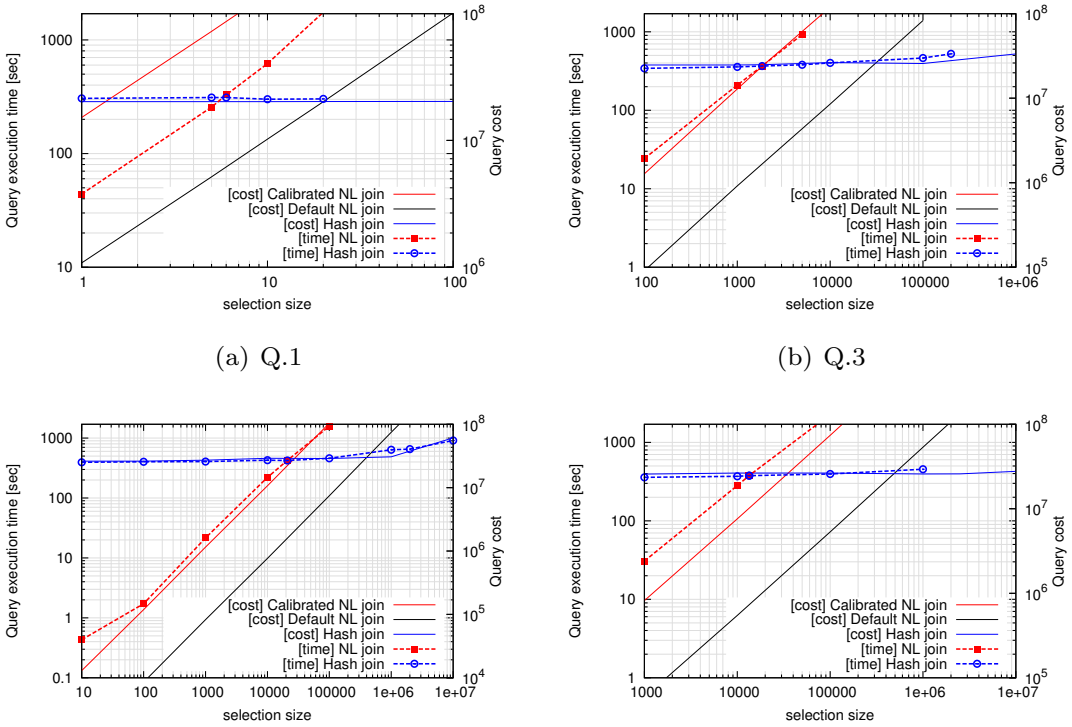


(a) ネステッドループ結合プラン



(b) ハッシュ結合プラン

図 7.3　ランダム入出力コスト c_r の自動較正用クエリプラン



(a) Q.1　(b) Q.3

(c) Q.7　(d) Q.8

図 7.4　クエリ実行時間と各クエリ実行プランのコスト曲線

表 7.5　各クエリの実行時間における損益分岐点とコスト見積りによる損益分岐点

クエリ	実行時間 BEP	コスト曲線 BEP（較正前）	コスト曲線 BEP（較正後）
Q.1	6	20 (誤差 +233%)	1 (誤差 -83%)
Q.3	1,930	31,430 (誤差 +1,528%)	1,932 (誤差 +0.1%)
Q.7	20,942	393,235 (誤差 +1,778%)	27,661 (誤差 +32%)
Q.8	13,406	489,593 (誤差 +3,552%)	35,650 (誤差 +165%)

ハッシュ結合のプラン選択精度について測定を行った．測定対象クエリは，自動較正に用いた TPC-H Q.3 に加えて，TPC-H Q.1，Q.7，Q.8 を用いた．これらのクエリは，いずれも図 7.3 に示すように最も外側の表における選択数を変更することで，選択率調整を行った．

結果を**図 7.4** に示す．グラフにおいて [cost] と示された曲線はクエリ最適化器によるコスト見積り値を，[time] と示された曲線はクエリ実行時間を表す．本実験では，

Calibrated L
87.0

表 7.6　自動較正後の入出力スループット向上率 L

PostgreSQL におけるネステッドループ結合のコストモデルと，ハッシュ結合のコストモデルによりコスト見積りを行っている．

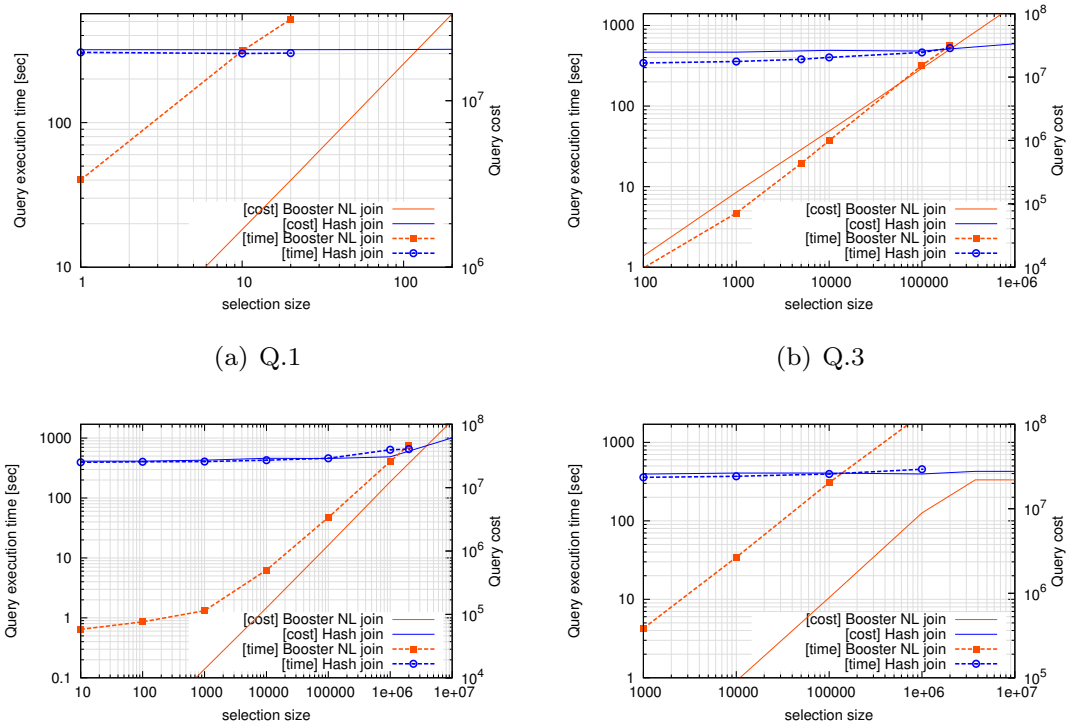
較正用クエリとして用いた Q.3 に関しては，較正前にはコスト見積りによる損益分岐点と実行時間の損益分岐点の誤差が +1,528% であったのが，較正後には誤差 +0.1% まで見積り精度が改善された．また Q.7，8 についても較正前は 1,000% 以上の損益分岐点見積り誤差が生じていたが，較正によって大幅に精度が改善されたことがわかる．一方，Q.1 に関しては較正によりネステッドループ結合のコスト高く見積もられすぎる結果となった．これは，Q.1 の索引検索キーとして用いた l_shipdate カラムが，二次索引であるにも関わらず実際にはクラスタ化索引に近い状態となっており，入出力のシーケンシャル性が高く，見積りに反して実行が高速であることに起因する．索引のデータ編成に関するより正確な統計情報がクエリ最適化器から利用することができれば，そのシーケンシャル性によってランダム入出力コストが一部シーケンシャル入出力コストに転化されるため，較正後の損益分岐点予測値はより精度が向上すると期待される．

7.4.3　入出力スループット向上率 L の自動較正とデータベースエンジン加速機構を用いたネステッドループ結合・ハッシュ結合選択精度

前述の実験において算出したランダム入出力コスト $c_r = 56.5$ を採用した上で，データベースエンジン加速機構を用いた場合のネステッドループ結合コストモデルにおけるパラメタ L の自動較正を行い，当該ネステッドループ結合とハッシュ結合の損益分岐点の予測精度の評価を行った．

L の自動較正により，$L = 87.0$ という値が得られた．当該パラメタは PostgreSQL に対して新たに追加したものであるため，較正前の値は存在しない．

図 7.4，**表 7.7** に実験結果を示す．較正用クエリである Q.3 においては，損益分岐点の誤差は +0.49% であった．また，Q.3 に比較的ワークロード特性の近い Q.7 においては，誤差 +96% となり，選択率の値域に比べると比較的限定された範囲で損益分岐点を予測できていることがわかる．一方，Q.1 では索引検索がほぼシーケンシャル入出力となるため，アウトオブオーダ型クエリ実行による性能向上が実質望まれない状況であるため，損



(a) Q.1　(b) Q.3

(c) Q.7　(d) Q.8

図 7.5　クエリ実行時間と各クエリ実行プランのコスト曲線

表 7.7　各クエリの実行時間における損益分岐点とコスト見積りによる損益分岐点

クエリ	実行時間 BEP	コスト曲線 BEP
Q.1	9	122 (誤差 +1,255%)
Q.3	193,744	192,787 (誤差 +0.49%)
Q.7	1,675,129	3,284,646 (誤差 +96%)
Q.8	141,621	N/A

益分岐点の予測が大きく外れている．前節の実験と同様，これはデータ編成に関する詳細な統計情報を得ることができれば改善可能である．また，Q.8 では選択数 3,750,000 を超える領域では索引によって選択可能な全てのタプルが選択対象となるため，これ以上の選択率ではコストが頭打ちとなり，損益分岐点を算出することができなかった．第 4 章で示したように，Q.8 は Q.3 と比べて結合テーブル数が多いために CPU 演算負荷が比較的高く，データベースエンジン加速機構による性能向上率が低いため，Q.8 のコストを過小に

見積もられている．これはすなわち，入出力待ち時間の支配的なクエリにおいても，正確に損益分岐点を予測するためには CPU 演算負荷をコストモデルにおいて考慮する必要があることを示唆する．

以上の結果より，較正用クエリにワークロード特性が近いクエリに関しては，提案するコストモデルにより 96% の誤差で損益分岐点を予測することが可能である一方，ワークロード特性が異なるクエリや物理的なデータ編成をコストモデルにおいて考慮することで，更なる改善の余地があることが確認された．

第 8 章

結論

8.1 本論文のまとめ

本論文では，従前のインオーダ型データベースエンジンにおける同期的なクエリ処理において，性能利得の大きい範囲のみを非同期化し，同期的クエリ処理と連携することで高速化を実現する同期・非同期連携クエリ処理を提唱した．そして，同期・非同期連携クエリ処理を実現するデータベースエンジン構成法として，既存のインオーダ型データベースエンジンに対し，同期・非同期協調制御機構を備えるアウトオブオーダ型データベースエンジンを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した．入出力の非同期化を行うデータベースエンジン加速機構は，アウトオブオーダ型データベースエンジンが先行的にバッファプールへとデータを読み込み，インオーダ型データベースエンジンにおける入出力処理時間を縮減することで，透過的にその処理性能を高速化する点に特徴を有する．オープンソースデータベース管理システム PostgreSQL を対象として当該加速機構の試作実装である PgBooster の開発を行い，その性能評価を行ったところ，TPC-H データセットを用いた実験においては最大で 350 倍，複数のクエリについて 100 倍以上の性能向上が確認された．また GPS データセットを用いた評価実験では，拡張データ型に対する問合せ処理が最大で 240 倍高速化されることが確認され，実データに対してもデータベースエンジン加速機構の有効性が確認された．

さらに，データベースエンジン加速機構によって入出力性能のみならずプロセッサの並列演算性能を活用するための拡張を施す設計を示した．TPC-H データセットと試作実装 PgBoosterMC を用いた初期性能評価実験の結果，CPU 演算負荷が高く PgBooster では低 CPU 負荷時の 50% 程度まで性能が低下する状況においても，PgBoosterMC は低 CPU 演算負荷時とほぼ同等の性能を示し，複数プロセッサコア活用により高効率に入出

力帯域を活用可能なクエリ領域がより広いことが確認された．また人の流れデータセットを用いた性能評価実験により，CPU 演算負荷の高い実データ処理においても，複数プロセッサコアの性能を活用することで 56.3 倍の性能向上が確認された．

また，アウトオブオーダ型データベースエンジンにおいて，複数のクエリ実行間で優先度に基いて動的資源調停を行う手法を提案した．TPC-H データセットを用いた評価実験により，優先度が高いクエリ実行が開始されると当該クエリにスループットを集中的に割当て，速やかにクエリ実行を完了するといった制御が実現されることを示した．

加えて，データベースエンジン加速機構を用いたクエリ実行のコストモデル，および実行環境に対してコストモデルを自動較正する手法の提案を行った．PostgreSQL において提案するコストモデルを実装し，TPC-H データセットを用いた評価実験を行った結果，提案する自動較正手法によって PostgreSQL のプラン選択精度が複数のクエリにおいて改善し，データベースエンジン加速機構を利用するプラン選択においては，較正用クエリにワークロード特性の近いクエリに関しては選択率の誤差 96% 程度で損益分岐点を予測可能であることが確認された．

8.2 今後の研究課題と展望

本論文では，データベースエンジン加速機構の設計と実装を行い，評価実験によって有効性を確認した．これにより，既存のインオーダ型データベースエンジン実装の実行論理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の高速性を取り込むことが実現可能であることを示した．一方，複数プロセッサコア性能活用に関する初期評価実験により，入出力のみを高速化するデータベースエンジン加速機構では，CPU 演算負荷が高い場合や，1000 ドライブ規模やそれ以上の環境における性能活用の困難さがうかがわれる結果も確認された．本論文では既存のインオーダ型データベースエンジンに対する適用の容易性を重要視し，バッファプールを介して入出力のみを高速化するという戦略をとったが，将来の計算機システムにおいては複数プロセッサコアの性能活用は必須であるといえよう．そのためのインオーダ型データベースエンジンとアウトオブオーダ型データベースエンジンがより緊密に連携可能なソフトウェアアーキテクチャについて，今後検討を進めてゆきたい．また，データベースエンジン総体としての利用性を考える上で，クエリ最適化器におけるコストモデルの正確性は重要である．本論文で提案する手法は比較的単純なコストモデルにもとづいており，較正用クエリとワークロード特性が異なるクエリに関しては必ずしも良い精度でプラン選択を行うに至っていないため，CPU 演算コストや物理的なデータ編成等の要素を考慮し，より正確なコストモデルの構築方法を検討し

たい．

今日に至る動向から計算機システム進化の今後を外挿すると，ストレージシステムはより高密度化・広帯域化し，プロセッサの単一ダイにおける並列演算性能は増加すると予想される．またクロック遅延による処理性能の制約を克服するため，ハードウェア自体も現在主流のクロック同期方式から脱却し，非同期的な側面がより強くなる可能性も考えられる [118]．その際には，ソフトウェアの非同期化に関する技術の重要性はより一層高まるであろう．同期的な論理に基づくソフトウェアを非同期へと転換することは，ソフトウェアの取りうる実行状態の可能性を爆発的に増加させ，多くの場合非決定的を生じるため，困難を伴う場合が多い．この文脈において，ソフトウェア自身の同期から非同期への不連続な転換を行わずとも，同期・非同期連携という両者が共存する形態により，非同期化による高速性を享受することが可能であるという一例を示した点に，本研究の価値があるといえよう．ソフトウェアの本格的な非同期化は未だ解決されていない問題に満ちた未開拓領域である．本研究を，この問題領域を切り拓いてゆく端緒としたい．

謝辞

本論文に纏めるにあたり，実に多くの方々から御指導や御協力を頂きました．ここに感謝の意を表したいと思います．

指導教員である東京大学生産技術研究所の喜連川優教授（国立情報学研究所長）には，素晴らしい研究テーマを与えて頂き，そして終始温かい激励とともに御指導，ご鞭撻頂きましたことを心より感謝申し上げます．東京大学大学院情報理工学系研究科修士課程に進学して以降，喜連川教授にはデータベースシステム研究の道に導いて頂いただけでなく，科学技術の先端を切り拓く研究に取り組む姿勢に関して数多くの御助言を頂きました．重ねて深く感謝申し上げます．

学位審査において，主査である国立情報学研究所の安達淳教授を始め，東京大学大学院情報理工学系研究科の坂井修一教授，江崎浩教授，田浦健次朗准教授，豊田正史准教授には，大変貴重な御指摘と御指導を賜りました．厚く御礼申し上げます．

本研究の実験システム構築は，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」の支援なくしては行うことはできませんでした．当該プログラムの支援に深く感謝申し上げます．

GIS データ処理に関する実験に際して，東京大学空間情報科学研究センターの柴崎亮介教授には実験用データセットをご提供頂き，当該データセットを用いた実験に関し有益な御助言を頂戴しました．また東京大学空間情報科学研究センターの金杉洋特任研究員，Teerayut Horanont 特任研究員には，当該データセットを用いた実験の実施にあたり多大なる御協力を頂きました．厚く御礼申し上げます．

SRA OSS, Inc. 日本支社の石井達夫取締役支社長，長田悠吾氏にはデータベースエンジン加速機構の試作実装 PgBooster をご試用頂き，業務において実際にデータベースシステムを開発・運用されている観点から貴重なご意見を頂戴しました．深謝申し上げます．

東京大学生産技術研究所の合田和生特任准教授には，東京大学大学院情報理工学系研究科修士課程に進学して以降，データベースシステム研究に関する技術・知識を大変丁寧に御指導頂きました．また本研究を進める中で数多くの御助言を頂き，実験環境構築にあたり数多くの便宜をはかっていただきました．東京大学生産技術研究所の山田浩之特任研究員には，実験に利用する計算機システムの管理に日頃より御協力頂き，研究に関する議論や研究生活の相談を通して大変お世話になりました．東京大学生産技術研究所の川道亮治特任研究員には，本研究におけるプログラム開発や実験の実施にあたり大変多くの御協力を頂きました．また，その他の多くの喜連川研究室の先輩，後輩には，日々の研究生活における様々な面で大変お世話になりました．改めてここに深く感謝申し上げます．

著者は，グローバル COE プログラム「セキュアライフエレクトロニクス」のリサーチアシスタント，その後は日本学術振興会特別研究員（DC2）として支援を受けることにより，経済的な問題を一切気にすることなく，研究に邁進することができました．勉学の途上にある者に対し，多大な支援を惜しまないこれらの寛容な諸制度に対して，深く感謝の意を表します．

最後になりますが，著者が志す道を進むことを応援し，故郷の富山から温かく見守って下さった両親に深く感謝します．そして，常に温かい励ましをもって研究生活を支えてくれた妻の桃子に心から感謝します．

参考文献

[1] Mieszko Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, Christopher W. Fletcher, Omer Khan, and Srinivas Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pp. 175–185, Washington, DC, USA, 2011. IEEE Computer Society.

[2] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, Vol. 30, No. 4, pp. 8–19, July 2010.

[3] Evangelos Eleftheriou, Robert Haas, Jens Jelitto, Mark A. Lantz, and Haralampos Pozidis. Trends in storage technologies. *IEEE Data Eng. Bull.*, Vol. 33, No. 4, pp. 4–13, 2010.

[4] David A. Patterson. Latency lags bandwith. *Commun. ACM*, Vol. 47, No. 10, pp. 71–75, October 2004.

[5] John Gantz and David Reinsel. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. Technical report, IDC, Dec 2012.

[6] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, Vol. 1, No. 3, pp. 189–222, September 1976.

[7] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, Vol. 1, No. 2, pp. 97–137, June 1976.

[8] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill higher education. McGraw-Hill Education, 2003.

[9] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, Vol. 35, No. 6, pp. 85–98, June 1992.

[10] Carrie Ballinger and Ron Fryer. Born to be parallel: Why parallel origins give teradata an enduring performance edge. *IEEE Data Eng. Bull.*, Vol. 20, No. 2, pp. 3–12, 1997.

[11] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pp. 225–237, 2005.

[12] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, Vol. 2, No. 1, pp. 385–394, August 2009.

[13] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pp. 145–156, New York, NY, USA, 2002. ACM.

[14] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, DaMoN '07, pp. 4:1–4:6, New York, NY, USA, 2007. ACM.

[15] 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, Jun 2009.

[16] 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, Mar 2013.

[17] 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, Mar 2013.

[18] 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, Mar 2013.

[19] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for db2 using solid state storage. *Proc. VLDB Endow.*, Vol. 2, No. 2, pp. 1318–1329, August 2009.

[20] Eugene Wong and Karel Youssefi. Decomposition—a strategy for query processing. *ACM Trans. Database Syst.*, Vol. 1, No. 3, pp. 223–241, September 1976.

[21] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source database, 2014.

[22] Oracle Corporation. Mysql : The world's most popular open source database, 2014.

[23] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, Vol. 6, No. 11, pp. 1080–1091, August 2013.

[24] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pp. 981–992, New York, NY, USA, 2008. ACM.

[25] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. *SIGMETRICS Perform. Eval. Rev.*, Vol. 15, No. 1, pp. 69–77, May 1987.

[26] H. C. Du and J. S. Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Trans. Database Syst.*, Vol. 7, No. 1, pp. 82–101, March 1982.

[27] Myoung Ho Kim and Sakti Pramanik. Optimal file distribution for partial match retrieval. *SIGMOD Rec.*, Vol. 17, No. 3, pp. 173–182, June 1988.

[28] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in bubba. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pp. 99–108, New York, NY, USA, 1988. ACM.

[29] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of

processors. *ACM Trans. Database Syst.*, Vol. 10, No. 1, pp. 29–56, March 1985.

[30] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. *Scientific and Statistical Database Management, International Conference on*, Vol. 0, p. 383, 2004.

[31] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, Vol. 1, No. 1, pp. 63–74, 1983.

[32] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pp. 359–370, New York, NY, USA, 2004. ACM.

[33] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pp. 481–492, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[34] Patrick Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst.*, Vol. 9, No. 1, pp. 133–161, March 1984.

[35] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, Vol. 2, No. 1, pp. 44–62, March 1990.

[36] Donovan A. Schneider and David J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pp. 469–480, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[37] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, and Honesty C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pp. 15–26, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[38] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel query processing in dbs3. In

Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on, pp. 93–102, Jan 1993.

[39] Mikal Ziane, Mohamed Zaït, and Pascale Borla-Salamet. Parallel query processing with zigzag trees. *The VLDB Journal*, Vol. 2, No. 3, pp. 277–302, July 1993.

[40] Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 829–840. VLDB Endowment, 2005.

[41] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. Nested loops revisited. In *Proceedings of the 2Nd International Conference on Parallel and Distributed Information Systems*, PDIS '93, pp. 230–242, Washington, DC, USA, 1993. IEEE Computer Society.

[42] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pp. 325–336, New York, NY, USA, 2006. ACM.

[43] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pp. 487–498. VLDB Endowment, 2006.

[44] Christoffer Hall and Philippe Bonnet. Getting priorities straight: Improving linux support for database i/o. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 1116–1127. VLDB Endowment, 2005.

[45] 合田和生, 早水悠登, 喜連川優. 100 ドライブ規模のディスクストレージ環境におけるアウトオブオーダ型データベースエンジン OoODE の問合せ処理性能試験. 電子情報通信学会論文誌 D, Vol. J97-D, No. 4, pp. 729–737, Apr 2014.

[46] 山田浩之, 合田和生, 喜連川優. Hadoop をはじめとする並列データ処理系へのアウトオブオーダ型実行方式の適用とその有効性の検証. 電子情報通信学会論文誌 D, Vol. J97-D, No. 4, pp. 774–792, Apr 2014.

[47] R. J. Feiertag and E. I. Organick. The multics input/output system. In *Proceedings of the third ACM symposium on Operating systems principles*, SOSP

'71, pp. 35–41, New York, NY, USA, 1971. ACM.

[48] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, Vol. 3, No. 3, pp. 223–247, September 1978.

[49] Binny S. Gill and Dharmendra S. Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 33–33, Berkeley, CA, USA, 2005. USENIX Association.

[50] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential i/o. *SIGOPS Oper. Syst. Rev.*, Vol. 41, No. 3, pp. 189–202, March 2007.

[51] E.V. Carrera and R. Bianchini. Improving disk throughput in data-intensive servers. In *Software, IEE Proceedings-*, pp. 130–130, Feb 2004.

[52] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. *SIGMOD Rec.*, Vol. 22, No. 2, pp. 257–266, June 1993.

[53] K. Korner. Intelligent caching for remote file service. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pp. 220–226, May 1990.

[54] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. Technical report, Providence, RI, USA, 1991.

[55] D. Kotz and C.S. Ellis. Practical prefetching techniques for parallel file systems. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pp. 182–189, Dec 1991.

[56] C.D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Distributed Computing Systems, 1991., 11th International Conference on*, pp. 2–9, May 1991.

[57] S. Bhatia, E. Varki, and A Merchant. Sequential prefetch cache sizing for maximal hit rate. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pp. 89–98, Aug 2010.

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

1995. ACM.

[59] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pp. 3–17, New York, NY, USA, 1996. ACM.

[60] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. *SIGMETRICS Perform. Eval. Rev.*, Vol. 25, No. 1, pp. 100–114, June 1997.

[61] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pp. 377–390, Berkeley, CA, USA, 2008. USENIX Association.

[62] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pp. 173–186, Berkeley, CA, USA, 2004. USENIX Association.

[63] Alan J. Smith. Disk cache—miss ratio analysis and design considerations. *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 161–203, August 1985.

[64] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: An idea whose time has come. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pp. 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[65] Oracle Corporation. Performance and scalability in dss environment with oracle9i, Apr 2001.

[66] 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, Mar 2000.

[67] 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, Mar 2005.

[68] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pp. 20:1–20:14, Berkeley, CA, USA, 2007.

USENIX Association.

[69] Stergios V. Anastasiadis, Rajiv G. Wickremesinghe, and Jeffrey S. Chase. Circus: Opportunistic block reordering for scalable content servers. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pp. 201–212, Berkeley, CA, USA, 2004. USENIX Association.

[70] Sheldon Finkelstein. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, SIGMOD '82, pp. 235–245, New York, NY, USA, 1982. ACM.

[71] Hongjun Lu and Kian-Lee Tan. Batch query processing in shared-nothing multiprocessors. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 238–245. World Scientific Press, 1995.

[72] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pp. 249–260, New York, NY, USA, 2000. ACM.

[73] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, Vol. 13, No. 1, pp. 23–52, March 1988.

[74] George Candea, Neoklis Polyzotis, and Radek Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, Vol. 20, No. 2, pp. 227–248, 2011.

[75] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pp. 383–394, New York, NY, USA, 2005. ACM.

[76] M. J. Carey, R. Jauhari, and M. Livny. Priority in dbms resource scheduling. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pp. 397–410, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[77] HweeHwa Pang, M.J. Carey, and M. Livny. Multiclass query scheduling in real-time database systems. *Knowledge and Data Engineering, IEEE Transactions on*, Vol. 7, No. 4, pp. 533–551, Aug 1995.

[78] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The oracle database

resource manager: Scheduling cpu resources at the application level. *HPTS*, Vol. 1, No. 2, pp. 2–4, 2001.

[79] IBM Corp. Db2 query patroller guide: Installation, administration, and usage, May 2004.

[80] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, Vol. 22, No. 2, pp. 297–306, June 1993.

[81] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[82] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, Vol. 30, No. 1, pp. 31–42, June 2002.

[83] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, Vol. 33, No. 1, pp. 157–168, June 2005.

[84] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, Vol. 9, No. 4, pp. 560–595, December 1984.

[85] Mary A. Davidson. The root of the problem. `https://blogs.oracle.com/maryanndavidson/entry/the_root_of_the_problem`, Sep 2010.

[86] Bruce Momjian. One million strong. `http://momjian.us/main/blogs/pgblog/2011.html#April_20_2011_2`, Apr 2011.

[87] Tomas Ulin. Driving mysql innovation. In *Percona Live*, Apr 2013.

[88] 藤原真二, 茂木和彦, 田中美智子, 田中剛, 合田和生, 喜連川優. Tpc-h ベンチマークの 100tb クラスを用いた商用アウトオブオーダ型データベースエンジンの評価と同クラスへの世界初登録. 第 6 回データ工学と情報マネジメントに関するフォーラム, Mar 2014.

[89] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, Vol. 25, No. 2, pp. 73–169, June 1993.

[90] Noel Yuhanna, Mike Gilpin, and Catherine Salzinger. Market Update: Open Source Databases. Technical report, Forrester Research, Inc, Jul 2008.

[91] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. *SIGMOD*

Rec., Vol. 15, No. 2, pp. 340–355, June 1986.

[92] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pp. 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[93] Transaction Processing Performance Council. `http://www.tpc.org/`.

[94] PostGIS – Spatial and Geographic objects for PostgreSQL. `http://postgis.net/`.

[95] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pp. 47–57, New York, NY, USA, 1984. ACM.

[96] Chuan-Heng Ang and T. C. Tan. New linear node splitting algorithm for r-trees. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, SSD '97, pp. 339–349, London, UK, UK, 1997. Springer-Verlag.

[97] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pp. 322–331, New York, NY, USA, 1990. ACM.

[98] A. Korotkov. A new double sorting-based node splitting algorithm for r-tree. *Programming and Computer Software*, Vol. 38, No. 3, pp. 109–118, 2012.

[99] George Candea, Neoklis Polyzotis, and Radek Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, Vol. 20, No. 2, pp. 227–248, April 2011.

[100] C. Balkesen, J. Teubner, G. Alonso, and M.T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 362–373, April 2013.

[101] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, Vol. 2, No. 2, pp. 1378–1389, August 2009.

[102] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, Vol. 5, No. 10, pp. 1064–1075, June 2012.

[103] Rob McNelly. A big step forward in storage. `http://ibmsystemsmag.blogs.com/aixchange/ibm-flash-system-820/`, July 2014.

[104] George Crump. Lab report: Designing a 2 million+ iops architecture, Sep 2013.

[105] Annie P. Foong, Bryan Veal, and Frank T. Hady. Towards ssd-ready enterprise platforms. In Rajesh Bordawekar and Christian A. Lang, editors, *ADMS@VLDB*, pp. 15–21, 2010.

[106] Eduardo Freitas. Best practices for microsoft sql server on hitachi universal storage platform vm, Jun 2009.

[107] Charlie Johnson and Jeff Welser. Future processors: Flexible and modular. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pp. 4–6, New York, NY, USA, 2005. ACM.

[108] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: Preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pp. 67–72, New York, NY, USA, 2006. ACM.

[109] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485, New York, NY, USA, 1967. ACM.

[110] Wentao Wu, Yun Chi, Shenghuo Zhu, J. Tatemura, H. Hacigumus, and J.F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pp. 1081–1092, April 2013.

[111] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, Vol. 35, No. 1, pp. 7:1–7:47, February 2008.

[112] Yifan Zhou. *Extensions of an Empirical Automated Tuning Framework*. dissertation, University of Maryland, 2013.

[113] Alfonso F. Cárdenas. Analysis and performance of inverted data base structures. *Commun. ACM*, Vol. 18, No. 5, pp. 253–263, May 1975.

[114] S. J. Waters. Hit ratios. *The Computer Journal*, Vol. 19, No. 1, pp. 21–24, 1976.

[115] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM*, Vol. 20, No. 4, pp. 260–261, April 1977.

[116] Arun Swami and K. Bernhard Schiefer. Estimating page fetches for index scans with finite lru buffers. *The VLDB Journal*, Vol. 4, No. 4, pp. 675–701, October 1995.

[117] Lothar F. Mackert and Guy M. Lohman. Index scans using a finite lru buffer: A validated i/o model. *ACM Trans. Database Syst.*, Vol. 14, No. 3, pp. 401–424, September 1989.

[118] Ivan Sutherland. The tyranny of the clock. *Commun. ACM*, Vol. 55, No. 10, pp. 35–36, October 2012.

発表文献

査読付き論文誌

- 早水悠登，合田和生，中野美由紀，喜連川優．オンライントランザクション処理における Dynamic Voltage and Frequency Scaling を用いたアプリケーション指向省電力化手法．情報処理学会論文誌 データベース（TOD50） Vol.4 No. 2，2011 年 6 月
- 合田和生，早水悠登，喜連川優．100 ドライブ規模のディスクストレージ環境におけるアウトオブオーダ型データベースエンジン OoODE の問合せ処理性能試験．電子情報通信学会論文誌，Vol.J97-D No.4，2014 年 4 月
- 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可能なデータベースエンジン加速機構．情報処理学会論文誌 データベース（TOD62） Vol.7 No.2, 2014 年 6 月
- 合田和生, 早水悠登, 喜連川優．アウトオブオーダ型データベースエンジン OoODE における優先度に基づく複数クエリ実行間の動的資源調停．電子情報通信学会論文誌．（投稿済）

国際学会

- Keisuke SUZUKI, Yuto HAYAMIZU, Daisaku YOKOYAMA, Miyuki NAKANO and Masaru KITSUREGAWA. Comprehensive Analytics of Large Data Query Processing on Relational Database with SSDs. Proceedings of The 25th Australasian database conference (ADC 2014), 2014.07 (to appear)
- Yuto HAYAMIZU, Kazuo GODA, Miyuki NAKANO and Masaru KITSUREGAWA. Application-aware Power Saving for Online Transaction Processing us-

ing Dynamic Voltage and Frequency Scaling in a Multicore Environment. Proceedings of Architecture of Computing Systems, 24th International Conference (ARCS 2011), pp. 50 - 61, 2011.02

査読付き国内学会

- 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 年 11 月

その他

- 早水悠登，合田和生，喜連川優. オンライントランザクション処理における Dynamic Voltage and Frequency Scaling の消費電力削減効果に関する実験的考察. 電子情報通信学会データ工学研究会，電子情報通信学会技術報告 Vol. 110, No. 162，DE2010-21，pp.41-46，2010 年 8 月
- 早水悠登，合田和生，中野美由紀，喜連川優. オンライントランザクション処理における Dynamic Voltage and Frequency Scaling を用いたアプリケーション指向省電力手法の実験的考察. 電子情報通信学会データ工学研究会，電子情報通信学会技術報告，Vol. 110，No. 328，DE2010-37，pp. 67-72，2010 年 12 月
- 早水悠登，合田和生，中野美由紀，喜連川優. Solid State Drive 搭載オンライントランザクション処理サーバにおける省電力化の実験的考察. 電子情報通信学会第 3 回データ工学と情報マネジメントに関するフォーラム／第 9 回日本データベース学会年次大会 (DEIM2011)，D8-1，2011 年 2 月
- 早水悠登，合田和生，中野美由紀，喜連川優. オンライントランザクション処理におけるスループットを考慮したプロセッサ省電力手法の実験的考察. 情報処理学会第 73 回全国大会，5M-9，2011 年 3 月
- 早水悠登，合田和生，中野美由紀，喜連川優. SSD 環境を対象とした Dynamic Voltage and Frequency Scaling 制御によるオンライントランザクション処理省電力化の実験的考察. 電子情報通信学会データ工学研究会，電子情報通信学会技術報告 Vol. 111，No. 76，DE2011-3，pp.13-18，2011 年 6 月
- 早水悠登，合田和生，中野美由紀，喜連川優. オンライントランザクション処理に

おける高速フラッシュストレージの性能活用に関する実験的考察．情報処理学会第 74 回全国大会，1N-5，2012 年 3 月

- 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．電子情報通信学会第 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／第 11 回日本データベース学会年次大会 (DEIM2013)，F3-2，2013 年 3 月
- 合田和生，早水悠登，山田浩之，喜連川優．アウトオブオーダ型データベースエンジン OoODE の試作とディスクストレージ環境における実験的性能評価．平成 25 年電気学会電子・情報・システム部門大会，2013 年 9 月
- 鈴木恵介，早水悠登，横山大作，中野美由紀，喜連川 優．SSD を用いた大規模データベースにおける複数問い合わせ処理高速化手法とその評価．電子情報通信学会第 6 回データ工学と情報マネジメントに関するフォーラム／第 12 回日本データベース学会年次大会 (DEIM2014)，D8-6，2014 年 3 月
- 早水悠登，合田和生，喜連川優．フラッシュストレージ環境におけるアウトオブオーダ型データベースエンジン OoODE の実験的クエリ処理性能評価．電子情報通信学会第 6 回データ工学と情報マネジメントに関するフォーラム／第 12 回日本データベース学会年次大会 (DEIM2014)，A2-3, 2014 年 3 月