DEIM Forum 2023 2a-1-5

動的複数クエリ最適化の検討とその潜在的有効性の確認

木村　元紀†　　早水　悠登††　　合田　和生††

† 東京大学 情報理工学系研究科　〒 113–8656 東京都文京区本郷 7–3–1
†† 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: †{kimura-g,haya,kgoda}@tkl.iis.u-tokyo.ac.jp

あらまし　SQL 等で宣言的に記述されたクエリの効率的実行のためには、多数存在する実行計画候補から効率的なものを選択するクエリ最適化が重要である。さらに、データベースに入力されるクエリは類似した処理を含むことに注目し、複数のクエリを同時に考慮することにより大域的に最適な実行計画を得る複数クエリ最適化も提案されている。しかし、既存の複数クエリ最適化は処理開始時点でクエリ最適化対象のクエリが全て揃っている必要があり実用的ではない。そこで本研究では複数クエリ最適化を動的に行い実行中に新たに入力されたクエリを考慮して再度クエリ最適化を行う動的クエリ最適化を提案する。また、本手法の潜在的な有効性を TPC-H ベンチマークを用いたシミュレーションにより検証した。

キーワード　問合せ最適化, 問合せ処理, データベース技術

1　はじめに

生成・蓄積されるデータは近年ますます増加しており、ビッグデータを素早く解析可能な基盤システムの構築は必要不可欠である。データベース管理システム (DBMS) はビッグデータ解析基盤として広く使われているシステムであり、その中でも関係データベース管理システム (RDBMS) は最も汎用的なものの一つである。RDBMS では SQL 等で宣言的に記述されたクエリが入力されると、クエリから実行計画を生成し、この実行計画に基づいてクエリの実行を開始する。実行計画は一つのクエリに対して多数存在するため、その中から実際に実行する実行計画を選択するクエリ最適化が実行性能向上のために重要である。

クエリ最適化では、実行計画は論理的実行計画と物理的実行計画の 2 つに分けられる。論理的実行計画は入力となる関係代数（テーブル）を葉ノード、関係代数上の演算を内部ノードとする木構造であるクエリ木によって表現され、各演算子の結合順序や適応順序が規定される。クエリ木上の内部ノードの入れ替えやクエリ木の回転等によってクエリの実行結果を変えないままに論理的実行計画を複数得ることができる。物理的実行計画は与えられた論理的実行計画中の各演算子（内部ノード）の物理的な実装を与える。例えば 2 つの関係代数を結合する結合演算子はよく知られた実装だけでも Nested-loop Join や Merge-sort Join, Hash Join など様々な種類があり、それぞれの実行コストは結合される関係代数の大きさや選択率によって大小関係が異なるためクエリ最適化によってどの実装を選択するかを決定する必要がある。

現代の DBMS は同時に多数のクライアントから利用されるため、頻繁に複数のクエリを同時実行する。しかし、一般の DBMS は各クエリを独立に実行し、各クエリ処理に類似の処理が含まれていてもその類似性を活用することができない。複数クエリ最適化はこの類似性を活用し、データや処理を複数クエリ処理間で共有することで実行を効率化することを目指す。RDBMS の実行性能の指標はクエリ実行のレイテンシやスループット、電力消費量など複数考えられるが、複数クエリ最適化ではデータや処理の共有により全体として必要な処理量が減少するので特にスループットを重視する RDBMS において実行性能を向上させることができる。

本論文では新たな複数クエリ最適化手法として、動的複数クエリ最適化を提案する。既存の複数クエリ最適化はバッチクエリ最適化、すなわち実行の開始前に最適化の対象となるクエリが全て揃っている必要があるが、動的複数クエリ最適化は既に進行している実行に対して新たに入力されたクエリを含めたクエリ最適化を可能とする。動的複数クエリ最適化によりデータや処理が共有される範囲がさらに拡大するためより効率的な実行が可能になると考えられる。

動的複数クエリ最適化のシミュレータを実装し、シミュレータ上で TPC-H ベンチマークを実行することにより、潜在的有効性を検証した。その結果、クエリ実行は複数クエリ最適化に対して平均 15%、最大 60%の高速化が可能であることが明らかになった。

本論文は本章を含めて 5 章から構成される。第 2 章ではクエリ最適化、特に適応的クエリ最適化及び複数クエリ最適化の既存研究について述べる。第 3 章では動的複数クエリ最適化を導入し、これを実現するためのいくつかの手法を紹介する。第 4 章ではシミュレーションにより動的複数クエリ最適化の潜在的有効性を検証した結果を示す。第 5 章では本論文の総括を行い、今後の研究課題についても述べる。

2　クエリ最適化

2.1　適応的クエリ最適化

通常のクエリ処理ではまずクエリ最適化を行って実行計画を

生成し、その後得られた実行計画に基づいてクエリ処理を行う。しかし、クエリ最適化時点で想定していた実行環境やデータの統計的性質が実際とは異なっていた場合、得られた実行計画は良い実行計画ではなく、効率的な実行を行うことはできない。適応的クエリ最適化は実行時情報を用いて都度クエリ最適化を行い、得られた実行計画を現在進行中の実行に反映することで、より効率的なクエリ処理を達成する。通常の実行計画はその実行計画に基づいて最初から最後まで実行を行うことを仮定しているため、実行がどのような中間状態を取るかは実行コストの推定以外の部分には現れない。しかし、適応的クエリ最適化は実行計画を実行時に変更するため、再クエリ最適化によって実行計画が変更された時に中間状態をその変更に合わせて変換できるかが重要である。どのような中間状態を取るかは論理的実行計画だけでは不十分で、物理的実行計画のレベルで実装を考慮する必要がある。

Eddy [2] は各演算子に対してタプルを動的にルーティングすることで、実行中の演算順序の入れ替えを可能にする。最も簡単には複数の選択演算子を通過すべきタプルがある時に選択率のより小さい演算子を先に通過するようにルーティングすることで全体としての処理量を削減する。結合演算子は交換法則を保つが、Nested-loop Join や Hash Join など演算子の左右によって実行コストが大きく変わる実装がよく用いられるため、結合演算子の左右交換も適応的クエリ最適化において重要である。Eddy は結合演算の各実装について中間状態を変えないままに演算子の左右を入れ替え可能な時点（Moment of Symmetry, MoS）を明らかにし、その時点でのみ再最適化を許すことで、選択演算子の時と同様にタプルルーティングによって結合演算子についても適応的クエリ最適化を可能にした。Hash Join は MoS を持たないが、Symmetric Hash Join(SHJ) [9,12] はハッシュ表を冗長に作成することにより常に左右を交換可能にした Hash Join である。Eddy において Hash Join の代わりに SHJ を採用することで適応的に Hash Join を考慮してクエリ最適化が可能になる。State Module(STeM) [8] は SHJ 演算をハッシュ表構築 (build) とハッシュ表検索 (probe) の二つの演算に分離することで、二項演算子を単項演算子の組み合わせとしてみなせるようにした。これにより Eddy におけるスケジューリングの自由度がさらに増加し、より柔軟な適応的クエリ最適化が可能となった。

従来の計算機上でのクエリ処理はディスクとのデータ入出力がボトルネックであったが、SSD 等の高速な補助記憶装置の発達や全データを主記憶装置上に配置するインメモリデータベースの出現により、プロセッサレベルの高速化が処理全体の高速化につながりはじめている。また、プロセッサの進歩により SIMD 等のベクトル演算の活用も高速化に重要な役割を果たすようになってきた。これらの高速化のためには実行されるネイティブコードレベルの最適化つまりコンパイラ最適化が必要になるが、ネイティブコードへのコンパイルにはクエリ処理にかかる時間に対して無視できない時間がかかる場合があるので、ネイティブコードレベルの適応的なクエリ処理は一般に難しい。Micro Adaptivity [7] は Vectorwise データベース [13] 中での用いられているネイティブコードレベルの適応的クエリ最適化フレームワークである。Micro Adaptivity では、各演算子に対して様々な高速化手法を適用した複数の実装を用意したり、その実装をいくつかのコンパイラでコンパイルすることで様々なバイナリを事前に用意する。実行時には ε-greedy のような強化学習手法を用いてより高速なバイナリを探索しながら実行を進めていくことで効率的な実行を可能にする。

Permutable Compiled Query(PCQ) [5] はネイティブコードレベルの適応的クエリ最適化に JIT コンパイルを組み合わせている。Micro Adaptivity のように通常の DBMS では事前に実行されるバイナリを全て用意しておく必要があるが、JIT コンパイルを用いることで必要になったその時に当該バイナリをコンパイルして用意することができる。しかし、実行中のクエリ最適化の度に毎回 JIT コンパイルを行っているとコンパイルの実行コストが JIT コンパイルによる利得を上回ってしまうため、PCQ では実行計画を適度な粒度に分解し各粒度ごとにコンパイルを行い、実行中に得られた各バイナリの実行順序を適応的に制御することで JIT コンパイルのコストを削減しつつ、実行性能の向上を達成している。

2.2 複数クエリ最適化

最も基本的な複数クエリ最適化手法として共通部分式共有 [6,10] がある。共通部分式共有は同時に入力された各クエリについてあらかじめ論理的実行計画を計算しておき、それら複数の実行計画中に同じ部分式が出現していれば、この部分式の実行結果を共有することで実行コストを削減する。元々の実行計画はテーブルを葉ノードとする木構造をしているが、共通部分式除去によって得られる大域的実行計画は有向非巡回グラフの形式をしている。つまり共有された演算子は 1 つ以上の親演算子に自身の出力を入力する。

共通部分式共有は部分式が完全に同一である場合に限って処理を共有するためクエリ最適化の可能な範囲が制限されるが、実際には部分式が同一でなくても選択演算子のみが異なる場合であれば共有可能である。データパスモデル [1,4] はこのような状況でも処理が共有可能となる処理モデルである。このモデルでは実行されている各クエリ処理に必要か不必要かを示すフラグを処理中のタプルに付与することで、あるクエリの選択演算子では棄却されるが他のクエリの選択演算子では採用されるような場合に対応する。データパスを用いた複数クエリ最適化では、はじめに各クエリに対してデータパスを作成し、その後それらをまとめて単一のデータパスに合成して実行する。この合成は従来は発見的方法によって行われていたが、[3] では最適化問題を解くことにより厳密にコストが最小なデータパスを得る。しかし、この最適化問題は非常に複雑であり計算量クラスは NP 困難であるため、あらかじめ入力されうるクエリ集合が明らかな場合にしか用いることができない。

[11] はデータパスベースの複数クエリ最適化と Eddy ベースの適応的クエリ最適化を組み合わせることで高効率な複数クエリの同時実行を可能にする。従来の発見的なデータパスの合成法を強化学習によって学習しながら実行を進めていくことによ

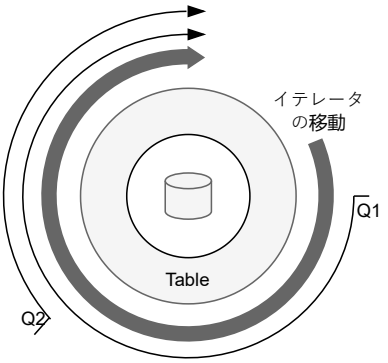

図 1: 逐次スキャンの統合

り、高コストな最適化問題の厳密解を求めることなく同程度の実行時効率を実現している。一方、Eddy ベースの適応的クエリ最適化を組み合わせたことにより結合演算子実装として SHJ しか用いることができない等の制約も存在する。

3 動的複数クエリ最適化

既存の複数クエリ最適化は全てクエリ最適化の対象となるクエリ群があらかじめわかっているため、クエリ最適化の解空間は静的に決まっている。動的複数クエリ最適化では、既に進行中のクエリ処理に対して新たに入力されたクエリを考慮して再度クエリ最適化を行うことで、より効率的な複数クエリの同時実行を目指す。動的複数クエリ最適化が提供する実行効率化は 2 種類ある。第一に、クエリ最適化の対象となるクエリが実行中に新たに入力されたクエリにまで拡大されることにより、クエリ最適化の対象空間が広がり、今までは選択できなかった実行計画を用いた真に大域的に最適な実行計画に到達できるようになる。第二に、動的複数クエリ最適化は実行中の実行計画の変更を可能にするため、事前に厳密にクエリ最適化を解くことができなくても、実行の途中から厳密解に基づいた実行に変更することで、未知のクエリが入力される場合でも厳密解を用いることができる。また、実行計画の変更が可能になることで強化学習を用いたクエリ最適化手法 [11] とも組み合わせることが可能である。

3.1 スキャン操作の統合

データベースの各テーブルに対するスキャン操作の統合は動的複数クエリ最適化によって可能となる処理の共通化手法の一つである。スキャン操作はおおまかに逐次スキャンとインデックススキャンの 2 種類がある。逐次スキャンはテーブルの全タプルを逐次的に読み取っていく操作であり、インデックススキャンはテーブルに対して構築されたインデックスを用いて特定の値を持つタプルのみを読み取る操作である。

複数の逐次スキャンを動的に統合する手法を図 1 に示した。通常の逐次スキャンはテーブルの先頭から開始し終端で終了するが、最終的にソートされる場合など実際にはどのような順番で読み取られるかは問題でない場合は多い。読み取り位置を示すイテレータとして循環イテレータを用いると、イテレータがテーブルの終端に達したらその次は先頭に戻ってくるようにすることができる。読み取りを開始した時のイテレータが指していた位置を記録しておくことによって、どの位置からでも逐次スキャンを行うことができる。あるクエリ Q1 の処理として逐次スキャンを実行しているときに、同じテーブルに対する逐次スキャンを必要とするクエリ Q2 が入力された場合を考える。この時、通常であれば Q1 の実行が完了してから Q2 の実行を行うが、循環イテレータを用いることによりどこから逐次スキャンを開始しても問題ないので、Q1 と Q2 の逐次スキャンを 1 つに統合することが可能となる。

インデックススキャンは特定の値を持つタプルのみを読み取ることが可能なので必要となるタプルが少量であればインデックススキャンを用いた方がアクセスコストが低くなる場合がある。しかし、他のクエリによって同じテーブルに対して逐次スキャンが行われている場合にはインデックススキャンで読み取られるであろうタプルも逐次スキャンによって必ず読み取られる。そのため、インデックススキャンも逐次スキャンと統合することが可能となる。

4 シミュレーションによる潜在的有効性の確認

動的複数クエリ最適化を用いた際の性能を検証するべくシミュレーションを行った。また、逐次的なクエリ実行および既存の複数クエリ最適化を用いた場合の性能も同条件でシミュレーションし、潜在的な性能向上の確認を行った。

4.1 シミュレーションの設定

シミュレーションにはよく知られたワークロードである TPC-H ベンチマークを用い、Scale Factor は 1 に設定した。複数のクエリが同時に入力される状況として TPC-H の Throughput テストをシミュレーション向けに修正したものを用いた。同時に接続されているクライアント数は 5 であり、各クライアントは現在実行中のクエリの実行が完了したら次のクエリを入力する。各クライアントのクエリの実行系列は TPC-H の Throughput テストで規定されている実行系列を用いた。ただしシミュレーションの都合上 Q4 は除外した。本シミュレーションはディスクからの読み取りコストにのみ注目したシミュレーションであり、クエリ最適化や集約、ソート等の CPU インテンシブな処理やメモリ上でのデータ操作にかかった時間は無視している。

シミュレーション中の各操作の実行コストは PostgreSQL 上で Explain Analyze を指定して実行した結果を用いた。この時の実行環境は Intel Xeon Gold 6132 プロセッサ（14 コア、28 スレッド）を 2 ソケット備えており、DRAM は 93GB である。OS は Ubuntu20.03 であり、PostgreSQL のバージョンは 15devel である。またクエリの並列実行を無効化するために `max_parallel_workers_per_gather` を 0 に設定した。

シミュレーションした実行形式は以下の 3 種類である。

- Single: クエリを一つずつ逐次的に実行
- Multi: 複数クエリ最適化による複数クエリのバッチ実行
- Dynamic-Multi: 動的複数クエリ最適化による実行

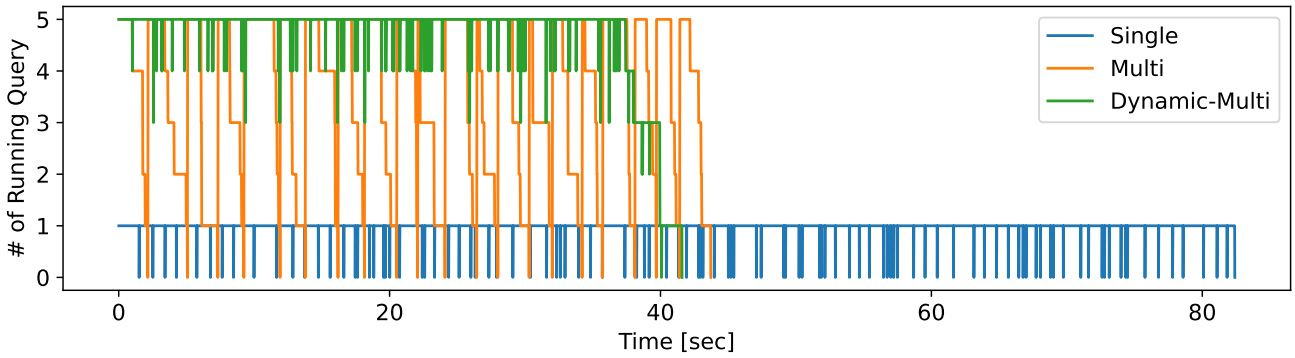

図 2: 各時刻におけるクエリの同時実行数

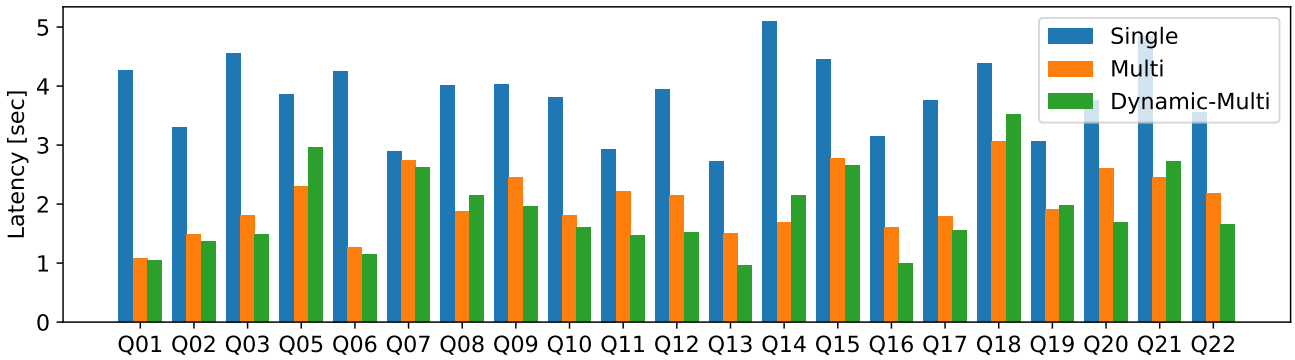

図 3: TPC-H (SF=1) の各クエリのレイテンシ

Single 及び Multi では DBMS はクエリバッファを持ち、現在実行中の処理が完了したらクエリバッファから次に実行するクエリを取り出して新たな処理を開始する。一方、Dynamic-Multi では入力されたクエリはバッファリングされることなく現在の処理に組み込まれる。

4.2 同時に実行されているクエリ数

シミュレーション中の各時刻に同時に実行されているクエリ数を図 2 に示した。逐次実行（Single）では同時に実行されているクエリ数はほぼ常に 1 であった。時折実行クエリ数が 0 になっているが、これはクエリが実行完了してから新たなクエリが入力されるまでの僅かな時間だけである。また、逐次実行において全てのクエリの実行が完了するまでにかかった時間は 82.4 秒であった。

複数クエリ最適化（Multi）では、最初は 5 つのクライアントから同時にクエリが入力されるため実行クエリ数は 5 であるが、実行が進むにつれて各クエリの実行が完了するため実行クエリ数が徐々に減少し、最終的に 0 になる。実行クエリ数が 0 になると、一つのバッチの実行が完了したことになるので新たなバッチの実行を開始する。新たなバッチ内には各クライアントから入力されたクエリが含まれるため、実行クエリ数は再び 5 になっている。以後、これを繰り返しながら実行が進行している。複数クエリ最適化において全てのクエリの実行が完了するのにかかった時間は 43.7 秒である。

動的複数クエリ最適化（Dynamic-Multi）では、ほぼ常に実行クエリ数が 5 で一定である。クエリが一つ実行完了すると即座に新たなクエリが入力されるため Multi と異なり実行クエリ数が 0 まで減少することはない。実行全体が終了に近づくとクライアントによっては全てのクエリの実行が完了し、同時に実行されているクエリ数が漸減し、最終的に 0 になる。動的複数クエリ最適化を用いるとこのように実行が進行していき、全てのクエリの実行が完了するのにかかった時間は 41.6 秒である。

4.3 各クエリのレイテンシ

各実行形式でのシミュレーションにおけるクエリの実行レイテンシを図 3 に示した。各クライアントは全てのクエリを 1 回ずつ実行するため、図 3 にはレイテンシの平均値を示した。逐次実行（Single）は全てのクエリにおいてレイテンシが最大である。これは逐次的なクエリ実行によりクライアントがクエリを入力しても他のクエリの実行完了を待機するため実際に実行が開始されるのはさらに後になるからである。

複数クエリ最適化（Multi）と動的複数クエリ最適化（Dynamic-Multi）を比較すると、例外はあるが動的複数クエリ最適化の方がレイテンシが比較的小さく、複数クエリ最適化に対して動的複数クエリ最適化によるスピードアップは最大で 1.60 倍、平均で 1.15 倍である。バッチ実行ではバッチの実行が開始されるまで待機する必要があるが、動的複数クエリ最適化ではこの待機が必要ない。また同時に実行されているクエ

リ数もほぼ常に最大値を達成しているので、共通化可能な処理が多くなり、実行が高速化されていると考えられる。

5 おわりに

本稿では既に実行が開始されているクエリ処理に対して新たに入力された未知クエリを考慮して実行時に複数クエリ最適化を行う動的複数クエリ最適化を提案した。また、動的複数クエリ最適化は既存の複数クエリ最適化よりもクエリのレイテンシを短縮することが可能であることを TPC-H ワークロードに対するシミュレーションにより示した。今後はシミュレーションにとどまらず、実際に動的複数クエリ最適化を可能とするデータベース管理システムを実装し、動的複数クエリ最適化の有効性を実験的に検証していきたいと考えている。

謝　辞

本研究の一部は，ビッグデータ価値協創プラットフォーム工学社会連携研究部門（東大、日立）の支援を受けたものである。

文　献

[1] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *Proc. SIGMOD'10*, pp. 519–530, 2010.

[2] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. SIGMOD'00*, pp. 261–272, 2000.

[3] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, Vol. 7, No. 6, pp. 429–440, 2014.

[4] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *Proc. SIGMOD'05*, pp. 383–394, 2005.

[5] Prashanth Menon, Amadou Ngom, Todd C. Mowry, Andrew Pavlo, and Lin Ma. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, Vol. 14, No. 2, pp. 101–113, 2020.

[6] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proc. ICDE'88*, pp. 311–319, 1988.

[7] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proc. SIGMOD'13*, pp. 1231–1242, 2013.

[8] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *Proc. ICDE'03*, pp. 353–364, 2003.

[9] Louiqa Raschid and Stanley Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *Proc. VLDB'86*, pp. 412–419, 1986.

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

[11] Panagiotis Sioulas and Anastasia Ailamaki. Scalable multi-query execution using reinforcement learning. In *Proc. SIGMOD'21*, pp. 1651–1663, 2021.

[12] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed Parallel Databases*, Vol. 1, No. 1, pp. 103–128, 1993.

[13] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In *Proc. ICDE'12*, pp. 1349–1350, 2012.