anagodb: 挑戦的なデータベースコア技術研究のための実践的コードベース

早水　悠登†　　川道　亮治††　　小沢　健史††　　合田　和生††

† 株式会社ジャスティス・テクノロジーズ　〒 153–8505 東京都目黒区駒場 4-6-1 駒場オープンラボラトリ 505
†† 東京大学生産技術研究所　〒 153–8505 東京都目黒区駒場 4-6-1
E-mail: †hayamizu@justice-tech.com, ††{kawamichi,ozawa,kgoda}@tkl.iis.u-tokyo.ac.jp

あらまし　データベースコア技術の研究を行う上では，新たな技術を実装し評価するための土台として，所謂トイプログラムの域を脱したコードベースが必要不可欠である．OSS のデータベースシステムがこの目的でしばしば用いられるが，既存設計の制約が障害となり，挑戦的な技術の実装・評価には必ずしも最適ではない場合も多い．本論文では，データベースコア技術研究のためのコードベースとして東京大学が開発する anagodb を紹介する．anagodb は分析系クエリの高速処理に特化した関係データベースシステムであり，東京大学にて培われた高速クエリ処理技法の知見に基づきフロムスクラッチで開発されている．コード規模が最小限となるよう機能性は絞り込みつつ，TPC-H ベンチマークを SQL クエリのまま実行できる程度の実用性を持つ．クエリオペレータ等をプラガブルに拡張可能な枠組みにより新規技術の実装を容易にするとともに，高速バルクローダにより大規模データを用いた実験を機動的に繰り返すことが可能である．本論文では，anagodb の設計と実装，およびその性能評価を示す．

キーワード　データベースコア技術, 研究用コードベース, 関係データベースシステム, OLAP , 問合せ処理

1　はじめに

データベースシステムの中核をなすデータベースコア技術の研究を行う上では，新たな技術を実装し評価するための土台として，所謂トイプログラムの域を脱したコードベースが必要不可欠である．

こうした研究における評価の土台として，オープンソースのデータベースシステムがしばしば用いられる．関係データベースシステムにおいては PostgreSQL [18] や MySQL [16] などがよく知られている例である．これらのオープンソースデータベースシステムを用いることの最大の利点は，サードパーティが提供するツール等も含めて，一定程度成熟したコードベースを利用して研究の評価実験を計画できる点にある．例えば PostgreSQL には，オープンソースの地理情報システム構築で広く使われている PostGIS [17] という拡張が存在し，地理情報向けのクエリ処理アルゴリズムや索引アルゴリズムの改善を行う研究においては，現実で使われるワークロードをそのまま評価対象として利用することが可能である．

一方で，挑戦的なデータベースコア技術を評価のために実装する際には，既存のソフトウェア設計やモジュール分割方針と整合性をとることが困難である場合も少なくなく，オープンソースデータベースシステムが実装の土台として好適ではない場合もしばしば存在する．既に確立した設計方針やソフトウェアアーキテクチャに基づく膨大なコードベースから成るオープンソースデータベースシステムを土台とした上で，新たな技術の本来持つポテンシャルを最大限に引き出した実装を行い，評価することは容易ではない，あるいは現実的には不可能である場合は少なくない．

著者らの研究チームは，分析系クエリ，とりわけ選択性の高いクエリを極めて高速に処理することができる，アウトオブオーダ型実行方式 [8] (Out-of-Order database execution, OoODE) なる新たなクエリ実行方式と，それに基づくアウトオブオーダ型データベースエンジンの構成法に取り組んできた．その基本となる考え方は，クエリに内在する処理の並列性に基づいて，実行時にクエリ処理を動的タスク分解するとともに，入出力をすべて非同期化することで，ストレージ入出力帯域と多数のプロセッサコアの演算性能を最大限に活用する点にある．

アウトオブオーダ型データベースエンジンの初期プロトタイプは，MySQL の InnoDB ストレージエンジンのストレージフォーマットを利用し，クエリ処理エンジンを独自に実装した [11]．このプロトタイプにより，アウトオブオーダ型データベースエンジンの持つ高速性を，実際に動作するプログラムによって実証した．アウトオブオーダ型データベースエンジンの I/O 高速性を既存のデータベースエンジンに組み込むというアイディアに基づき，PostgreSQL に対してプラグイン型加速機構 PgBooster [25] を開発し，実際にその有効性を示した．関係データベースシステムにとどまらず，非構造データを扱う並列処理システムにおいてもアウトオブオーダ型データベースエンジンのアプローチが有効であることを示すため，Hadoop/Hive をベースに Hadooode [24] を開発し，128 ノードクラスタにおいて高い処理性能とスケーラビリティが得られることを示した．またクラウド環境において動的伸縮可能なクエリ処理エンジンというアイディアに基づき，AWS DynamoDB をストレージエンジンとして利用するクエリ処理系 [14] を開発し，1M IOPS

を超える高い I/O 処理性能と，ユーザの要求に応じた処理性能の伸縮が実現可能であることを示した．

ここまでに記したように，アウトオブオーダ型データベースエンジンというデータベースエンジン構成法の有効性を示すため，著者らは個別の研究アイディアごとに様々なデータベースエンジンを開発してきた．こうした取り組みにより，個別のテーマ毎にはその有効性を示すことに成功してきた．しかしながら，その開発を進める上では既存のコードベースを利用することの難しさも数多く経験した．もともと想定されていないアウトオブオーダ型データベースエンジンという構成法を，既存の巨大なコードベースに適用すること自体が困難であり，開発作業だけでも多くの労力と時間を要する．また，オープンソースのコードベースを利用する場合，大幅に改変されたコードベースを最新のアップストリームコードに追随しながら維持することは極めてコストがかかるため，新たな研究を進めるためにはコードベースの維持を諦めざるを得ない状況も経験した．

こうした経験を経て，著者らの研究チームでは継続的なデータベースコア技術研究のためのコードベースとして，新たに anagodb というデータベースシステムを開発している．anagodb はアウトオブオーダ型データベースエンジンをはじめとする，著者らの研究チームにおいて培われた高速クエリ処理技法の知見に基づき，フロムスクラッチで開発されているデータベースシステムである．分析系クエリの高速処理という目的に特化することでコード規模を最小限に抑えて，研究用途における開発効率をできるだけ高めることを第一としている．その上で，一般的な関係データベースと同様にテーブルや索引を管理する機能や，大規模データを取り扱うためのバルクロード機能を備え，TPC-H ベンチマークを SQL クエリのままで実行することが可能である程度に，実用的な SQL データベースシステムとしての機能性を持つように設計・実装されている．

本論文では，anagodb においてアウトオブオーダ型実行を実現するためのソフトウェア設計について説明するとともに，現段階での anagodb の性能評価結果を示す．

2 anagodb の設計と実装

anagodb はアウトオブオーダ型データベースエンジンを基礎とする関係データベースシステムであり，C 言語によって実装されている．anagodb は一般的な関係データベースシステムと同様に，下記の機能を備える．

- 二次記憶装置に格納されたデータベース上でのリレーションや索引の管理
- 専用のプロトコルおよび JDBC 経由でのクエリや管理コマンドの受付けと処理
- SQL で記述された関係クエリのクエリ実行プランへの変換
- 複数の実行モードによるクエリ実行のサポート: シリアル実行 (動的タスク分解や非同期 I/O 発行を行わないシングルスレッド実行)，OoODE 実行 (動的タスク分解と非同期 I/O 発行を用いるアウトオブオーダ型のクエリ実行)
- 大規模データのバルクロード

SQL によって記述できるクエリの範囲は成熟した関係データベースシステム実装と比較すると限定的ではあるものの，TPC-H ベンチマークの標準クエリを実行可能な程度の実用性を有するよう，SQL のサブセットをサポートするように設計・実装されている．anagodb は，分析系データベースのワークロードに目的特化しているため，現在はトランザクション機能についてはサポートしていない．anagodb は，単一の Linux サーバにおいてデータベースシステムを構成できるだけではなく，複数の Linux サーバをネットワークで接続したシェアードナッシング型のデータベースを構成可能である．本論文では，anagodb の基本設計を議論の対象とするため，簡単のために単一サーバでの環境を前提として議論する．

2.1 クエリ実行と動的タスク分解

anagodb におけるアウトオブオーダ型実行方式によるクエリ処理の基本的な流れを以下に示す．anagodb が新たなクエリを受付けると，クエリ実行プランへとコンパイルする．一般的な関係データベースシステムでは木構造のクエリ実行プランが採用されることが多いが，anagodb ではオペレータをリストで繋げた線形構造でクエリ実行プランを構成する．オペレータは前段オペレータから入力をレコードで受け取り，後段オペレータに対して出力をレコードとして渡すことが基本インタフェースとして規定されているが，ソートやハッシュ結合などレコード以外の構造体を受け渡すオペレータも一部存在する．クエリの実行が始まると，まず anagodb は最初のオペレータを実行するための最初のタスクを生成し，実行する．各タスクの実行中に，動的タスク分解が可能な処理を検出するたびに，anagodb は当該処理を別個に実行するための新たなタスクを生成する．

動的タスク分解が可能な処理は，データベースオペレータごとに規定することができる．例えば，**全表走査**オペレータはストレージに格納されたテーブルからレコードを取得するが，一般には複数レコードをまとめてページとして構造化したり，また複数ページをまとめたブロックとしてエクステントなどが定義される構造もしばしば用いられる．こうした構造を利用すると，各エクステントの走査や，各ページの走査を個別のタスクとして動的タスク分解を行うことが可能であり，複数の I/O 命令を同時に発行することでストレージの I/O 帯域や，フィルタ処理のための CPU 資源を効率的に活用することができる．

また別の例としては，**B^+tree 索引走査**オペレータがある．B^+tree 索引を用いたテーブルの範囲検索を行う場合，選択条件に合致する可能性がある子ノードの探索は全て独立して行うことが可能であるため，各子ノードの探索を個別のタスクとして動的タスク分解を行うことができる．

また**ハッシュ結合** (ハッシュ表ビルド処理およびプローブ処理) のオペレータや，ネステッドループ結合のオペレータ，ソート処理やサブクエリ実行，集約演算などについても同様に動的タスク分解可能な処理を規定することができる．

anagodb において，これらデータベース演算を行うオペレータはプラグイン機構によって抽象化されており，所定のインタフェースを満たす関数とオペレータ記述子を定義することで，

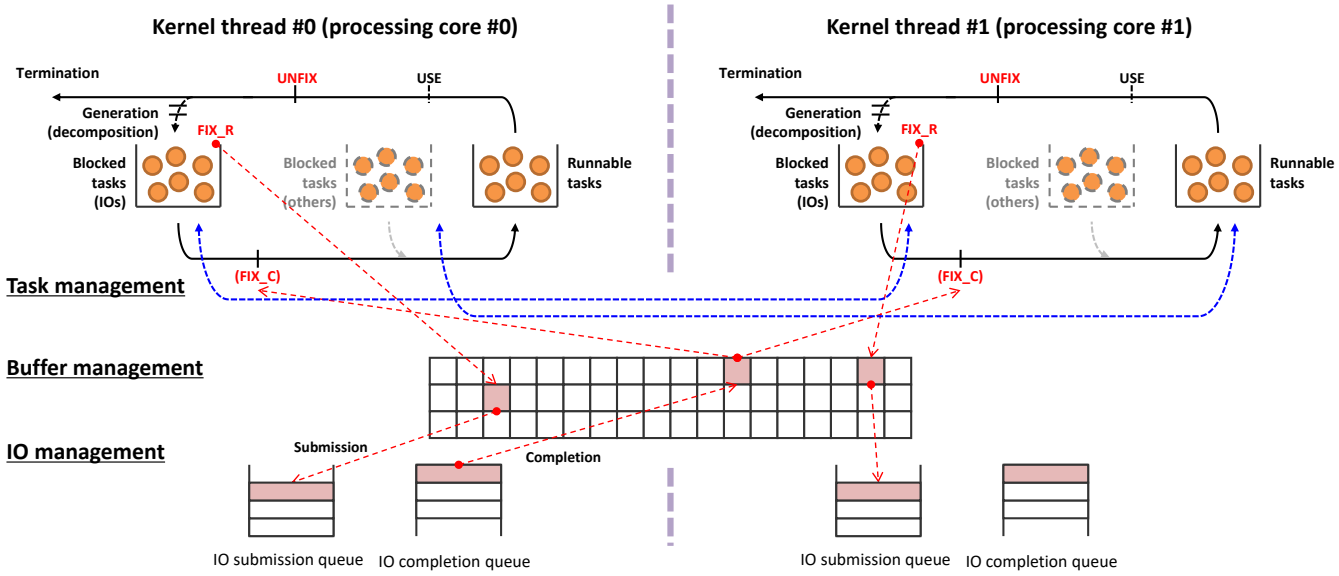


図 1　anagodb におけるタスクと I/O の管理方式

新たなオペレータを追加することが可能である．このプラグイン機構は，サポートする演算の範囲を比較的容易に追加可能であるほか，同じ機能を持つオペレータを異なるアルゴリズム等で方式で複数実装し，実行時に動的にオペレータ実装を切替えるといった様々な研究上の試行錯誤が行いやすいことを意識して設計されている．

2.2　タスクと I/O 管理

各オペレータにおいて動的タスク分解が可能な点を細粒度に規定することで，anagodb においては単一クエリ実行においても多数の並列実行可能なタスクが生成され，また多数の非同期 I/O が発行されることとなる．これら多数のタスクと非同期 I/O を効率的に管理することが高速なクエリ処理を実現する上で重要である．図 1 に anagodb におけるタスクと I/O の管理方式の概要を示す．

a）　タスク管理

anagodb はプロセッサ毎に専用のカーネルスレッドを割当て，スレッド毎にタスクを管理するタスクキューを持つ．タスクキューには，I/O やサブクエリ待ちなどの理由で実行がブロックされているキューと，実行可能なタスクが格納されるキューが存在し，カーネルスレッドはメインループで実行可能なタスクキューからタスクを取り出して実行する処理を繰り返す．

タスクにはそれぞれコルーチンが紐付けられており，タスクの実行状態はコルーチンによって管理される．[6], [13]．コルーチンは *local binding* ポリシ [10] に従い，原則として割当てられたカーネルスレッド内で実行され，またそのコルーチンから生成された子タスクのコルーチンも同じカーネルスレッド内で管理される．このポリシにより，プロセッサコア間のコヒーレンシ管理コストやコンテキストスイッチによるオーバヘッドを抑制することができる．

表 1　評価実験環境

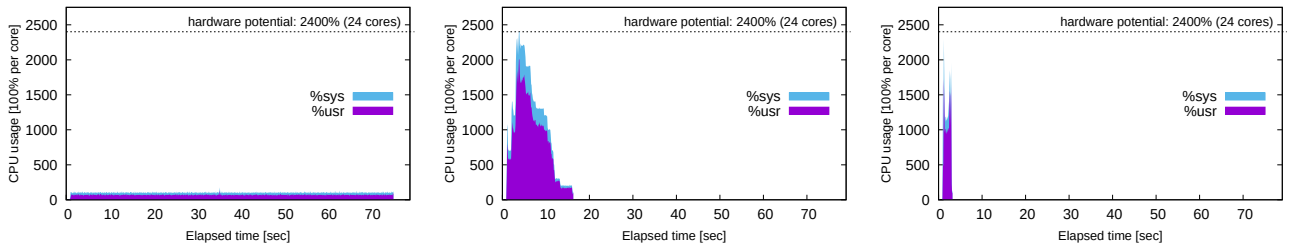
Experimental Database Server	
Server type	AWS EC2 i3en.6xlarge
	24 vCPU, 192GB RAM
Storage (database)	ext4 fs on mdraid striped volume with
	x4 7500GB NVMe SSD (instance store)

b）　メモリ管理

anagodb はメモリ管理においても local binding ポリシを採用する．カーネルスレッドが初期化される際に，そのカーネルスレッドが動作するプロセッサコアの NUMA ノードのローカルなメモリ空間からメモリプールを確保し，コルーチンからメモリ確保のリクエストが生じた際には当該メモリプールからメモリ領域を確保する [1], [19]．このポリシにより，コンテキストスイッチや NUMA ノード間のメモリアクセスによるコストを抑制することができる [12]．

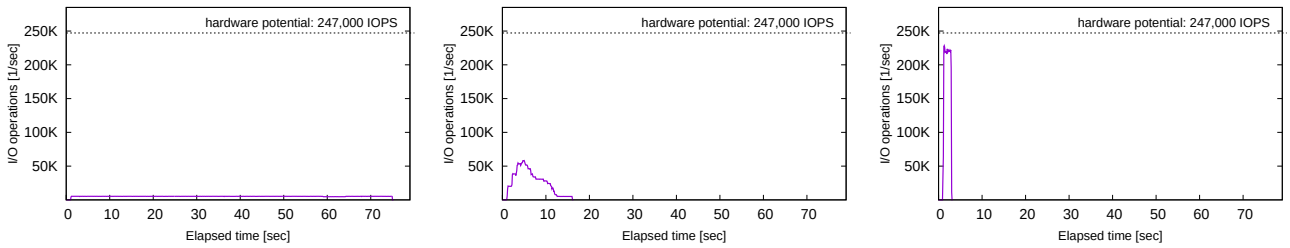
c）　I/O 管 理

anagodb では基本的に全ての I/O は非同期 I/O を前提として設計され，コルーチンと組合わせることでオペレータの処理を記述する．コルーチンが二次記憶からページを取得する際には，まずバッファマネージャにおいてバッファページの fix 操作を試みる．先行する処理によって fix されている等の理由により，もしページが既にバッファ上に存在する場合には，コルーチンは当該ページを利用して処理を継続する．もしページがバッファ上に存在しない場合には，コルーチンは実行がサスペンドされ，バッファマネージャが非同期 I/O リクエストを I/O マネージャに発行する [7]．I/O 完了時には，I/O マネージャは取得したデータをバッファページに転送し，対応するタスクを実行可能なタスクキューへと移動させる．anagodb は他のイベント (ネットワーク通信等) についても同様に非同期 I/O を基本として管理する．



(a) Serial　(b) Static Partitioning　(c) OoODE

図 2　Query(C) 実行時の CPU 利用率の挙動



(a) Serial　(b) Static Partitioning　(c) OoODE

図 3　Query(C) 実行時の I/O 挙動

d)　タスクの負荷分散

複数のプロセッサコアを効率よく活用するため，anagodb はタスクの負荷分散機構を備えている．ただし，各スレッドが一定量の実行可能なタスクを保持していれば十分であり，anagodb は動的タスク分解によって細粒度なタスクを多数生成するため，タスクの負荷分散ポリシは比較的保守的に設定されている．具体的には，anagodb は定期的に各スレッドの実行可能なタスク数を確認し，タスク数が全体の平均値から一定割合以上少ないと判定されるスレッドが存在する場合，タスクマイグレーションを行う．このポリシにより，カーネルスレッド同士の排他制御の機会を抑制しながら，効果的にタスクの負荷分散を行うことができる．

著者らの経験では，数十プロセッサコアと数十の磁気ディスクやフラッシュメモリデバイスを備えるようなデータインテンシブ処理向けのサーバ構成において，この負荷分散ポリシによって演算資源と I/O 資源を十分に効率良く活用できることを確認している．

3　評 価 実 験

本節では，anagodb の性能を示すための実験結果をまとめる．評価実験には，AWS クラウド上に構成した表 1 に示すデータベースサーバを利用した．データベース用のボリュームは，mdraid によって 2 つの NVMe SSD インスタンスストレージをストライピングすることで構成し，当該ボリュームを ext4 形式でフォーマットして利用した．anagodb が利用する実験用データベースは，TPC-H ベンチマークのデータセット (scale factor: 100) をデータベースにロードし，TPC-H ベンチマークの標準クエリで用いられている結合条件や選択条件に利用される属性についてそれぞれ索引を生成することで構築した．anagodb は 24 スレッドで起動し，各スレッド毎に 512MB，総じて 12GB のメモリをバッファキャッシュとして利用するよう設定した．また比較のために PostgreSQL においても同様の手順でデータベースを構築した．PostgreSQL はバージョン 17.2 を利用し，共有バッファサイズを 12GB，ワークメモリサイズを 12GB に設定して利用した．

評価実験においては，TPC-H ベンチマークの標準クエリに加えて，評価用クエリとして Query(A) から Query(D) までの 4 つのクエリを作成して利用した．これらクエリの SQL は論文末尾の付録に記載する．

3.1　anagodb の実行時挙動と資源利用率

anagodb にて評価用クエリ (C) を実行している際の CPU 利用率挙動を図 2 に，IOPS(毎秒の I/O 発行回数) 挙動を図 3 に示す．この実験では，クエリの実行方式による挙動の違いを示すため，anagodb を次の 3 つの動作モードで実行し，比較している．(1) **Serial**: 動的タスク分解や非同期 I/O 発行を行わず，1 スレッドによりクエリを実行する，(2) **Static Partitioning**: 実行前にクエリの処理を 24 分割して，それぞれについて Serial と同様の方式で実行する，(3) **OoODE**: 動的タスク分解と非同期 I/O 発行を用いてクエリを実行する．

Serial 実行では，1 スレッドによる実行のため CPU 利用率は最大 1 コア分の 100%，I/O 発行は最大で 5,200 IOPS 程度にとどまり，実行に 74.0 秒を要した．Static Partitioning 実行では，クエリが 24 スレッドで並列に実行されることから，Serial 実行に比べて CPU 利用率および IOPS が増加し，実行に 15.1 秒を要した．Static Partitioning 実行 は Serial 実行に対して 4.89 倍の性能向上がみられるが，負荷の不均衡により CPU 資源および I/O 資源を十分に活用できていないことがわかる．本実験においては，`c_custkey` の値の剰余によって各スレッドの

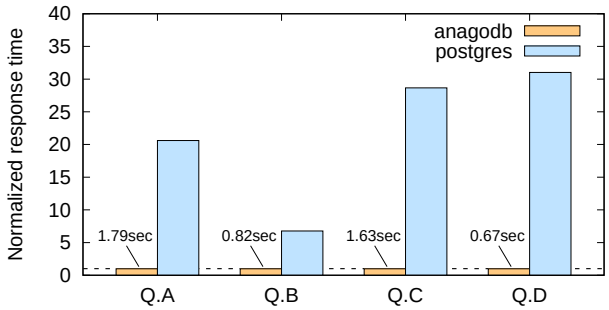


図 4 **選択性の高いクエリの正規化した実行時間比較** (anagodb の実行時間を 1 とし，PostgreSQL と実行時間を比較): Query(A) から (D) いずれにおいても anagodb は PostgreSQL に対して非常に短い実行時間でクエリを完了しており，Query(D) において最大 31.0 倍高速であった．

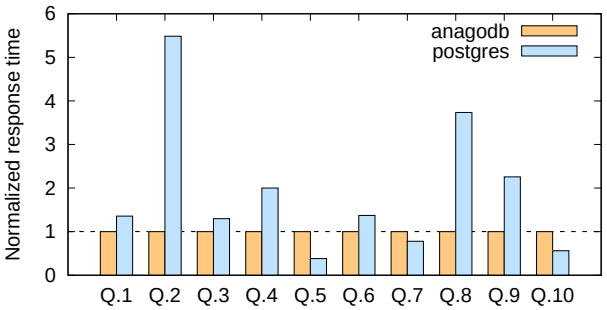


図 5 **TPC-H ベンチマーク標準クエリの正規化した実行時間比較** (anagodb の実行時間を 1 とし，PostgreSQL と実行時間を比較，anagodb の実際の実行時間は脚注 1 参照): 10 クエリ中 7 クエリでは anagodb が PostgreSQL に対して同程度または大幅に上回る性能を確認した．

処理対象を分割しているが，各スレッドが処理を完了する時間は最小 0.8 秒から最大 8.0 秒まで大きなばらつきが見られた．

OoODE 実行では，実行には 2.04 秒を要し，Serial 実行に比べて 36.2 倍，Static Partitioning 実行に比べて 7.4 倍の性能向上が見られた．実行開始直後から終了直前まで 220,000 IOPS 程度の I/O 発行が行われている．これは事前のマイクロベンチマークによって計測された実質的な上限性能である 247,000 IOPS の 89% に相当し，非常に高い効率で I/O 資源を活用できていることがわかる．本実験環境では，実行速度が I/O 性能によって律速されているため，CPU 利用率が最大の 24 コア分 2,400%に到達することはなく，1,000%から 1,600%を前後する挙動を示した．

3.2 選択性の高いクエリ

アウトオブオーダ型データベースエンジンは動的タスク分解と非同期 I/O 活用にその特徴を持つが，従来型のデータベースエンジンと比べて最も性能向上が見られるのは選択性の高いクエリである．一般に，選択性の高いクエリにおいては，走査対象とするデータを絞り込むために索引を利用するが，索引走査や，それに付随するネステッドループ結合の処理は，アウトオブオーダ型データベースエンジンにおける動的タスク分解と相性がよく，非常に高い効率で CPU 資源と I/O 資源を活用することができる．

本実験では，アウトオブオーダ型データベースエンジンにおいて最も性能優位性を発揮できるケースにおける評価を行うため，選択性の高いクエリとして評価用クエリ (A) から (D) を用いてその実行時間を計測した．比較対象として，PostgreSQL においても同様の計測を行いその結果を比較した．

結果を図 4 に示す．この図では，縦軸は anagodb の実行時間を 1 として正規化した値をプロットし，anagodb の実際の実行時間をラベルとして付記している．このグラフより，評価用クエリ (A) から (D) のすべてにおいて，anagodb は PostgreSQL に対して非常に短い実行時間でクエリを完了していることがわかる．その性能差は，評価用クエリ (D) において最大 31.0 倍であった．anagodb はいずれのクエリにおいても平均で 200,000 IOPS 以上と高効率に I/O 資源を活用することができている．一方 PostgreSQL では，クエリ実行プランの一部で並列走査を用いるクエリ実行プランが採用されているものの，全表走査の並列実行が行われるために，選択性の高いクエリにおいては無駄な I/O 発行が増加していること，またクエリ実行プランのうち索引走査を行う部分においては並列処理がなされず，結果として 5,000 IOPS 程度と anagodb の Serial 実行と同程度の I/O 資源活用率であったことなどから，十分に資源を活用した実行が行われていないことがわかった．

以上の評価により，アウトオブオーダ型データベースエンジンが高い性能優位性を持つ領域である，選択性の高いクエリにおいては，anagodb は既存データベース実装と比較して大幅な性能向上を達成していることが確認できた．

3.3 TPC-H ベンチマーク標準クエリ

TPC-H ベンチマークの標準クエリは，多くの場合テーブルの大部分のデータを走査して集計するクエリであり，全表走査やハッシュ結合が最も効率的である傾向が強い．このようなクエリにおいては，シーケンシャルな I/O アクセスパターンが主となるため，並列化による演算資源や I/O 資源の活用が，選択性の高いクエリに比べると比較的容易である．とりわけ，全表走査やハッシュ結合の並列処理は分析系クエリにおいて重要なワークロードとして，理論面において十分研究されており，また PostgreSQL 等の既存実装においても長期間にわたり実装効率が洗練されている．そのため，アウトオブオーダ型実行方式を適用すること自体による従来型データベースエンジンに対する性能向上の余地は小さい．

本実験では，このように本質的な性能優位性が期待されないワークロードにおいても，anagodb が既存実装に対して同程度の性能を発揮することが可能であることを評価するため，TPC-H ベンチマークの標準クエリ Q.1 から Q.10 を実行して，PostgreSQL と実行時間を比較した．

結果を図 5 に示す．この図では，縦軸は anagodb の実行時間を 1 として正規化した値をプロットしている．anagodb の実

際の実行時間は脚注に付記する [1]．このグラフより，TPC-H ベンチマーク Q.1 から Q.10 において anagodb が PostgreSQL に対して処理性能が低いのは Q.5, Q.7, Q.10 のみであり，他のクエリにおいては同程度またはある程度高速であることがわかる．PostgreSQL では全てのクエリにおいて全表走査とハッシュ結合を基本とするクエリプランが採用されており，全表操作とハッシュ結合の並列実行により演算資源や I/O 資源の利用率が一定程度高いことが確認できた．Q.5, Q.7, Q.10 に関しては，いずれも 5 段以上のハッシュ結合を伴うクエリであり，anagodb のハッシュ結合の実装が十分洗練されていないことから生じている性能低下が確認された．また，TPC-H 標準クエリであっても，Q.2 や Q.8 のように比較的選択性の高いクエリも存在し，こうしたクエリでは Q.2 で 5.4 倍，Q.8 で 3.7 倍と一定の性能向上が確認できた．

以上の評価により，アウトオブオーダ型データベースエンジンによる根源的な性能優位性がない領域のクエリにおいても，anagodb は既存データベース実装と同程度の性能を発揮することが可能であるが，ハッシュ結合においては改善の余地があることが確認された．

4 関 連 研 究

挑戦的なデータベースコア技術の研究は，純粋な学術研究というよりも，ある種のコードベースの開発プロジェクトとして推進される場合も少なくない．データベースシステムが学術分野として見なされるようになってから最も初期の例としては，Codd の提唱した関係モデル [5] に基づいて開発された INGRES [22] や System-R [2] が挙げられる．

それ以降も，新たなデータベースシステムのあり方を示す研究においては独自のコードベースが開発される場合が多く，こうして開発されたコードベースに基づいて更に新たな研究が進められたり，また商用製品へと展開される場合も多い．代表的な例としては，組み込み DB である Berkeley DB [15]，列指向データベースの C-Store [20] や MonetDB [3]，インメモリのハイブリッド OLTP/OLAP システムの Hyper [9]，分散ストリーム処理系の Apache Flink [4]，高速トランザクション処理システムである H-Store [21]，SILO [23] などが挙げられる．

5 お わ り に

本論文では，挑戦的なデータベースコア技術研究のためのコードベースとして開発されている anagodb について紹介した．anagodb は，アウトオブオーダ型実行方式に基づくデータベースエンジンであり，動的タスク分解と非同期 I/O 活用によって，選択性の高い分析系クエリを極めて高速に実行できる点に特徴がある．本論文では，アウトオブオーダ型実行の動的タスク分解によるタスク生成と，生成されるタスク及びそれに紐付く I/O を管理する anagodb の設計について紹介した．また，評価実験により anagodb が性能優位性を有する選択性の高いクエリにおいて，演算資源と I/O 資源を高効率に活用してクエリを高速に実行可能であることを示した．また，潜在的な性能優位性が期待されない TPC-H ベンチマーク標準クエリにおいても，既存データベースシステムと同程度の性能を発揮することが可能であることを示した．

1： 各クエリにおける，anagodb での実際の実行時間を記す．Q.1: 168.6 sec, Q.2: 4.3 sec, Q.3: 60.8 sec, Q.4: 55.8 sec, Q.5: 189.4 sec, Q.6: 30.0 sec, Q.7: 89.8 sec, Q.8: 53.0 sec, Q.9: 306.9 sec, Q.19: 136.4 sec.

謝　　辞

本研究の一部は，最先端研究開発支援プログラム (内閣府)，革新的研究開発推進プログラム (内閣府)，高度な IoT 社会を実現する横断的技術開発 (NEDO)，ビッグデータ基盤アーキテクチャの研究開発 (東大，KDDI)，戦略的イノベーション創造プログラム (内閣府) の支援を受けたものである．

付　　録

評価実験に利用した評価用クエリ (A) から (D) の SQL コードを下記に示す．

```
-- Query(A)
select sum(l.l_quantity) as sum_qty,
sum(l.l_extendedprice*(1-l.l_discount)*(1+l.l_tax))
from lineitem l
where l.l_partkey between 1 and 8000;
-- Query(B)
select p.p_brand, sum(l.l_quantity),
sum(l.l_extendedprice*(1-l.l_discount)*(1+l.l_tax))
from part p, lineitem l
where p.p_partkey = l.l_partkey
  and p.p_name like 'goldenrod lavender%'
group by p.p_brand;
-- Query(C)
select o.o_orderpriority, sum(l.l_quantity),
sum(l.l_extendedprice*(1-l.l_discount)*(1+l.l_tax))
from customer c
  join orders o on c.c_custkey = o.o_custkey
  join lineitem l on o.o_orderkey = l.l_orderkey
where c.c_name like 'Customer#00001%'
group by o.o_orderpriority;
-- Query(D)
select n.n_name, sum(ps.ps_supplycost),
sum(l.l_quantity),
sum(l.l_extendedprice*(1-l.l_discount)*(1+l.l_tax))
from customer c
  join orders o on c.c_custkey = o.o_custkey
  join lineitem l on o.o_orderkey = l.l_orderkey
  join partsupp ps on l.l_partkey = ps.ps_partkey
   and l.l_suppkey = ps.ps_suppkey
  join supplier s on s.s_suppkey = ps.ps_suppkey
  join nation n on s.s_nationkey = n.n_nationkey
```

```
where c.c_name like 'Customer#000001%'
group by n.n_name;
```

文　献

[1] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proc. HiPC 2006*, Vol. 4297 of *LNCS*, pp. 338–352, 2006.

[2] 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, and I. L. Traiger. System r: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, Vol. 1, No. 2, pp. 97–137, 1976.

[3] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. *VLDB Journal*, Vol. 10, No. 2-3, pp. 231–246, 2005.

[4] Paris Carbone, Alexander Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, Vol. 36, pp. 28–38, 2015.

[5] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, Vol. 13, No. 6, pp. 377–387, 1970.

[6] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, Vol. 31, No. 2, pp. 6:1–6:31, 2009.

[7] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proc. SYSTOR*, pp. 120–127, 2022.

[8] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada, and Masaru Kitsuregawa. Out-of-order execution of database queries. *Proc. VLDB Endow.*, Vol. 13, No. 12, pp. 3489–3501, 2020.

[9] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pp. 195–206. IEEE, 2011.

[10] Genki Kimura, Yuto Hayamizu, Rage Uday Kiran, Masaru Kitsuregawa, and Kazuo Goda. Efficient parallel mining of high-utility itemsets on multicore processors. In *Proc. ICDE*, pp. 638–652, 2023.

[11] Masaru Kitsuregawa and Kazuo Goda. アウトオブオーダ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, 6 2009.

[12] Zoltan Majó and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proc. SYSTOR*, p. 12, 2011.

[13] Chris D. Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, Vol. 95 of *LNCS*. 1980.

[14] Akihiro Okuno, Yuto Hayamizu, Kazuo Goda, and Masaru Kitsuregawa. 共有ストレージ型データベースエンジンにおける動的演算資源調整手法の提案. 情報処理学会論文誌 データベース, Vol. 11, No. 2, pp. 30–43, 7 2018.

[15] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pp. 183–191. USENIX Association, 1999.

[16] Oracle Corporation. Mysql: The world's most popular open source database, 2025. Accessed: 2025-01-08.

[17] PostGIS Project Steering Committee. Postgis: Spatial and geographic objects for postgresql, 2025. Accessed: 2025-01-08.

[18] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database, 2025. Accessed: 2025-01-08.

[19] Jaehyun Song, Minwoo Ahn, Gyusun Lee, Euiseong Seo, and Jinkyu Jeong. A performance-stable NUMA management scheme for linux-based HPC systems. *IEEE Access*, Vol. 9, pp. 52987–53002, 2021.

[20] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pp. 553–564. ACM, 2005.

[21] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pp. 1150–1160. VLDB Endowment, 2007.

[22] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, Vol. 1, No. 3, pp. 189–222, 1976.

[23] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 18–32. ACM, 2013.

[24] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. Nested loops revisited again. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pp. 3708–3717, 2023.

[25] Hayamizu Yuto, Goda Kazuo, and Kitsuregawa Masaru. 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（TOD）, Vol. 7, No. 2, pp. 104–116, 06 2014.