DEIM Forum 2018 C6-2

動的演算資源調整機構を有する 共有ストレージ型データベースエンジンの リソースモニタを用いた実行時挙動の解明

奥野　晃裕† 　早水　悠登† 　合田　和生† 　喜連川　優†,††

† 東京大学 生産技術研究所 〒 153-8503 東京都目黒区駒場 4-6-1
†† 国立情報学研究所 〒 101-8430 東京都千代田区一ツ橋 2-1-2
E-mail: {okuno,haya,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし　パブリッククラウド環境では，一般に資源の迅速な調達を可能とする資源伸縮性を提供している．著者らは，データベースエンジンにおいてパブリッククラウド環境の提供する演算資源の伸縮性を活用することを目的とし，動的演算資源調整機構を有する共有ストレージ型のデータベースエンジンの研究に取り組んでいる．当該機構は共有ストレージ型データベースエンジンにおいてクエリの実行時に演算資源を調整することを可能とするが，演算資源調整の実行時におけるデータベースエンジンの過渡的な挙動は十分に把握できておらず，動的演算資源調整手法の改良や応用に向けた研究を進めていくにあたって，精緻な観測を行い当該手法実行時の詳細な挙動を把握することが必要となる．本論文では動的演算資源調整実行時の精緻な観測を可能とするリソースモニタ機構を示し，パブリッククラウド環境における実験的評価によって，動的演算資源調整の実行時挙動を明らかにする．

キーワード　データベースエンジン，動的資源調整，クラウド，リソースモニタ

1. はじめに

自らハードウェアを所有し情報システムを構築するオンプレミス環境に加え，パブリッククラウド環境と呼ばれる新たな利用形態が広まってきている．パブリッククラウド環境は，Amazon や Google などの事業者が提供する演算資源やストレージなどをインターネットを介して利用する形態であり，2017 年時点で 25.7%のシェアを持ち，今後更に利用は拡大するものと予測されている [1]．パブリッククラウドの環境の有する特徴として資源の迅速な調達・破棄を可能とする資源の伸縮性が挙げられる．ユーザは資源伸縮性を利用することで，逐次変化するビジネス要求に対して必要な量の資源を確保することができる．

一方，関係データベースエンジンは，データの管理・処理を担う中心的なソフトウェアとして，今日の情報システムにおいて重要な役割を果たしている．しかしながら，従来の関係データベースエンジンは固定された量の資源を前提として設計されており，パブリッククラウド環境が提供する資源伸縮性を利用することは困難である．

著者らは関係データベースエンジンにおいて資源伸縮性を活用することを目的とし，共有ストレージ型データベースエンジンにおいて実行中のクエリに対して割当てる演算資源を調整する動的演算資源調整手法を提案してきた [2]．当該手法を適用することにより，実行中のクエリに対して追加の演算資源を割当てることによって当該クエリの実行を加速する，あるいは演算資源の全てがクエリの実行に割当てられている状況において高優先度のクエリの実行が要求された場合に，低優先度クエリの実行に割当てられた演算資源を高優先度クエリへと振り替えることにより，即座に高優先度クエリの実行を開始するなど，より柔軟に演算資源を調整しユーザの要求に即したクエリの実行が可能になる．

動的演算資源調整の実用にあたっては，動的演算資源調整の実行時にどのような挙動を示し，実行完了にどの程度の時間を要するかを把握することが重要となる．例えば，動的演算資源調整の実用の一つとして，あるクエリが全ての計算資源を利用して実行している状況においてより優先度の高いクエリの実行が要求された時に，実行中クエリへの演算資源の割当てを減らし当該演算資源をもって高優先度のクエリを即座に実行する，というものが挙げられる．しかしながら，演算資源の割当てに要する時間によっては，動的演算資源調整を行わずに実行中クエリの完了を待った方が早く高優先度クエリを開始できる，という状況も考えられる．著者らの提案する動的演算資源調整手法は，クエリの結果の正しく保ったままクエリの実行中に割当てる演算資源を調整することを調整することを可能とするが，当該手法に基づく演算資源調整を実行した際の各演算子における共有ストレージへの入出力量や演算子間のネットワーク送受信量の変動などの過渡的な挙動は十分に把握できていない．動的演算資源調整手法の改良を実施して，演算資源調整の実行完了までに要する時間を短縮し，当該手法に基づく応用の研究を進めていくためには，当該手法を実行した際の挙動を把握するための精緻な観測が必要となる．本論文では，共有ストレージ型データベースエンジンにおけるリソースモニタを示し，当該リソースモニタを用いて動的演算資源調整を実行中に詳細な観測を行うことにより，動的演算資源調整実行時の挙動を明らかにする．

本論文の構成は以下の通りである．2. 章では，動的演算資源調整を有する共有ストレージ型データベースエンジン，および

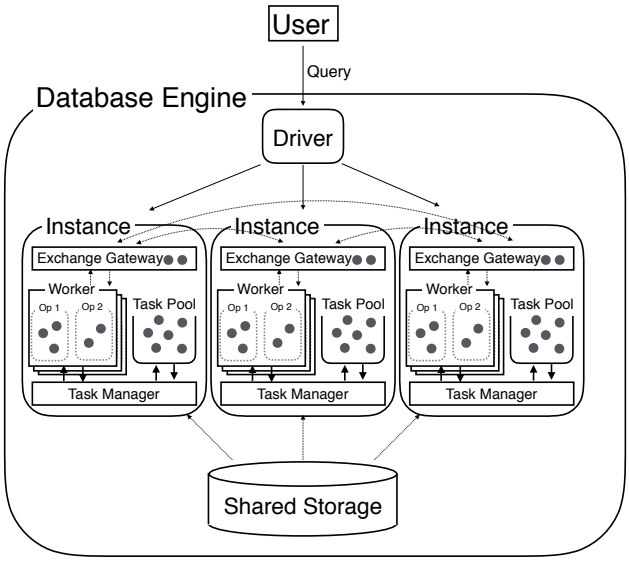


図 1: 動的演算資源調整を有する共有ストレージ型データベースエンジンの概要

Fig. 1 Overview of Database Engine with Dynamic Resource Adjustment Mechanism

当該データベースエンジンにおいて動的演算資源調整を行う手法について述べる．3. 章では，前章で述べた共有ストレージ型データベースにおけるリソースモニタについて述べる．4. 章ではパブリッククラウド環境における評価実験の結果を示す．5. 章では関連する研究について記し，6. 章では本論文のまとめと今後の展望について述べる．

2. 動的演算資源調整機構を有する共有ストレージ型データベースエンジン

2.1 共有ストレージ型データベースエンジンの概要

図 1 に動的演算資源調整機構を有する共有ストレージ型データベースエンジンの概要を示す．当該データベースエンジンは共有ストレージ型のアーキテクチャを採用しており，全ての演算資源（インスタンス）が一つのストレージを共有する．当該データベースエンジンでのクエリ実行は，ドライバなるプロセスがユーザからクエリを受け取ることで開始される．ドライバはクエリを受け取ると当該クエリに基づく実行計画を作成し，実行計画を複数のグループに分割する．本論文において当該グループを段と称し，ドライバは各段をインスタンスへと配布する．各インスタンスはドライバから実行計画の段を受け取ると，ワーカと称するプロセスを開始して，当該ワーカおいて受け取った実行計画の段に基づく演算を実行する．演算の実行において，入出力が必要な演算においてはワーカから共有ストレージへとネットワークを介した入出力が行われる．実行計画を複数の段に分割したため，ある演算の実行に異なる演算の結果を必要とする場合がある．その為，クエリの実行には段の間でのデータの送受信，即ち複数のワーカ間でのデータの送受信が必要となる．ワーカ間でのデータ送受信はエクスチェンジと称する機構を介して行われる．クエリの実行は複数のエクスチェンジを介した複数のワーカが連なるパイプライン構造となり，当該パイプラインにおいて，ワーカにおける各段の演算が並列に実行される．

2.2 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の手法

前節に示した共有ストレージ型データベースデータベースエンジンにおいては，クエリの実行中に新たなインスタンスにおいてワーカを起動する，あるいはワーカ起動中のインスタンスにおいて当該ワーカを停止することによって，クエリの実行中に割当てるインスタンスを追加・削除する，即ちクエリ実行時に動的に演算資源の調整を行うことが可能である．しかしながら，クエリ実行中の各ワーカにはエクスチェンジを介して他のワーカから随時データが送られ，当該受信データの処理が行われ続けているため，ワーカにおけるデータの処理状況を考慮せずに単にワーカの起動・停止を行うとデータが正しく処理されず，クエリの正しい結果が得られない可能性がある．その為，動的演算資源調整手法においては，クエリの実行中に全てのデータを正しく処理しつつワーカの起動・停止を行う手法が必要となる．

まず，図 2 にワーカ起動の手法を示す．ワーカ起動においては，新規ワーカの準備が完了する前に，前段のワーカからデータが送信されると当該データが正しく処理されず失われてしまう．その為，まず新規ワーカにおいて送信されたデータをデータを処理し後段のワーカへと送信する準備が整ってから，前段のワーカから新規ワーカへの接続を確立しデータの送信を開始する．

次に，図 3 にワーカ停止の手法を示す．ワーカ停止処理では停止ワーカに送られてきた全てのデータを正しく処理する必要がある．その為，まず前段のワーカから停止ワーカへのデータ送信を停止し，停止ワーカが受けとるデータを確定させる．その後に，停止ワーカにおいて前段のワーカからのデータ受信の完了を待ち，受け取ったデータを全て処理して後段のワーカへの送信を完了した後に，当該ワーカを停止する．この手順に従うことで，停止を行うワーカにおいて受け取ったデータが全て正しく処理されることを保証する．

以上のワーカ起動・停止の手法を用いることで，共有ストレージ型データベースエンジンにおいて，全てのデータを正しく処理しつつワーカの起動・停止を行い，クエリ実行に割当てるインスタンスを追加・削除することが可能となる．

3. 共有ストレージ型データベースにおけるリソースモニタ

2. 章で示した手法によって，クエリの実行時に割当てる演算資源の調整を行うことが可能になるが，当該手法の実行時に実行計画の各演算子における入出力処理やエクスチェンジを介した送受信がどのような過渡的挙動を示すかは十分に把握できていない．本論文で提案する共有ストレージ型データベースエンジンでは，各インスタンスにおいて実行された複数のワーカがエクスチェンジを介したパイプラインを構成しており，当該ワーカにおける演算では共有ストレージ型データベースへの入出力が非同期に実行される．そこで，各インスタンスにおいて

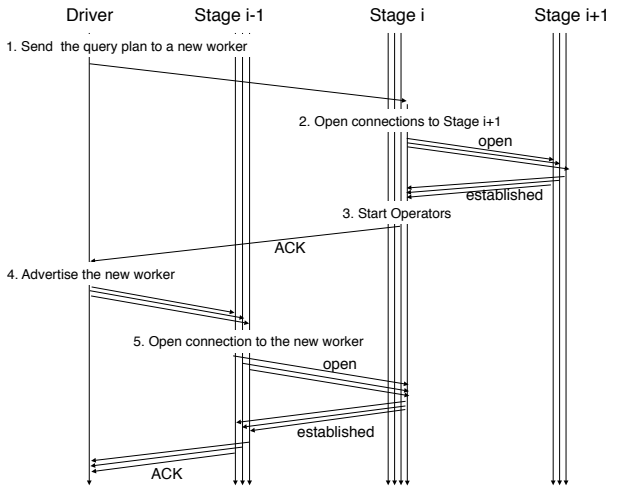


図 2: i 段目のワーカ追加の手続き
Fig. 2 Method of adding a worker at stage i

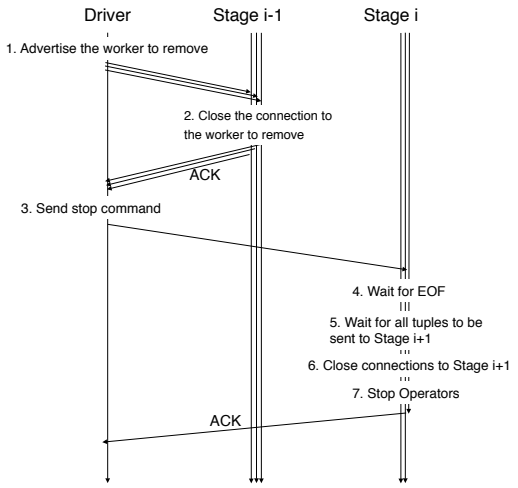


図 3: i 段目のワーカ削除の手続き
Fig. 3 Method of remove a worker at stage i

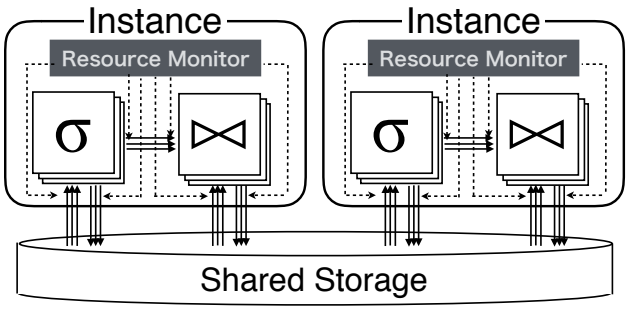


図 4: 共有ストレージ型データベースにおけるリソースモニタ
Fig. 4 Resource monitor for the shared storage database engine

当該インスタンスで実行された演算の挙動を観測するリソースモニタを新たに設ける．4 は 2 表の結合演算からなるクエリ実行におけるリソースモニタの観測例を示す．リソースモニタは，各演算子における入出力の発行・完了の回数，およびエクスチェンジでの送信・受信数を計測しており，各計測数を一定のタイミングで記録する．当該リソースモニタを用いてクエリ実行時の入出力数，エクスチェンジの送受信数を計測することにより，インスタンスの追加・削除操作の実行時に，ある時点において手続きがどこまで進んでいるかを把握し，手続きのどこがインスタンス追加・削除操作を律速しているかを観察することが可能になる．また，入出力の発行と完了の両方を計測することで，共有ストレージの入出力における遅延の影響を観測することを可能とする．

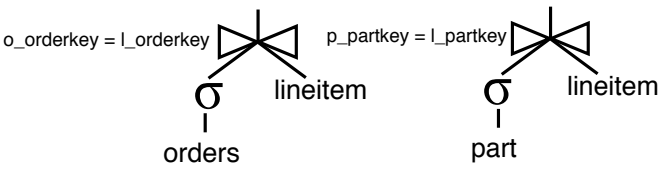
4. パブリッククラウド環境における評価実験

4.1 動的演算資源調整機構を有する共有ストレージ型データベースエンジンの試作

著者らは 2. に示した共有ストレージ型データベースエンジンの試作実装を進めてきた．当該試作では，2. で示したインスタンス追加・削除の操作を実装しており，当該操作のインターフェースとして，ドライバプロセスにインスタンス追加・削除のコマンドを設けている．実行中のクエリに対して当該コマンドを実行することで，新規インスタンスでのワーカの起動，またはワーカ起動中のインスタンスでのワーカ停止が実行され，当該クエリの実行へのインスタンスの割当てが変更される．さらに，3. に示したリソースモニタを当該試作において実装し，クエリの実行時においてデータベースエンジン内部の各演算子毎の入出力数の観測を可能とした．

表 1: AWS（N. Virginia region）実験環境諸元
Table 1 Specifications of AWS (N. Virginia region) environments

EC2 c4.8xlarge	**64 Instances**
CPU	36 vCPU
Memory	60 GiB
OS	Amazon Linux 64bit (hvm)
Hardware	Shared (Default Tenancy)
Network Bandwidth	10Gbps
Network Location	Colocated (Placement Group)



(a) クエリ 1 (b) クエリ 2

図 5: 評価実験に用いたクエリのプランツリー
Fig. 5 Plan trees of queries for evaluation experiments

4.2 実験環境

前節で示した試作を用いて，パブリッククラウド環境における評価実験を行った．パブリッククラウド環境としては Amazon Web Service（AWS) を用い，計算資源として EC2 を，共有ストレージとして DynamoDB をそれぞれ用いた．表 1 に実験環境の諸元を示す．

データセットとしては，TPC-H ベンチマーク [3] の dbgen を用い，$ScaleFactor = 1000$ で生成したデータを DynamoDB のテーブルとしてロードした．また，各テーブルには TPC-H の仕様に定められた二次索引を作成した．

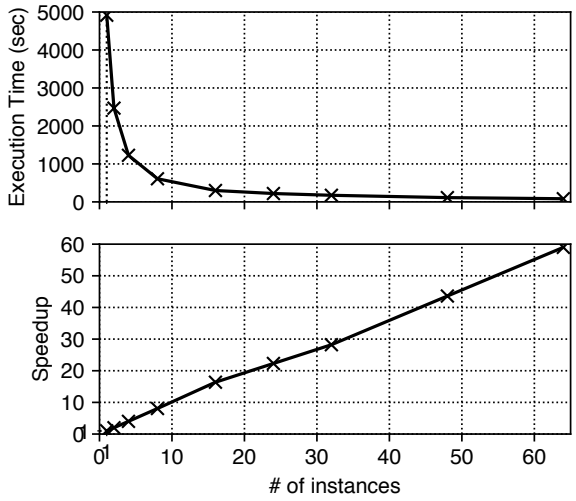


図 6: クエリ Q1 の実行時間および性能向上率
Fig. 6 Execution time and speedup of Q1

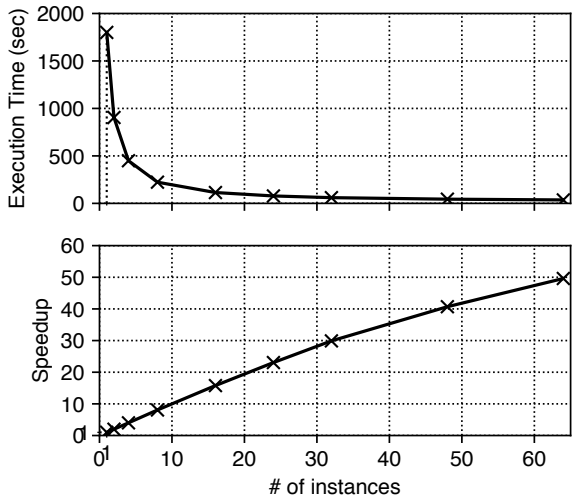


図 7: クエリ Q2 の実行時間および性能向上率
Fig. 7 Execution time and speedup of Q2

実験には図 5 に示す orderes，lineitem の結合演算を行うクエリ Q1 と，part，lineitem の結合演算を行うクエリ Q2 の 2 つのクエリを用いた．orders の o_orderkey と，part テーブルの p_partkey に対して選択演算を行い，選択率は 1%となるように調整した．また，結合演算の方式は二次索引による Index Join を用いた．

実験に用いたクエリは，orders または part テーブルのスキャンを行う段と，lineitem テーブルとの結合を行う段からなる 2 段の実行計画となるが，クエリの実行時には各インスタンスにおいて各段に対するワーカを実行し，各インスタンスの実行内容が等しくなるようにした．インスタンスの追加・削除にあたっては，当該インスタンスでの全てのワーカを起動・停止するようにした．

4.3 演算資源に対するスケーラビリティ

本実験では，演算資源に応じてクエリの実行速度の向上が得られることを示すために，クエリ実行に割当てたインスタンス数に対する実行時間を計測した．クエリの実行時間と，インスタンス数 1 の時を 1 とした性能向上率を図 6，図 7 に示す．Q1，Q2 ともにインスタンス数に応じて性能向上が得られ，インスタンス数 64 の時にインスタンス数 1 の時と比べて，Q1 では 59.9 倍，Q2 では 49.6 倍の性能向上が得られた．また，Q1 は Q2 に比べ 64 インスタンスの時に 2.73 倍の実行時間を要した．これは本実験の設定においては，Q2 では part のスキャン速度が lineitem の取得処理速度を上回るため，lineitem の取得処理が全体の処理を律速するのに対し，Q1 では orders のスキャン処理速度が lineitem の取得処理速度が下回り，orders のスキャン処理によって全体の処理が律速されたためである．以上の結果から，本論文で示したデータベースエンジンでは，利用するインスタンス数に応じた性能向上が得られることを明らかにした．

4.4 実行中クエリに対する動的演算資源調整

実行中クエリに割当てる演算資源を提案手法によって調整可能であることを示すために，Q1，Q2 の実行中に割当てるインスタンス数の追加・削除の操作を行い，割当てられたインスタンス数，およびクエリ全体でのストレージへの毎秒リクエスト回数（IOPS）を 1 秒ごとに計測した．

Q1 の実行においては，まずインスタンス数 8 でクエリの実行を開始し，30 秒ごとに割当てるインスタンス数が 32,64 になるようにインスタンスの追加を行った．その後，30 秒毎にインスタンス数が 32,8 になるようにインスタンスの削除を行った．

Q2 の実行においては，まずインスタンス数 8 でクエリの実行を開始し，10 秒ごとに割当てるインスタンス数が 32,64 になるようにインスタンスの追加を行った．その後，10 秒毎にインスタンス数が 32,8 になるようにインスタンスの削除を行った．

Q1 の測定結果を図 8 に，Q2 の測定結果を図 9 に示す．Q1，Q2 ともにインスタンスの追加・削除の操作によって，クエリ実行における IOPS を増加・減少させることができ，Q1 では最大 1.28M，Q2 では最大 2.66M の IOPS が得られた．また，IOPS が増加し定常状態に達した時の IOPS について，8 インスタンスでの IOPS を 1 とした時の性能向上率を表 2 に示す．Q1，Q2 ともにインスタンス数に応じた性能向上が得られた．

次にインスタンスの追加・削除操作を実行した際に，インスタンススの変化が IOPS の増加・減少に反映されるまで要した遅延に着目する．インスタンスの追加については，Q1，Q2 ともにインスタンスの追加操作から IOPS が増加し定常状態に達するまでに 4 秒程度の遅延を要した．一方，インスタンスの削除について，Q1 ではインスタンス削除操作から 2 秒程度で IOPS が減少したのに対し，Q2 では 5 秒程度の遅延を要した．本実験の結果から，提案手法によってクエリの実行中に割当てる計算資源を調整し，IOPS を増加・減少させることを可能とするが，割当てた計算資源が IOPS へと反映されるまでに要する時間はインスタンスと追加と削除によって異なること，またクエリに種類によっても IOPS への反映までに要する遅延が異なることを示した．

4.5 インスタンス追加時の実行時挙動

2. 章で示した手法では，インスタンスの追加時には，まず新

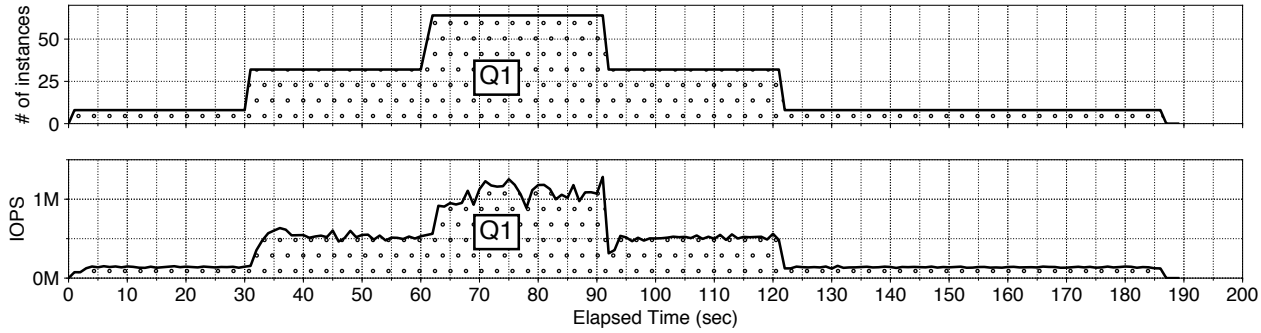


図 8: クエリ Q1 に対する動的演算資源調整

Fig. 8 Dynamic resource adjustment for Q1

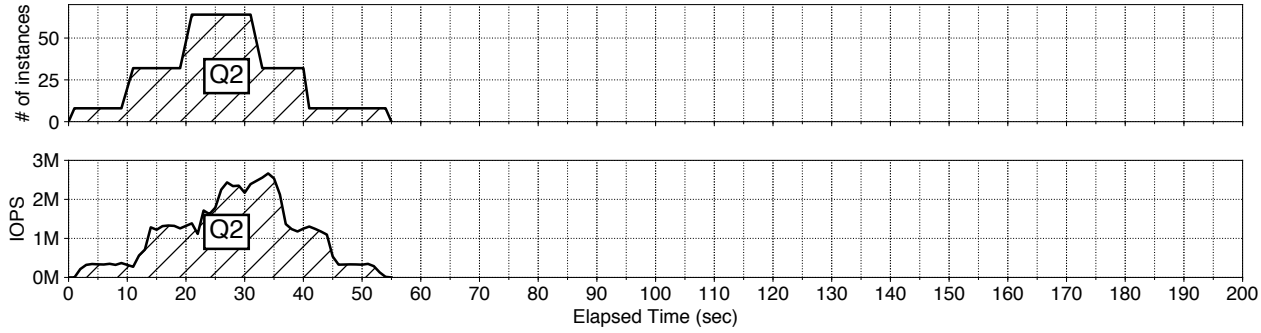


図 9: クエリ Q2 に対する動的演算資源調整

Fig. 9 Dynamic resource adjustment for Q2

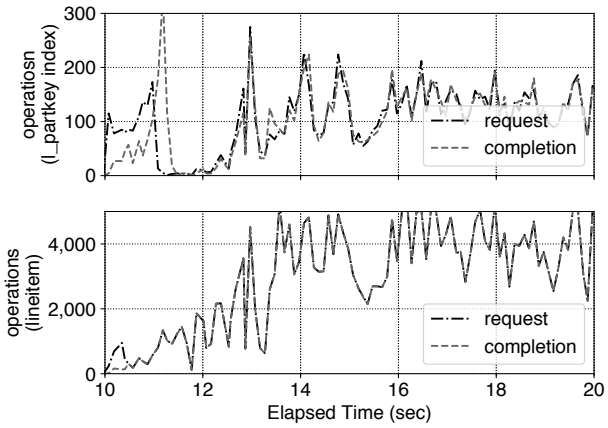


図 10: クエリ Q2 のインスタンス追加時の挙動

Fig. 10 Behavior during adding instances of Q2

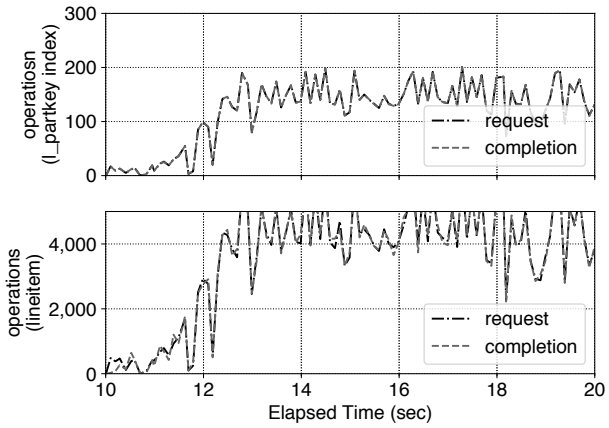


図 11: タスクの同時実行数を制限した時のクエリ Q2 のインスタンス削除時の挙動

Fig. 11 Behavior during adding instances of Q2 with limiting the number of concurrent tasks

表 2: 動的演算資源調整を行った時の IOPS 性能向上率

Table 2 Speedup while adjusting computing resources

インスタンス数	Q1 性能向上率	Q2 性能向上率
8	1	1
32	3.71	3.83
64	7.75	6.77

規ワーカから後段ワーカへの接続を確立し，新規ワーカの準備が整った後に，前段ワーカから新規ワーカへのデータ送信が開始され処理が行われる．前節の実験ではインスタンス追加操作が IOPS の増加へ反映されるまでの 4 秒程度の遅延を要した．当該遅延の原因を明らかにするために，前節の実験において，Q2 について 8 インスタンスから 32 インスタンスへとインスタンス追加操作を行った 10 秒時点からの，ワーカ内の各演算における共有ストレージへの入出力の発行・完了の回数を，3. 章で示したリソースモニタによって 0.1 秒ごとに計測した．

追加したインスタンスのうちの 1 インスタンスでの測定結果を図 10 に示す．図 10 は，上から l_partkey インデックスへの入出力発行及び完了回数，lineitem テーブルへの入出力発行及

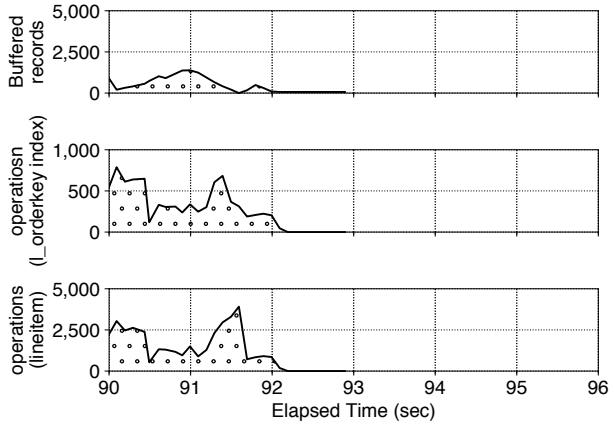


図 12: クエリ Q1 のインスタンス削除時の挙動
Fig. 12 Behavior during deleting instances of Q1

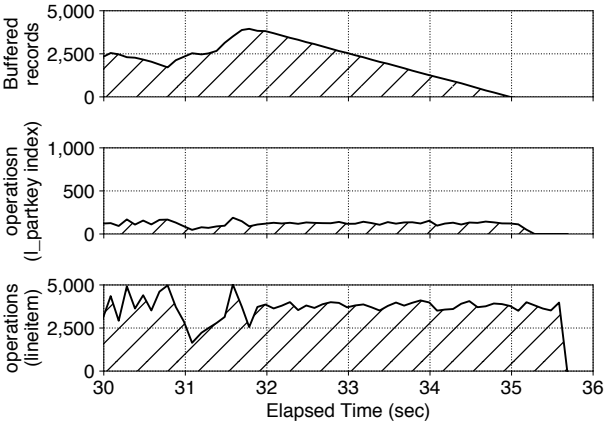


図 13: クエリ Q2 のインスタンス削除時の挙動
Fig. 13 Behavior during deleting instances of Q2

び完了回数を表す．l_partkey インデックスへの入出力は，インスタンス追加操作開始直後に大量に入出力リクエストを発行し，1 秒程度でそれらの入出力が完了，凡そ 2 秒程度入出力の発行・完了ともに低下した後に，凡そ 4 秒時点までまで入出力数が上昇し，16 秒時点から定常状態に達した．lineitem テーブルへの入出力は l_partkey インデックスへの入出力数が上昇する 4 秒時点まで緩やかに上昇した後に定常状態に達した．Q2 全体の IOPS は lineitem テーブルへの IOPS が大半を占めているため，Q2 におけるインスタンス追加操作から IOPS の増加への遅延は，lineitem テーブルへの入出力数が安定するまでの時間に起因し，当該時間は l_partkey インデックスへの IOPS が安定するまでの時間によって決まると推測される．提案手法では，インスタンスの追加操作の開始時に前段ワーカからデータが送信され，当該データによる大量のタスクが作成されて，一時的に定常状態の IOPS を超える入出力が発行される可能性がある．当該入出力数は系の定常状態を超えるものであるため，当該入出力を基に各演算子の入出力数が発振し，当該発振によって入出力数が安定するまでに遅延が発生した推察される．

また，DynamoDB のような共有ストレージでは一度に大量の入出力が発行されると，入出力完了までの遅延が増加することがある．本実験においてはインスタンス追加操作の開始直後に l_partkey インデックスへ大量の入出力が発行され遅延が増加した．本論文で前提としたクラウド上の共有ストレージでは，当該遅延の発生を考慮する必要があり，当該遅延の影響を低減する手法は今後の課題としたい．

発振による遅延を低減する方法はいくつか考えられるが，l_partkey インデックスへの入出力を行うタスクの同時実行数を明示的に制限する方法が挙げられる．l_partkey インデックスへの入出力タスクの同時実行数を 32 に制限した計測結果を図 11 に示す．定常状態に達した後の IOPS は変わらない一方で，l_partkey インデックスへの入出力が操作開始直後からなだらかに上昇し，2 秒程度で定常状態に達した．同時実行数を制限しない場合と比べ，IOPS が定常状態に達するまで 2 秒程度短縮することができた．また，入出力の上昇がなだらかであった為，同時実行数を制限しない場合に発生した，l_partkey インデックスへの入出力の遅延の増加も回避することができた．

以上の結果から，インスタンスの追加操作時には多数の入出力が同時に発行されて IOPS が発振することによって，操作開始から IOPS の上昇への遅延が発生することを明らかにした．また，IOPS の発振に起因する遅延は入出力タスクの同時実行数を制限することで回避可能であることを示した．しかしながら，あるクエリにおける入出力タスクの最適な同時実行数は自明ではない．本実験に用いた Q2 は lineitem テーブルへの入出力数に対して l_partkey インデックスへの入出力が十分に小さいため，IOPS を保ったまま同時実行タスク数を制限することが容易であったが，クエリの制約の条件や結合の条件，あるいはインスタンス毎の負荷の偏りなどで，最適な同時実行数は容易に変化することが推測される．クエリの実行中に各演算子における実行状況をモニタし，タスクの同時実行数を動的に調整する仕組みが必要になると考えられ，今後の課題としたい．

4.6 インスタンス削除時の実行時挙動

インスタンス削除時には，まず前段ワーカから停止ワーカへのデータ送信を止め，その後に，停止ワーカで全てのデータを受信し，当該データを処理して当該処理の結果を後段ワーカに送信してからワーカを停止する．第 4.4 節の実験において，インスタンス削除操作を実行してから IOPS に反映されるまでに Q1 では 2 秒程度，Q2 では 5 秒程度の遅延を要した．当該遅延の原因を明らかにするべく，第 4.4 節の実験において，64 インスタンスから 32 インスタンスへとインスタンス削除を行った時点，即ち，Q1 では 90 秒時点，Q2 では 30 秒時点からそれぞれ 6 秒間について，リソースモニタを用いて用いて計測を行った．

削除をおこなった 1 インスタンスでの結果を図 12，13 に示す．図 12，13 は，上から，orders テーブルあるいは part テーブルでスキャンされてから lineitem との結合処理待ちのレコード数，l_orderkey インデックスあるいは l_partkey インデックスへの入出力数，lineitem テーブルへの入出力数を表している．liteitem との結合処理待ちのレコードは演算子内のキューやエクスチェンジにおけるソケットバッファ等に存在している．インスタンス削除処理の実行時にはこのバッファされたレコード

を全て受信するまで待つ必要がある．

Q1 では orders のスキャン処理が律速段階となるため，バッファされたレコード数は少なく，かつ当該レコードに対して速やかに結合処理が進み，インスタンス削除処理から約 2 秒後には l_orderkey インデックス，lineitem テーブルに対する入出力処理が完了した．一方，Q2 では lineitem の取得処理が律速段階となるため，多くのレコードが lineitem の結合処理待ちとしてバッファされていた．また orders テーブルの 1 レコードに対して lineitem テーブルの平均 4 レコードが結合されるのに対し，part テーブルについては 30 レコードが結合されるため，結合処理待ちとしてバッファされたレコードに対する結合処理の速度が遅く，全てのレコードが処理されて，入出力処理が完了するまでに 5 秒を要した．これらの結果から，インスタンス削除操作の実行時には，どの演算子が律速段階となっているか，また結合処理においてはテーブル間の結合されるレコード数の比率が，インスタンス削除操作から IOPS への応答性に影響することを明らかにした．

5. 関 連 研 究

クラウド環境における資源の伸縮性を利用するデータベースエンジンは，パブリッククラウド事業者が AWS Aurora [4] や BigQuery [5] などを商用サービスとして提供しているほか，学術界においても，ワークロードやユーザのサービス要求に応じてクエリへの演算資源の割当てを調整する手法 [6–8] などが提案されている．また，クラウド環境においては演算資源を利用した時間に応じて必要なコストが変わることに着目し，ユーザに対してクエリに用いる演算資源に直接課金をさせるのではなく，クエリ毎の予測実行時間を提示し当該実行時間に応じた課金を行う手法 [9] や，クエリの実行中に演算資源の故障するなどでクエリの完全な結果が得られなくなった場合に，不完全な結果を許容することによってクエリの再実行を回避し演算資源の利用量を削減してコストを抑える選択肢をユーザに提供する手法 [10] などが提案されている．これらの手法は，クラウド環境の提供する資源伸縮性を活用し，クエリ実行に割当てる演算資源を調整するが，従来では演算資源の調整の機会はクエリの実行前に限られていた．著者らが提案する動的演算資源調整の手法を用いることで，クエリの実行中においても割当てる演算資源を変更することが可能となる．

利用する演算資源を処理の実行中に調整し負荷分散を行う手法として，MapReduce の研究 [11] や，SAN による共有ディスククラスタを利用した研究 [12–14] が挙げられる．本論文では関係データベースエンジンにおける動的な演算資源の調整を目的としており，これらの研究とは対象を異にする．Dynamo [15] や BigTable [16]，Cassandra [17] などの分散キーバリューストアの多くは，起動したまま負荷に応じて利用する演算資源を調整する機能を提供している．しかしながら，分散キーバリューストアでは限られた処理のみが可能であり，関係データベースエンジンが提供する複雑なクエリ処理を実行することはできない．

本論文ではキーバリューストアである DynamoDB を共有ストレージとして用いたが，同様にキーバリューストアやオブジェクトストレージを共有ストレージとして用いる共有ストレージ型データベースエンジンの研究が盛んに行われており，[18–22]，近年においてもキーバリューストアをを共有ストレージとして用いた共有ストレージ型データベースエンジンにおいて，分析的なクエリを高速に実行する手法 [23] などが提案されている．本論文では，共有ストレージ型データベースエンジンにおいて，クエリ実行時に割当てる演算資源を調整することでクラウド環境の提供する資源伸縮性の活用を目的としている．

データベースエンジンのアーキテクチャとしては共有ストレージ型の他に無共有型も広く用いられている．無共有型では共有ストレージと異なりデータを分割して各演算資源に分散して配置するため，クエリの実行効率がデータの配置に影響を受けやすい，ワークロードに応じて各演算資源内のみでクエリ処理が完了するようにデータの配置を調整する手法 [24–26] が提案されている．著者らが提案する動的演算資源調整手法は共有ストレージ型を前提としているが，無共有型においても演算資源間のデータ移動などを行うことで適用可能となると考えられ，今後の課題としたい．

6. お わ り に

本論文では，共有ストレージ型データベースエンジンにおいて動的演算資源調整を実行した際の挙動を明らかにするために，パブリッククラウド環境において試作実装を用いた評価実験の結果を示し，動的資源調整調整の実行時挙動について論じた．インスタンス数に対するスケーラビリティの評価実験では，64 インスタンスで 59.9 倍，49.6 倍の性能向上が得られた．また，提案手法によって実行中クエリに対して割当てる演算資源を調整し，それによってクエリ実行における IOPS を調整可能であることを示した．インスタンス追加時には 4 秒程度，インスタンス削除時には 2 秒，あるいは 5 秒程度の遅延を要した．インスタンス追加操作の実行時には，操作開始時に大量の入出力タスクが実行されることで，入出力数の発振が起こり，当該発振によってクエリ全体の IOPS が上昇するまでに遅延が発生した．当該遅延は入出力タスクの同時実行数を制限することで回避可能であることを明らかにした．インスタンス削除操作実行時の遅延には，律速段階となる演算子や，結合処理時のテーブル間のレコード数の比率が影響することを明らかにした．今後は，1 リクエスト単位のトレースによってより詳細な実行時挙動の把握に取り組み，本論文で明らかにした IOPS が定常状態に達するまでの遅延を低減に取り組むとともに，提案手法を用いて複数クエリ実行における負荷分散の研究にも取り組んでゆきたい．

謝辞 本研究の一部は，内閣府革新的研究開発推進プログラム（ImPACT）「社会リスクを低減する超ビッグデータプラットフォーム」の下に実施したものである．また，本研究の一部を進めるに当たっては，Amazon Web Services, Inc より AWS Cloud Credits for Research Program による実験環境資源の提供を受けた．感謝する次第である．

文　　献

[1] Spending on IT Infrastructure for Public Cloud Deployments Will Return to Double-Digit Growth in 2017, According to IDC.

[2] 奥野晃裕, 早水悠登, 合田和生, 喜連川優. クラウド環境に於けるクエリ実行時の資源調整機構を備えた高速データベースエンジンの試作に関する一考察. 信学技報, DE2017-25, Vol. 117, No. 374, pp. 7–12, Dec 2017.

[3] The TPC-H Benchmark.

[4] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD Conference*, pp. 1041–1052, 2017.

[5] Sérgio Fernandes and Jorge Bernardino. What is BigQuery? *IDEAS*, pp. 202–203, 2015.

[6] Ryan Marcus and Olga Papaemmanouil. WiSeDB - A Learning-based Workload Management Advisor for Cloud Databases. *PVLDB*, 2016.

[7] Dimokritos Stamatakis and Olga Papaemmanouil. SLA-driven workload management for cloud databases. *ICDE Workshops*, pp. 178–181, 2014.

[8] Liangzhe Li and Le Gruenwald. An SLA and Operation Cost Aware Performance Re-tuning Algorithm for Cloud Databases. *CLOUD*, pp. 966–969, 2016.

[9] Yue Wang, Alexandra Meliou, and Gerome Miklau. Lifting the Haze off the Cloud - A Consumer-Centric Market for Database Computation in the Cloud. *PVLDB*, 2016.

[10] Willis Lang, Rimma V Nehme, and Ian Rae. Database Optimization in the Cloud - Where Costs, Partial Results, and Consumer Choice Meet. *CIDR*, 2015.

[11] Sven Groot, Kazuo Goda, and Masaru Kitsuregawa. Towards improved load balancing for data intensive distributed computing. *Proceedings of the 26th ACM Symposium on Applied Computing, Technical Track on Cloud Computing (SAC2011)*, pp. 27–32, May 2011.

[12] 合田和生, 田村孝之, 小口正人, 喜連川優. San 結合 pc クラスタにおけるストレージ仮想化機構を用いた動的負荷分散並びに動的資源調整の提案とその評価. 電子情報通信学会論文誌. D-I, 情報・システム, I-情報処理 = The transactions of the Institute of Electronics, Information and Communication Engineers. D-I, Vol. 87, No. 6, pp. 661–674, jun 2004.

[13] Kazuo Goda, Takayuki Tamura, Masato Oguchi, and Masaru Kitsuregawa. Run-time load balancing system on san-connected PC cluster for dynamic injection of CPU and disk resource - A case study of data mining application. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, pp. 182–192, 2002.

[14] Masato Oguchi and Masaru Kitsuregawa. Runtime data declustering over san-connected PC cluster system. In Rakesh Agrawal and Klaus R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, p. 275. IEEE Computer Society, 2002.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pp. 205–220, 2007.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pp. 205–218. USENIX Association, 2006.

[17] Apache Cassandra. http://cassandra.apache.org/.

[18] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1: the fault-tolerant distributed RDBMS supporting google's ad business. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pp. 777–778. ACM, 2012.

[19] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, Vol. 38, No. 1, pp. 5:1–5:45, 2013.

[20] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pp. 251–264. ACM, 2008.

[21] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pp. 123–133. www.cidrdb.org, 2011.

[22] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the design and scalability of distributed shared-data databases. In Sellis, et al. [27], pp. 663–676.

[23] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *PVLDB*, Vol. 10, No. 11, pp. 1526–1537, 2017.

[24] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, Vol. 3, No. 1, pp. 48–57, 2010.

[25] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. *EDBT*, pp. 430–441, 2013.

[26] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In Sellis, et al. [27], pp. 17–30.

[27] Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 2015.