anagodb の高速クエリ実行を実現する細粒度タスク管理実装技術

川道　亮治†　　早水　悠登†　　小沢　健史†　　合田　和生†

† 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: †{kawamichi,haya,ozawa,kgoda}@tkl.iis.u-tokyo.ac.jp

あらまし　東京大学において開発が進められている 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, とりわけデータを選択的に絞り込む分析系クエリにおける高速性を特長とする. anagodb の内部では, 1 つのクエリは多数の細粒度タスクへと動的に分解されて実行される. クエリ処理において発生する負荷は, データ特性や分布, クエリの選択条件などによって都度異なるため, 事前に最適な負荷分散方式を定めることは一般に困難である. そのため, anagodb は実行時の動的なタスク分解と負荷分散を主軸とする実行戦略を採用している. この戦略は, 潜在的には高効率な負荷分散が可能である一方, 生成されるタスク数の事前予測が困難であり, タスク数が増えすぎることで資源を使い尽くして実行が破綻してしまう可能性があるため, 適切な実行時のタスク管理が重要となる. 本論文では, anagodb におけるタスクを主体とするクエリ処理の概要を紹介し, クエリ処理の効率性や適切なタスク数の抑制を実現するための細粒度タスク管理の実装技術について解説する.

キーワード　anagodb, 関係データベースシステム, 細粒度タスク管理, データベースコア技術, 分析クエリ処理

1　はじめに

東京大学において開発が進められている anagodb [4], [11] 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る. データベースコア技術研究の土台となる本格的な関係データベースシステムのコードベースとしてフロムスクラッチで設計・開発され, 主にデータを選択的に絞り込む分析系クエリ処理の高速性に特長を有する. 研究の機動性を維持するためにコード規模を最小限に抑えるべく機能性は絞り込んでいるものの, SQL による TPC-H ベンチマークの実行対応や, 大規模データを取り扱うための高速並列バルクローダ, shared-nothing 型のマルチノードクラスタ構成など, いわゆるトイプログラムの域を脱した本格的な関係データベースシステムとして開発されている.

昨今のデータベースサーバは, 多数のプロセッサコアによる演算資源や, 多数の半導体チップを内包する SSD/NVMe ストレージから構成される広帯域な I/O 資源などが高密度に搭載されるに至っている. データベースシステムにおいてこうしたハードウェア資源を効率的に活用するためには, クエリ処理における並列性の活用が不可欠であり, これまで数多くの並列クエリ処理のアプローチが研究されてきた [1], [6], [9].

anagodb が主たる対象とする選択性の高い分析系クエリでは, 索引を用いた細かい単位のストレージ I/O が多発する傾向にある. また, データ特性やクエリの選択条件によって負荷の発生状況が大きく変動する. そのため, 事前に最適な負荷分散戦略を定めることが一般に困難であり, また効率的な負荷分散を実現できる処理単位の粒度も一様ではない. このことから, anagodb では一つのクエリを多数の細粒度タスクへと実行時に動的分解するという実行戦略を採用している [4].

細粒度タスクへの動的タスク分解という実行戦略は, 潜在的には多数の演算・I/O 資源を効率的に充足させる可能性を有する一方, 個々のタスク生成・管理のオーバーヘッドが生じることや, 生成されるタスク数の事前予測が困難であり, タスク数が増えすぎることで資源を使い尽くして実行が破綻してしまう可能性がある, といった課題が存在する. そのため, 過度なオーバーヘッドの抑制や, 適切なタスク数の維持を実現するための細粒度タスク管理の実装技術が重要となる.

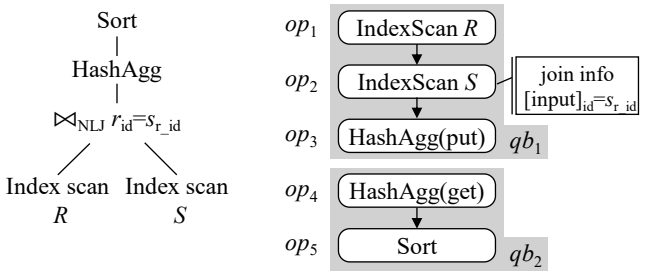
本論文では, 細粒度タスクへの動的タスク分解にもとづくクエリ実行戦略の概要を紹介した後に, 現在の anagodb における細粒度タスク管理の実装技術について解説する.

2　動的タスク分解に基づくクエリ実行戦略

本節では, まずは簡単な実例から動的タスク分解に基づく実行戦略の考え方を説明する.

例えば, テーブル R の全表操作を行うクエリを考える. ここでは, テーブル R の物理編成は, レコードの配列が固定長のページに格納され, 連続するページ $p_1, p_2, \ldots, p_N$ がストレージ上に配置されているものとする. 動的タスク分解に基づく実行戦略では, まず最初に「テーブル R を全表操作する」というタスク t_0 が生成され, 実行が駆動される. テーブル R の各ページに対する操作は独立に実行可能であることから, タスク t_0 はその実行過程で, 「各ページ p_i を走査する」 という子タスク t_i を生成し, 自身が担うべき処理を複数の子タスクへと分解してゆく. タスク t_i は, ページ p_i を走査し, その結果得られたレコードをクエリの出力バッファへと書き込む処理を担う. もしレコード毎に追加の演算処理を行う場合には, 各レコード r に対して「レコードに対して演算を行う」という子タスクを生成してタスク分解してゆくことも可能である.

このように, 動的タスク分解に基づく実行戦略では, クエリ実行で発生する処理のうち, 相互に独立して実行可能な単位をタ



(a) Conventional (b) anagodb

図 1: クエリ実行計画の例: (a) 一般的な関係データベースシステムでは基本的な処理単位をノードとする木構造で表現される一方, (b) anagodb では基本的な処理単位がオペレーションとして線形に接続されてクエリブロックを構成する.

スクとして捉え, 実行時に子タスクとして切り出すことで, 並列実行可能な細粒度タスクの集合へとクエリ処理を分解してゆく. これにより, 多数のプロセッサコアや I/O 帯域を充足させ, 効率的に演算・I/O 資源を活用することを狙いとする実行戦略である.

2.1 クエリ実行計画の構造

一般的な関係データベースシステムにおけるクエリ実行では, 木構造で表現されるクエリ実行計画を用いて, クエリ実行計画の各ノードに対して NEXT 関数を呼び出すことで 1 タプルずつ結果を生成してゆくイテレータプロトコル [3] に基づく戦略を採用するものが多い. この戦略では, クエリ実行計画を構成する各ノードが, 必要に応じて子ノードに対して NEXT 関数を呼び出して入力タプルを取得してクエリ処理を進めてゆく. 入力タプルの必要性が生じた時点で, その生成のための処理が NEXT 関数によって駆動されることから, その形態は demand-driven なデータフロー [3] とも称される.

これに対して, anagodb ではタスクの出力レコードを子タスクの入力として処理が駆動されてゆく. そのため, クエリ処理のうち入力なしに実行可能な処理単位を起点として実行が始まり, その出力をもって次の処理単位を駆動してゆく. Demand-driven なデータフローに対して, このような形態は data-driven なデータフロー [3] とも称される.

図 1 に, 一般的な関係データベースシステムのクエリ実行計画と, それに対応する anagodb のクエリ実行計画の例を示す.

anagodb において, クエリ実行計画は**オペレーション**と**クエリブロック**から構成される. オペレーションはテーブル走査や結合, 集約演算などの基本的なクエリ処理の単位を表し, 線形に接続されてクエリブロックを構成する. オペレーションが接続されている場合, 前段のオペレーションが出力したレコードに基づいて, 後段のオペレーションの処理が駆動される. 集約演算やソートなど, 入力レコードを全て消費しないと結果を生成できないオペレーションも存在し, これらのオペレーションがクエリブロックの終端となる.

2.2 クエリ実行計画とタスク生成によるクエリ実行

anagodb におけるクエリ実行は, このクエリ実行計画のクエリブロックを逐次的に実行してゆくことで進められる. 各クエリブロック実行では, まずクエリブロック先頭のオペレーションに対応するタスクが一つ生成され, その実行を開始する. 例えば図 1(b) では, まずクエリブロック qb_1 のオペレーション op_1 を実行するタスクが生成される. このタスクはテーブル R を索引走査するタスクであり, B$^+$-tree 索引を前提とする場合には索引の内部ノードや葉ノードの探索はそれぞれ独立して実行可能であるので, 動的タスク分解により各ノードの探索を行う子タスクが生成されてゆく. 葉ノードを探索するタスクはテーブル R のレコード r を取得し, 各レコード r を入力として後続のオペレーション op_2 を実行するタスクが生成される.

クエリブロック qb_i の最初のタスクを発端として生成された全てのタスクが完了した時点でクエリブロックの実行が完了し, 次のクエリブロック qb_{i+1} の実行へと移行する. クエリブロックの終端となるオペレーションの種類 (集約演算, ハッシュ結合のためのビルド演算, ソートなど) ごとにその実行結果を格納する専用のバッファが用意されており, クエリブロック間のデータ受け渡しはこれらのバッファを介して行われる.

一般的なクエリ実行計画におけるノードと, anagodb のクエリ実行計画におけるオペレーションは, テーブル走査など基本的な処理単位に対応することや, レコードを入出力単位とするという点では共通するが, anagodb のオペレーションは線形に接続されている点が異なる. そのため, 一般的なクエリ実行計画において二つの入力をとるノードに直接対応するオペレーションは存在せず, 等価な処理を記述する際にはオペレーションやクエリブロックの組み合わせによって表現される. 例えば図 1 におけるネステッドループ結合は, テーブル R のレコード r それぞれに対して, 対応する S を索引走査して結合を実行する処理を表している. anagodb では, オペレーション op_1 が出力するレコード r ごとにタスクが生成され, 後続のオペレーションとして S を索引走査するオペレーション op_2 が駆動されるため, op_2 の処理完了時に入力レコード r と結合処理を行うことでネステッドループ結合と等価な処理を実現することができる.

2.3 タスクの実行状態とライフサイクル

anagodb におけるタスクは生成から完了に至るまで複数の状態を遷移し, その状態に基づいて実行のスケジューリングが行われる. タスクはその実行を中断, 再開することが可能であり, anagodb のバッファ管理層や I/O 管理層と連携するコルーティン [7] であると見なすことができる.

クエリ実行を担うメインループ処理を Algorithm 1 に, またタスクの処理中に利用される制御フローのための補助関数を Algorithm 2 に示す. 以降の説明では, 行番号について注記なしで言及する場合には, Algorithm 1 の行番号を指すものとする.

タスクが生成され, 実行可能な状態になると, そのタスクは *runnable* 状態となる. Runnable 状態のタスクは実行スケジューラによって適宜選択され, 実行が始まると *running* 状態となる (行 16–18). タスクの実行が完了すると *completed* 状態

Algorithm 1 anagodb のクエリ実行メインループ

```
 1: function ExecuteQueryPlan(plan p, context ctx)
 2:     for each qb in p.queryBlocks do
 3:         t_0 ← CreateTask(qb.operations[0])
 4:         t_0.context ← ctx
 5:         Push t_0 into runnable task queue
 6:         ctx.nRunnable ← 1
 7:         ctx.nRunning ← 0
 8:         ctx.nSuspended ← 0
 9:         Suspend thread until RemainingTasks(ctx) = 0
10: function RemainingTasks(context ctx)
11:     n ← ctx.nRunnable +ctx.nRunning +ctx.nSuspended
12:     return n
13: function MainLoop
14:     while true do
15:         if runnable task queue is not empty then
16:             t ← Pop a task from runnable task queue
17:             ctx ← t.context
18:             ctx.nRunnable−−; ctx.nRunning++
19:             if t is suspended before then        ▷ suspended task
20:                 ⇝ ResumeTask(t, ctx)
21:             else                                 ▷ newly created task
22:                 ExecuteTask(t, ctx)
23:             if t is completed then
24:                 ctx.nRunning−−
25:                 ReleaseTask(t)
26:             else if t is suspended then
27:                 // Jump back with SuspendTask
28:         if any IO request req is completed then
29:             t ← req.issuerTask; ctx ← t.context
30:             ctx.nSuspended−−; ctx.nRunnable++
31:             Remove t from suspended task queue
32:             Push t into runnable task queue
33:         if any subquery subq is completed then
34:             t ← subq.issuerTask; ctx ← t.context
35:             ctx.nSuspended−−; ctx.nRunnable++
36:             Remove t from suspended task queue
37:             Push t into runnable task queue
```

となり, 当該タスクに関連する資源が解放される (行 23–25).

タスクの処理を進行させるためにストレージ I/O やネットワーク I/O が必要となる場合, 当該タスクは SuspendWithIO 関数を呼び出すことでその実行を中断し, *suspended* 状態となり, ストレージやネットワークに対して I/O リクエストが発行される. このとき, 行 20 または行 22 から呼び出されていたタスク実行処理からメインループへと制御が戻される. SuspendWithIO 関数の中では, I/O リクエストの非同期発行が行われ, 現在実行中のタスクを suspended 状態に遷移させた後に, プログラムの継続状態を保存してメインループへと制御を戻す (Algorithm 2 行 6) [1]. 非同期に発行された I/O リクエストの状態はメインループ内でポーリングされ (行 28–32), I/O リクエスト完了時には当該リクエストを発行したタスクが runnable 状態に復帰する.

またタスクが中断される理由には, I/O リクエストを待つ以外にもサブクエリの結果待ちが存在する. この場合には SuspendWithSubquery 関数が呼び出され, サブクエリに紐付く一連のタスク群の完了待ちを行うタスクもまた suspended 状態となる. メインループ内ではサブクエリの完了状態がポーリングされ (行 33–37), サブクエリのタスク群が全て完了すると, 当該サブクエリを発行したタスクが runnable 状態に復帰する.

ユーザーからクエリ実行が要求されると, クエリ実行を駆動するために ExecuteQueryPlan 関数が呼び出され, 並行するスレッドで実行されているメインループが全てのタスクを完了するまで待機する (行 9). このステップを全てのクエリブロックに実行してゆくことで, クエリ全体の実行が完了する.

Algorithm 2 anagodb のタスク実行制御のための補助関数

```
 1: function SuspendWithIO(task t, IO request req)
 2:     req.issuerTask ← t; ctx ← t.context
 3:     Issue IO request req asynchronously
 4:     ctx.nRunning−−; ctx.nSuspended++
 5:     Push t into suspended task queue
 6:     ⇝ Save the state of t and jump back to MainLoop
 7:     ⇜ // resumed by ResumeTask in MainLoop
 8:     return the result of completed IO request req
 9: function SuspendWithSubquery(task t, subquery subq)
10:     subq.issuerTask ← t ; ctx ← t.context
11:     ctx′ ← new context for subq
12:     for each qb in subq.queryBlocks do
13:         t_0 ← CreateTask(qb.operations[0])
14:         t_0.context ← ctx′
15:         Push t_0 into runnable task queue
16:         ctx′.nRunnable ← 1
17:         ctx′.nRunning ← 0
18:         ctx′.nSuspended ← 0
19:         ctx.nRunning−−; ctx.nSuspended++
20:         Push t into suspended task queue
21:         ⇝ Save the state of t and jump back to MainLoop
22:         ⇜ // resumed by ResumeTask in MainLoop
```

3 効率的な細粒度タスク管理

動的タスク分解に基づくクエリ実行戦略に基づいて関係データベースシステムの実装を行う上では, クエリ実行に際して生成される多数の細粒度タスクを効率的に管理することが重要である. 本節では, anagodb における細粒度タスク管理の実装技術について解説する.

3.1 NUMA アウェアなタスク管理構造

昨今のデータベースサーバでは多数のコアを持つプロセッサが複数搭載される NUMA アーキテクチャが主流となっており, プロセッサコア間およびソケット間のメモリアクセス競合を可能な限り抑制することが実行効率において重要である.

1：anagodb においては, スタック領域の待避と `setjmp`,`longjmp` により継続の保存と復帰を実現している.

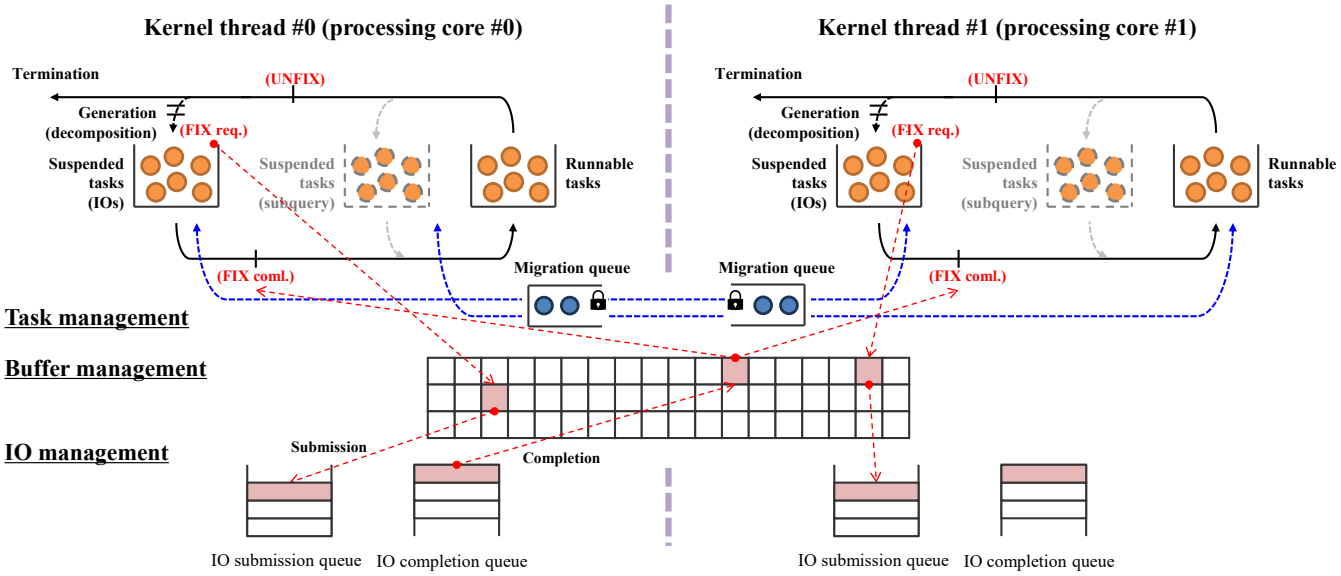


図 2: anagodb における細粒度タスク管理の概要

図 2 に, anagodb におけるタスク管理構造の概要を示す. anagodb のタスク管理層のデータ構造は基本的に各プロセッサコア毎に分離しており, 各プロセッサコアが利用するメモリ領域は当該コアが属する NUMA ノードにおいて確保される *local binding* ポリシー [5] を原則としている. すなわち, runnable タスクキューや suspended タスクキュー, タスクコンテキストなどのタスク管理に関するデータ構造は各プロセッサコア毎に分離されて管理される. また, 各プロセッサコアには専用のカーネルスレッドを 1 つ割当て, 当該カーネルスレッドが Algorithm 1 に示すメインループを実行することで, 当該プロセッサコア上でのタスク実行を担当する.

各スレッドが保持するタスク数に一定以上の偏りが生じた場合には, タスクマイグレーションを行うことにより負荷分散を行う. 各スレッドは専用のキューを保持しており, 他スレッドはこのキューにタスクを追加することで, 当該スレッドにタスクをマイグレーションさせることができる. タスク管理層において発生するロック競合は, 基本的にこのマイグレーションキューに対するロック競合のみとなるよう設計されている. マイグレーションキューのアクセス頻度を抑制することで, 多数のスレッドが可能な限り相互干渉することなく実行可能な構造となっている.

バッファ管理層におけるバッファプールはプロセッサコア間で共有されるが, メニーコア環境において優れたスケーラビリティを示すことが知られている Nb-GCLOCK [10] によるバッファページ管理を採用し, 毎秒 100 万 I/O を超えるバッファページの FIX/UNFIX 処理が可能であることを確認している. また性能試験用途として, コア間で重複したページがキャッシュされることを許容し, 各プロセッサコア毎にバッファプール自体も完全に分離する動作モードもサポートしている.

3.2 タスクの優先度制御と動作モード制御

図 1(b) の例で示したように, anagodb ではクエリ実行計画

Algorithm 3 anagodb のタスクマイグレーションポリシー

```
// Returns ID of the destination thread for migrating a task,
// or NULL if no migration is needed
N := # of threads
T(k) := # of tasks in thread k
θ := migration threshold
function MigrationDestThread
    // Phase 1: Check the need for migration
    T̄ = Σ_{k=0}^{N-1} T(k) / N
    if T(k_self) < T̄ then
        return NULL                          ▷ No migration needed
    // Phase 2: Select a destination thread for migration
    for i in 1 to N − 1 do                   ▷ Check in a round-robin manner
        k_dest ← (k_self + i) mod N          ▷ Starts from the neighbor
        if T(k_dest) < θ × T(k_self) then
            return k_dest
    return NULL
```

の各クエリブロックを逐次的に実行する仕組みを採用し, クエリブロック先頭のオペレーションが起点となり, 順次後続のオペレーションを実行するタスクが生成されてゆく. すなわち, 前段のオペレーションを実行するタスクであるほど, そのタスクから派生して生成される子孫のタスク群の数は多く, まだ後段のオペレーションを実行するタスクであるほど, そのタスクから派生して生成される子孫のタスク群の数は少なくなる.

anagodb の動的タスク分解に基づくクエリ実行戦略の特性上, ハードウェア資源を効率的に活用するためには, 一定数以上のタスクが各プロセッサコアに供給され続けることが望ましい. しかしながら, タスク管理に割当て可能なメモリ資源は有限であり, 過度に多くのタスクが生成されすぎるとクエリ実行の継続そのものが困難となってしまう.

このため, anagodb ではタスクの優先度制御と, タスク数の制限超過時の動作モード切替えによってクエリ実行の安定性を

確保している．

3.2.1 タスクの優先度制御

anagodb では, タスクが現在実行しているオペレーションの先頭からのオフセット番号をタスクの優先度として設定し, 優先度の高いタスクから実行されるようにスケジューリングを行っている．優先度が高いタスクは, すなわちクエリブロックにおいてより後段に位置するオペレーションを実行しているタスクであり, そのタスクを実行しても新たな子タスクが多数生成される見込みが低く, タスクキューのタスク数を過度に増加させる可能性が低い．

ただし, 各プロセッサコアのタスクキューが保有するタスク数が一定数以下の場合には, 優先度制御を逆転させ, 優先度の低いタスクから実行されるようにスケジューリングを行う．これにより, 子タスクを多数生成することが見込まれる, クエリブロックのより前段にあるオペレーションを実行するタスクが優先的に実行される．この優先度制御を逆転させる閾値は実行環境によって最適な値は異なると考えられるが, 現在は経験的に多くの環境で良好な性能を示す値として 100 タスクを基準値として採用している．

タスク生成の親子関係を木構造として捉えた場合, 優先度の高いタスクを先に実行する制御は深さ優先探索として, 優先度の低いタスクを先に実行する制御は幅優先探索として捉えることができ, この木構造の探索中の境界領域に位置するのが, その時点でのタスクキューに保持されているタスク群に相当する．優先度が高いタスクと低いタスクのスケジューリングを切替える事で, 保持されているタスク数を増えすぎない範囲で維持することを狙う制御となっている．

3.3 タスクのプロセッサコア間負荷分散

anagodb では local binding ポリシーに基づき, 生成されるタスクは原則として生成元タスクが実行されていたプロセッサコア上のタスクキューに追加され, 実行を待つことを基本とする．ただし, プロセッサコアがタスクキューに保持する runnable タスク数が 0 となった場合には, 当該コアはアイドル状態となり実行効率が低下してしまうため, プロセッサコア間でタスクを交換するタスクマイグレーション機構を備えている．

タスクマイグレーションはプロセッサコア間のコヒーレンシ管理コストや, 排他制御によるプロセッサコアのストール発生などのオーバーヘッドを伴うため, できるだけマイグレーションの発生を抑制することが望ましい．ハードウェア資源の活用においては, 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, 必ずしも全てのスレッドが均等にタスクを保持している必要はない．そのため, anagodb のタスク負荷分散ポリシーはタスク数の厳密な均衡は求めず, 各スレッドが一定程度のタスク数を保持する状況を維持することを目標として, タスクマイグレーションポリシーを設定している．

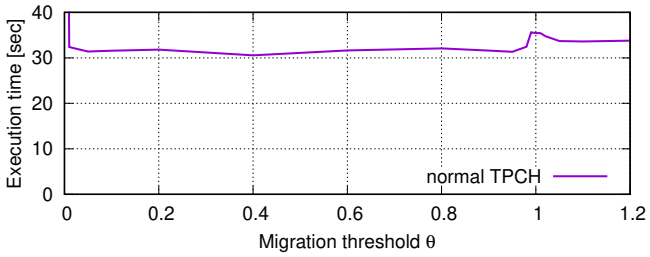
Algorithm 3 に, anagodb のタスクマイグレーションポリシーを示す．anagodb の各スレッドは, 新たにタスクが生成されるたびに MigrationDestThread 関数を呼び出し, タスクマイグレーションの必要性を確認する．MigrationDestThread 関数が NULL 以外のスレッド ID を返した場合には, 当該スレッドのマイグレーションキューにタスクを追加する．各スレッドは, メインループにおいて runnable タスクを確認する前にマイグレーションキューを確認し, マイグレーションキューにタスクが存在する場合には, 当該タスクを自身の runnable/suspended タスクキューに追加することで, マイグレーションされたタスクを実行可能な状態にする．

MigrationDestThread 関数におけるマイグレーション先のスレッド選択は二段階で行われる．まずフェーズ 1 では, マイグレーションの必要性を確認する．具体的には, 自身のスレッドが保持する runnable タスク数が, 全スレッドの平均値よりも少ない場合には, マイグレーションは不要と判断し, NULL を返す (行 6–8)．次にフェーズ 2 では, マイグレーション先のスレッドを選択する．具体的には, 自身のスレッド ID を起点として, 隣接するスレッドから順に, 当該スレッドが保持する総タスク数が, 自身のスレッドが保持する総タスク数の一定割合未満であるスレッドを探索し, 最初に見つかったスレッドをマイグレーション先として返す (行 10–13)．この際, 判定に用いるタスク数の比率をマイグレーション閾値 θ として設定している．$\theta = 0$ の場合にはタスクマイグレーションは一切発生せず, クエリ開始時に最初のタスクが投入されたスレッド内でのみタスクが実行され, 実質的にシングルスレッド実行となる．$\theta > 0$ では θ の値が大きいほどタスクマイグレーションが発生しやすくなり, $\theta = 1$ では自身より少ないタスク数を持つスレッドをマイグレーション先として選択する状態となる．

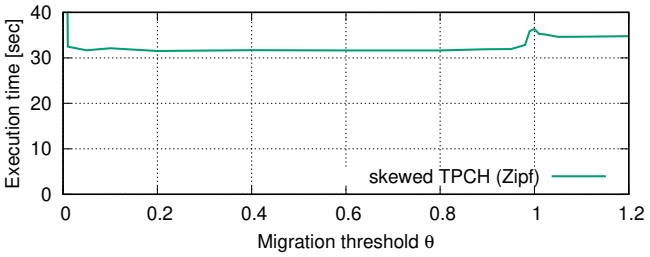
3.3.1 動作モード制御

anagodb の基本的な動作モードはアウトオブオーダ型実行方式 [2] を基本とするものであり, 動的タスク分解によって多数の細粒度タスクが生成されることで, ハードウェア資源を効率的に活用することを狙いとしている．しかしながら, タスク数が過度に増加してしまった場合には, システム全体のメモリ資源を使い尽くしてしまい, 新たなタスクの生成が不可能となってしまうリスクが存在する．

このため, anagodb ではタスク数が事前に定めた閾値を超過した場合には, 動作モードをインオーダ型実行方式へと切替える仕組みを備えている．インオーダ型実行方式では, 新たに子タスクを生成する際に, タスクキューへの追加は行わず, 子タスクを実行する関数を直接呼び出す．そして, 子タスクの完了後に親タスクの処理を再開する形態となる．また, タスク実行中の I/O リクエストは全て同期的に処理完了を待つ形態へと切替える．すなわち, これまでは動的タスク分解によって子タスクとして処理が分岐していたものを, 単なる関数呼び出しのコールスタックを利用した逐次処理と切替えた動作モードであると言える．インオーダ型実行方式は, 3.2.1 節で述べたタスク生成の親子関係を木構造として捉えた場合には, 完全に深さ優先探索として処理を進めることに相当する．これにより, タスク実行によって新たに生成されるタスクの同時に存在しうる数はクエリ実行計画のオペレーション数に比例する範囲に抑制されるため, 予測可能な有限のメモリ資源で処理を継続することが可能となる．



(a) 標準 TPC-H データセット



(b) 偏りあり TPC-H データセット (Zipf 分布)

図 3: タスクマイグレーション閾値 θ とクエリ実行時間: (a) 標準 TPC-H データセット, (b) 偏りあり TPC-H データセット (Zipf 分布) いずれにおいても, $0 < \theta < 1$ の殆どの範囲でクエリ実行時間は安定し, θ の値に対してロバストであることが分かる.

表 1: 性能評価実験環境

CPU	Intel Xeon Gold 6442Y × 2 (24 cores each)
Memory	DDR5 256 GB
Storage (OS)	NVMe SSD 447 GB
Storage (DB)	NVMe SSD 7.68 TB × 8 (RAID 0, 40 TB)
OS	Ubuntu (Linux kernel 5.15)

表 2: 閾値 θ とタスクマイグレーション発生回数・実行時間

	標準 TPC-H		偏りあり TPC-H	
θ	回数	実行時間	回数	実行時間
0.95	2,364,530	31.8 sec	2,368,370	32.2 sec
1.00	64,034,493	37.8 sec	63,590,026	36.3 sec
1.05	46,294,925	34.6 sec	46,734,452	34.7 sec

4 性能評価

マイグレーション閾値 θ によって anagodb のクエリ処理性能がどの程度影響されるかを確認するため, TPC-H データセットを用いた性能評価実験を行った. この実験では, 標準の TPC-H データセット [8] に加えて, 結合のカーディナリティに Zipf 分布に従った偏りが生じるように加工したデータセットも用いている. いずれもスケールファクター 100 でデータセットを生成し, anagodb にロードしてデータベースを構築している. 具体的には, customer 表 1 レコードあたりの orders 表レコード数, orders 表 1 レコードあたりの lineitem 表レコード数が, 平均値は標準データセットと同一でありつつ, Zipf 分布に従うようにデータを加工している. 評価に用いたクエリは付録に示すものであり, customer, orders, lineitem, partsupp, supplier, nation を順にネステッドループ結合と索引検索を用いて結合してゆくクエリである. 実験は表 1 に示す 48 コア環境を用いて行い, anagodb は 48 スレッドで実行している.

実験結果を図 3 に示す. 図 3a は標準 TPC-H データセットを用いた場合の結果であり, 図 3b は偏りあり TPC-H データセットを用いた場合の結果である. いずれの結果においても同様の傾向が見られ, マイグレーション閾値 θ が 0 よりも僅かに大きくなり, スレッド間でタスクマイグレーションが発生するようになった段階でクエリ実行時間が大幅に短くなり, θ が 0.95 程度まではほぼクエリ実行時間が一定の値を取っている. これは, タスクマイグレーションの発生頻度を調整する閾値 θ の値に対して, anagodb の処理性能が一定程度ロバストであり, 精緻なチューニングを行わなくとも, ハードウェア性能を十分に活用し良好な性能を得られることを示している.

図 3 において, $\theta = 1$ 前後ではクエリ実行時間が境界的な振る舞いを示している. $\theta = 0.95, 1.00, 1.05$ のそれぞれについて, クエリ実行によって発生したタスクマイグレーションの回数を表 2 に示す. $\theta = 1.00$ では, 各スレッドは自身とほぼ同数のタスクを持つスレッドもマイグレーション先として選択可能となるため, タスクマイグレーションの発生回数が約 26〜27 倍と大幅に増加し, これによって実行時間が増加していると考えられる. 一方で, $\theta = 1.05$ では $\theta = 1.00$ よりも 5〜9%程度実行時間が短く, タスクマイグレーション回数も約 26〜27%減少している. θ の値が大きくなると, Algorithm 3 のフェーズ 2 においてはより積極的にマイグレーションを行う判断がなされるが, これによってスレッド間のタスク数より早く均衡し, Algorithm 3 のフェーズ 1 においてマイグレーションの必要性が否定されるケースが増えるため, マイグレーション回数が減少する傾向が見られると考えられる.

以上の結果より, anagodb が現在採用するタスクマイグレーションポリシーは, 各スレッドで生成されるタスクが Zipf 分布に従って一定程度偏っていたとしても, 適切にタスクを各スレッドに分散させ, マルチコア環境におけるハードウェア資源を高効率に活用していることが確認された. また, タスクマイグレーション閾値 θ の値に対する anagodb の処理性能のロバスト性についても確認された.

5 おわりに

本論文では, 関係データベースシステム anagodb における細粒度タスク管理の実装技術について解説した. 動的タスク分解に基づくクエリ実行戦略は, 一般的な関係データベースシステムで多く採用されている demand-driven なデータフローとは異なり, data-driven なデータフローに基づいてクエリ実行が駆動される. クエリ実行の処理において並列実行可能な処理を起点として, 実行時に動的に細粒度タスクへと分解してゆくこと

で, 多数のプロセッサコアや I/O 帯域を充足させ, 効率的に演算・I/O 資源を活用することを狙う実行戦略である. この実行戦略を実際の関係データベースシステムに実装する上では, 多数の細粒度タスクを効率的に管理するとともに, タスク数の過度な増加によるシステム資源の枯渇を防止することが重要である. これを実現するための実装技術として, 本論文では anagodb における NUMA アウェアなタスク管理構造とコア間のタスク負荷分散機構, およびタスクの優先度制御と動作モード制御によるタスク数の抑制技術について解説した. TPC-H データセットを用いた性能評価実験では, タスクマイグレーション閾値 θ の広い範囲にわたってクエリ実行時間が安定しており, 精緻なパラメータチューニングを要さずに良好な性能が得られることを確認した. また, データ分布に Zipf 分布に従う偏りがある場合でも同様の傾向が見られ, anagodb のタスクマイグレーション機構がデータ特性に依存せず効果的に負荷分散を実現していることを示した.

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.

文　　献

[1] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hetexchange: encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proc. VLDB Endow.*, Vol. 12, No. 5, pp. 544–556, January 2019.

[2] 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.

[3] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, Vol. 6, No. 1, pp. 120–135, feb 1994.

[4] Yuto Hayamizu, Ryoji Kawamichi, Tsuyoshi Ozawa, Masaru Kitsuregawa, and Kazuo Goda. anagodb: Offering massive parallelism for database engine. In *Proceedings of the 2025 International Conference on Management of Data*, SIGMOD/PODS '25, pp. 115–118, New York, NY, USA, 2025.

[5] 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.

[6] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 743–754, New York, NY, USA, 2014. Association for Computing Machinery.

[7] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 31, No. 2, pp. 1–31, 2009.

[8] Transaction Processing Performance Council. TPC-H benchmark specification. Standard Specification Revision 3.0.1, TPC, 2024.

[9] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. Elastic pipelining in an in-memory database cluster. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1279–1294, New York, NY, USA, 2016. Association for Computing Machinery.

[10] Makoto Yui, Jun Miyazaki, Shunsuke Uemura, and Hayato Yamana. Nb-gclock: A non-blocking buffer management based on the generalized clock. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pp. 745–756, 2010.

[11] 早水悠登, 川道亮治, 小沢健史, 合田和生. 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. 第 17 回データ工学と情報マネジメントに関するフォーラム (DEIM2025), 3 2025. 9C-01.

付　　録

1　性能評価用クエリ

```
SELECT N.N_NAME, SUM(PS.PS_SUPPLYCOST),
 SUM(L.L_QUANTITY) AS SUM_QTY,
 SUM(L.L_EXTENDEDPRICE*(1-L.L_DISCOUNT)*(1+L.L_TAX))
    AS SUM_CHARGE
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;
```