博士論文

データベースエンジンにおける クラウド資源の高次利用手法に関する研究

（A Study on Sophisticated Utilization of Cloud Resources for Database Engines）

奥野 晃裕

目次

図目次

表目次

第 1 章

序論

1.1 はじめに

1.1.1 研究背景

IT システムを構成する環境としてクラウドと呼ばれるサービスの利用が広まっている．クラウドには単一の定義は存在しないが，アメリカ国立標準技術研究所の報告書におけるクラウドコンピューティングの定義は数多く引用されており，当該報告書においてクラウドコンピューティングは次のように定義されている．「構成変更可能な計算資源（ネットワーク，サーバ，ストレージ，アプリケーション，サービスなど）からなる共有プールに対して，要求に応じて容易にネットワークを介したアクセスを可能とするモデルであり，最小限の管理努力，最小限のサービス提供者との交信によって迅速に資源の用意，解放が可能である」（[1]，p.2，訳は引用者による）．クラウドは提供する機能の形態によって，Software as a Service(SaaS)，Platform as a Service(PaaS)，Infrastructure as a Service(IaaS) などに分類される．本論文では仮想化されたサーバ資源であるインスタンスをネットワークを介してサービスとして提供する形態である IaaS を主に対象とし，以下クラウドと述べた時は IaaS を指す．計算機環境としてのクラウドは，サーバを自ら保有・管理する従来のオンプレミス環境と異なり，全ての資源は仮想化されたものであり，一つのシステムを複数のユーザが共有して利用するマルチテナント方式が主流となっている．クラウド環境においてユーザはオンデマンドに迅速な資源の調達・破棄が可能であり，クラウド環境が提供する新たな機会としてユーザに広く受け入れられている．その一方で，クラウド環境における資源の利用にあたっては，資源が仮想化されていることによりオンプレミスと比べて均質な資源を確保することが困難である，またビジネスから生

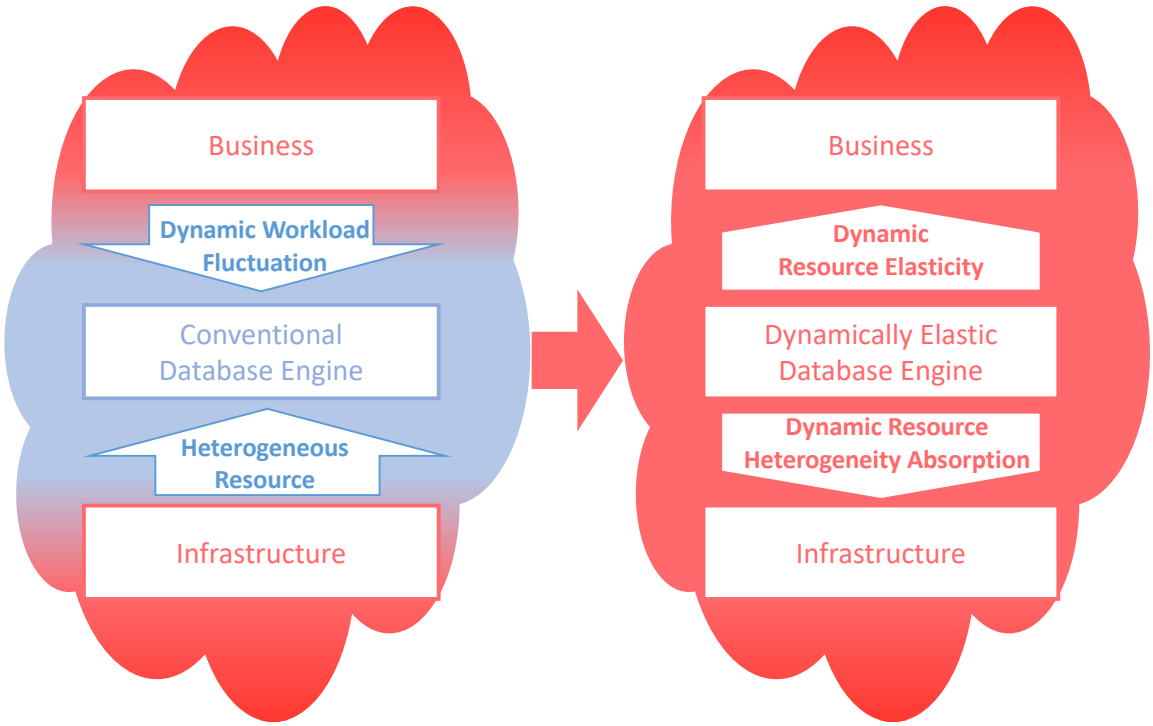


図 1.1: データベースエンジンにおける動的資源伸縮と動的資源非均質性吸収によるクラウド環境への適合

じるワークロードの変動性が従来より動的になってきているといった，システムに対する新たな課題が生じる．従来のソフトウェアの多くはオンプレミス環境において実行されることを前提に設計されているが，クラウド環境における当該課題を踏まえた設計を行うことによって，よりクラウド環境に適合した資源の利用が可能になると考えられる．

一方，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 [2]，ビッグデータとも呼ばれる当該大規模データの解析による利活用が競争優位性の源泉となるとされている．高速なビッグデータ解析のためには多くの資源が必要となるが，価格性能比の観点から，単体あるいは少数の高価で高性能なサーバを用いるのではなく，多数のコモディティサーバをネットワークで接続するクラスタシステムが用いられることが多い．当該クラスタシステムにおいて，クラウド環境はシステム運用開始前の予測需要のみではなく運用開始後の実需要に基づいたインスタンスの構成やインスタンス数の迅速な変更が可能になるなどの利点があり，ビッグデータ解析システムにおけるクラウド環境の利用が広まっている．並列データベースエンジンは複数のサーバを用いてクエリ処理を並列化することによってクエリ実行を高速化するデータベースエンジンであり，ビッグデータ

解析に利用される主なクラスタシステムの一つである，従来の並列データベースエンジンの多くはオンプレミス環境を前提に設計されており，クラウド環境において十分に適合しているとは言えない．ビッグデータ解析におけるクラウド環境の資源を利用するにあたって，クラウド環境により適合した並列データベースエンジンの設計が重要になるものと考えられる．

本論文では，クラウド環境における課題である，ワークロードの動的変動性ならびに資源の非均質性という 2 つの課題に着目する．本論文では，並列データベースエンジンの代表的なアーキテクチャの一つである共有ストレージ型並列データベースエンジンについて，ワークロードの動的変動性に対してはクエリ実行時の動的資源伸縮を実現する資源調整手法を，ならびに資源の非均質性に対しては動的資源非均質性吸収を実現する負荷分散手法を提案することにより，クラウド環境に適合しクラウド資源の高次利用を可能とする並列データベースエンジンの実現を目指す（図 1.1）．

1.1.2 動的資源伸縮を実現する資源調整機構

オンプレミス環境においては，一般にサーバの調達開始から実際にサーバが利用可能になるまでは数週間から数ヶ月の時間が必要であり，サーバの破棄についても即座に行うことはできないが，クラウド環境では，ユーザは仮想化されたサーバであるインスタンスをオンデマンドに調達・破棄することが可能であり，当該調達・破棄の実行は一般に数十秒から数分程度で完了する．また，オンプレミス環境では，多くの場合サーバは調達時に購入されるため，サーバを破棄した場合に削減可能なものは当該サーバの保守・管理費用のみである．一方，多くのクラウドではインスタンスを実際に利用した時間に応じて金額が決まる従量課金性が用いられているため，インスタンスの破棄を行った場合には，破棄したインスタンスの量に応じた利用コストの削減が可能となる．

サーバ調達におけるこれらの特徴から，あるシステムにおける資源需要予測と実際の資源需要，ならびに当該需要予測に基づいたオンプレミス環境，クラウド環境に各々環境におけるサーバ調達による利用可能資源量の変動の概念図を図 1.2 に示す．図 1.2 では簡単のため資源需要予測は線形に増加するものとする．

オンプレミス環境においては，システムの資源需要の予測に基づいて，事前にサーバ調達を開始することによって，利用可能な資源量が，予測需要を上回らないようにする．図 1.2 では事前に開始したサーバ調達が t_1 の時点で完了し利用可能な資源量が増加している．しかし，システムの資源需要予測が実際の需要を上回る，あるいは需要の変動サイクルにおいて需要の低い時間帯であるなどの理由により，利用可能資源量が資源需要を上

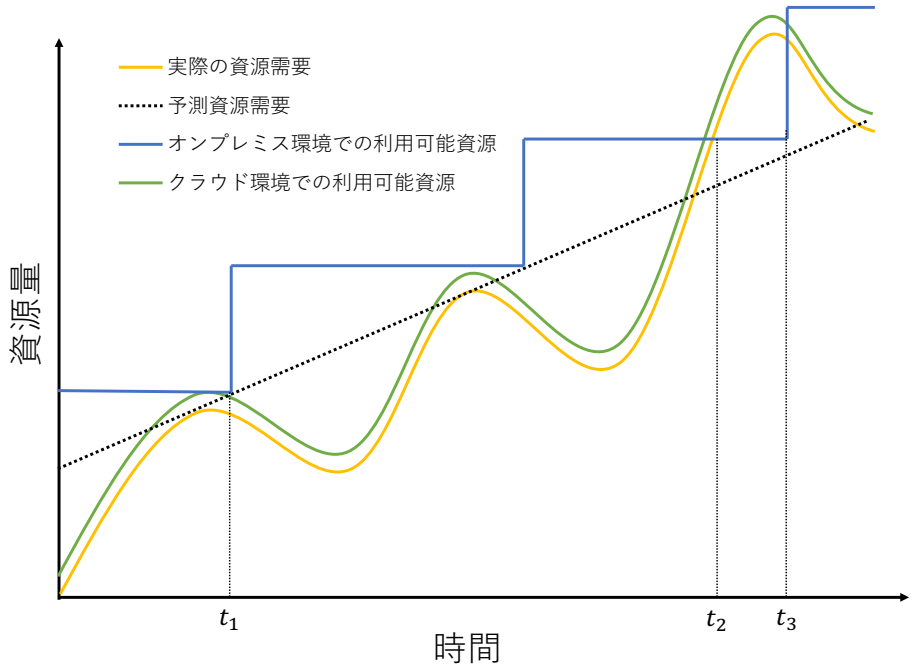


図 1.2: オンプレミス環境およびクラウド環境における割当て資源の変動の概念図

回る時間帯が存在し，当該時間帯においては資源の余剰が発生することになる．さらに，突発的な需要の上昇などによって資源需要が利用可能資源量を上回る状況が損残滓，当該状況において利用可能資源は不足することとなるが，オンプレミス環境においてはサーバ調達に長時間を要するため，当該資源不足を即座に解消することは困難である．図 1.2 では，t_2 の時点において資源需要が利用可能資源を上回り資源不足が発生する．t_2 時点でサーバ調達を開始するが，実際に資源がシステムに割当てられ利用可能となるのは t_3 であり，t_2 から t_3 までの期間は資源不足が続くこととなる．

一方，クラウド環境においてはインスタンスの調達に要する時間が短いために，実際の資源需要の変動に併せてインスタンスを調達・破棄することが可能であるため，オンプレミス環境と比べて大幅に余剰資源の量を削減し，資源の利用コストを削減することが可能である．また，突発的な資源需要の上昇についても，インスタンスを即座に調達することによって必要な資源を短時間で確保することが可能である．

本論文では，図 1.2 におけるクラウド環境での資源調達のように，システムにおいて要

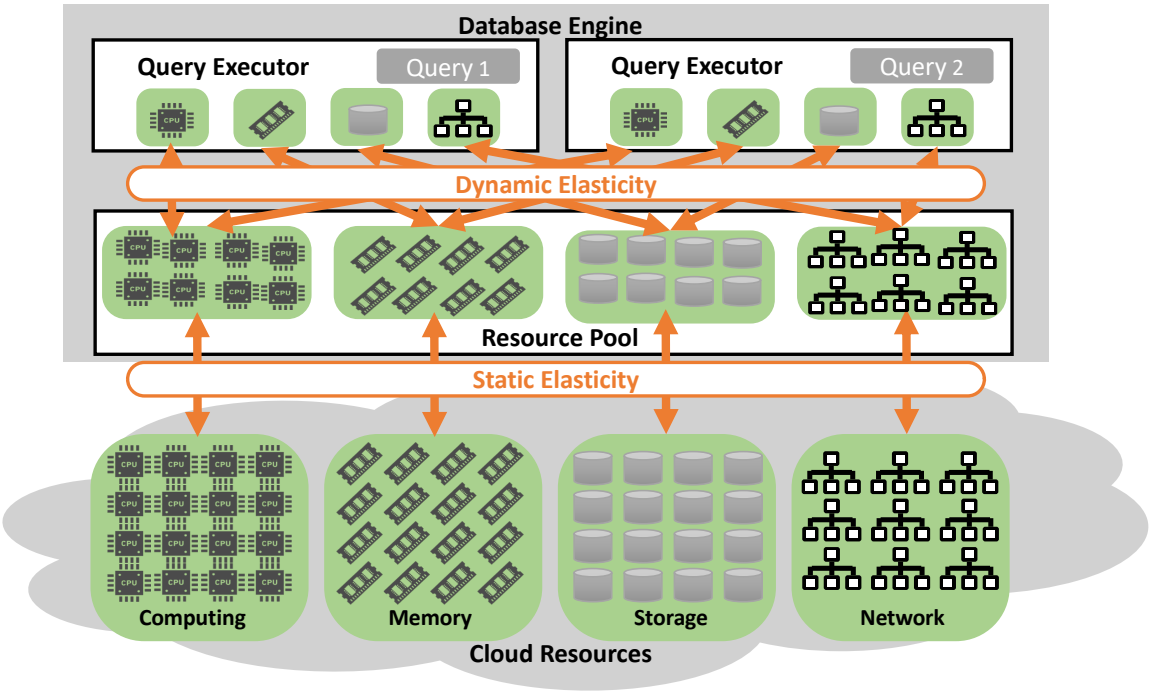


図 1.3: 動的資源伸縮性を有するデータベースエンジン

求に応じて割当てる資源の量を調整可能な性質のことを，**資源伸縮性**と称する．単一のサーバ内で処理が完結するために利用する資源量に応じてシステム全体の性能向上が得られるシステム，例えば Web サーバなどでは，クラウド環境が提供する迅速なオンデマンド資源調達を利用して資源伸縮性を実現することは容易であり，実際に広く利用されている．しかし，並列データベースエンジンのように複数のサーバが協調して動作するソフトウェアにおいては資源伸縮性の実現は容易ではない．

並列データベースエンジンにおける資源伸縮性について，並列データベースエンジン全体として利用可能な資源量の調整が考えられる．本論文では当該性質を**静的資源伸縮性**と称する．従来の並列データベースエンジンの多くはデータベースエンジンに割当てる資源の構成変更にシステムの再起動を要するが，Snowflake[3] や Redshift[4] のようなクラウド環境を前提として設計された並列データベースエンジンでは，データベースエンジンの再起動を行うことなく割当て資源の構成変更が可能であり，これらの並列データベースエンジンにおいては静的資源伸縮性が備えられている．しかしながら静的資源伸縮性ではデータベースエンジンにおいて実行中の各クエリに対してその実行に割当てる資源の量は一定であるため，クエリの実行に割当てる資源については資源伸縮性が実現されていると

は言えない．本論文では，並列データベースエンジンにおいて実行中の各々のクエリに対しその実行に割当てる資源を調整可能な性質を**動的資源伸縮性**と称する．図 1.3 に本研究における目的である動的資源伸縮性を有するデータベースエンジンの模式図を示す．並列データベースエンジンにおける動的資源伸縮性の実現により，クエリへの割当て資源調整による資源利用効率の向上の機会が，従来はクエリ実行開始時のみだったところが任意のタイミングへと大幅に拡大されることになる．実際の並列データベースエンジンの利用においては異なる負荷特性や異なるフットプリントのクエリが随時実行されるため，あるクエリの実行中に他のクエリが完了することにより新たに資源が利用可能など，クエリ実行中に資源の利用状況は変化する．動的資源伸縮性を持たない並列データベースエンジンでは，クエリ実行開始後に利用可能となった資源を割当てることを不可能であるが，動的資源伸縮性を備えた並列データベースエンジンでは，クエリ実行中に新たに利用可能になった資源を，当該クエリの実行に追加の資源として割当てることにより，実行中クエリの処理速度を加速することが可能となる．また，並列データベースエンジンにおける利用可能資源の全てが実行中のクエリに割当てられている場合に，ユーザが優先して完了させたいクエリの実行が要求された状況において．動的資源伸縮性がないデータベースエンジンでは，実行が要求されたクエリの優先度にかかわらず，いずれかの実行中クエリが完了し当該実行中クエリに割当てられた資源が解放されるまで，新たにクエリを開始することはできない．動的資源伸縮性を備えたデータベースエンジンにおいては，実行を要求したクエリより低い優先度のクエリが実行中であれば，当該実行中クエリの資源割当てを減らし，新たに利用可能となった資源を用いることによって高優先度のクエリの実行を即座に開始することが可能となる．

本論文では，データベースエンジンにおいてクエリの実行に用いる資源の一つである演算資源に着目し，共有ストレージ型データベースエンジンにおいて，クエリの実行時に当該クエリの実行に割当てる演算資源をクエリの実行中に調整する手法を提案する．当該手法によって，演算資源がクエリの実行速度を律速している状況においては，クエリの処理スループットを加速，あるいは減速させることが可能となる．本論文では，提案手法を実装した共有ストレージ型データベースエンジンの試作を示し，AWS を用いた当該試作による評価実験を示し，その有効性を明らかにする．

1.1.3　動的資源非均質性吸収を実現する負荷分散機構

クラウド環境におけるインスタンスは，クラウド事業者が所有するデータセンタに収められたサーバを仮想化して提供されている資源であり，ユーザはクラウドサービスにおい

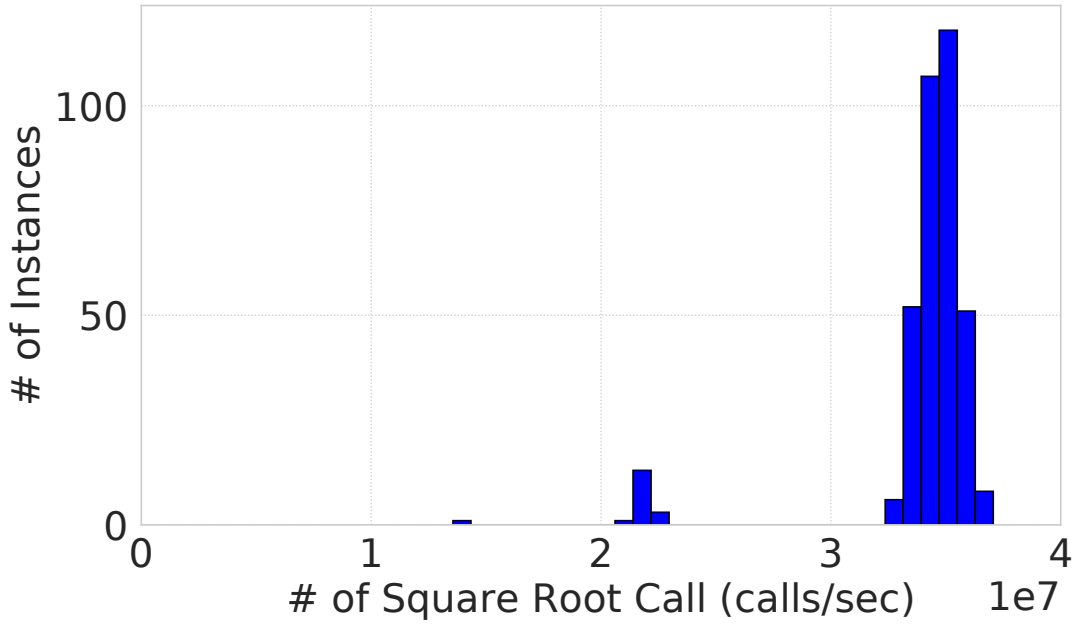


図 1.4: クラウド環境における非均質なインスタンスの例

てオンデマンドにインスタンスを利用することが可能である．しかし，クラウド環境において複数のインスタンスを確保する場合には，以下に述べる理由から均質な処理能力のインスタンスを確保が困難であるという状況が起こり得る．まず，クラウド事業者がユーザに提供可能なインスタンスの量には実際には限りがある為，ユーザは要求した処理能力のインスタンスを常に確保できるとは限らない．当該状況においてはユーザは異なる処理能力のインスタンスを利用する必要がある．また，クラウド環境が提供するインスタンスは仮想化されたものである為，同じ処理能力とされるインスタンスにおいても実際に有する処理能力は異なる可能性がある [5–7]．図 1.4 は当該状況の例の一つを示しており，パブリッククラウド環境の一つである Amazon Web Service(AWS) において，均一な処理能力とされるインスタンス（c4.8xlarge) を 360 インスタンス起動して，各々のインスタンスにおいて 10 秒間にわたって平方根を繰り替え計算し，各々のインスタンスにおいて計測された平方根の秒間計算回数の分布を表したものである．多くのインスタンスは等しい計算回数であったが，18 インスタンスでは 65% 以下の計算回数であり，40% 程度の計算回数のインスタンスが 1 つ存在した．当該結果は，同じ処理能力とされるインスタンスにおいて，実際に得られた処理能力が異なっていたことを示している．さらに，インスタンスは仮想化されたサーバであるため，同一ハードウェア上で実行されている別インス

タンスからの負荷印加や，インスタンスを別のハードウェアに移行するインスタンスマイグレーションなどによって，インスタンスの処理能力が動的に変動する可能性がある．即ち，クラウド環境ではインスタンスは動的な非均質性を有しており，複数のインスタンスを組み合わせて利用するクラスタシステムにおいて，クラウド環境の資源を高効率で利用するためには，クラウド環境の動的資源非均質性を吸収することが求められる．

一方，オンプレミス環境では，自らハードウェアを所有・管理するため，複数のサーバを組み合わせて利用する場合に，各サーバの処理能力を均質なものにすることは容易である．従来の並列データベースエンジンの多くはオンプレミス環境を前提としているため，均質な処理能力のサーバを用いてクエリが実行されることを想定しており，クエリ実行においては各サーバに可能な限り等しい仕事を割り当てる．クエリ実行に割当てるサーバの処理能力に偏りがある場合，各サーバに等しい負荷を割り当てると，処理能力の低いサーバにおける処理時間がクエリ実行時間を律速することとなり，処理能力の高いサーバの利用効率が低下する．非均質なサーバの処理性能に起因するクエリ実行速度低下を軽減するためには，各サーバの処理能力に応じて当該サーバに負荷を割当てることによって，サーバ間の負荷の偏りを均衡化する負荷分散手法が必要となる．さらに，クラウド環境におけるサーバの処理性能は動的に変動する可能性があるため，並列データベースエンジンにおいてクラウド環境の資源を高効率で利用するためには，クエリの実行時に得られる情報を用いて各サーバの処理能力を推定し，当該推定処理能力に基づいて負荷を均衡化する動的負荷分散によって，クラウド環境の動的な資源非均質性の影響を吸収することが求められる．

本論文では，共有ストレージ型並列データベースエンジンについて，演算資源の処理能力に応じて負荷を分散させることにより，演算資源の処理能力の利用効率を向上させる動的負荷分散手法を示す．当該手法は，データベースエンジンにおいてクエリの実行に用いている演算資源の処理能力をクエリの実行中に推定し，当該推定に基づきクエリ処理を構成する各々の演算に要する処理時間を見積ることにより，処理時間が不均衡であった場合には，演算資源間で負荷を移送しすることによって処理時間を均衡化する．本論文ではインデックス結合，ハッシュ結合およびハッシュ集約といった関係データベースエンジンにおける典型的な演算を対象として，演算に要する処理時間の見積り手法および負荷移送手法を示す．一般に，ハッシュ結合はのデータセット空間の広範のデータに対して処理が行われる高選択率のクエリで有効であり，インデックス結合はデータセット空間の一部のデータに対して処理が行われる低選択率のクエリで有効となることが知られている．[8]．ハッシュ結合とインデックス結合の両方について動的負荷分散手法を示すことにより，広範な選択率のクエリに対して動的負荷分散による演算資源の利用効率の向上が得られるこ

とが期待される．本論文では，提案手法に基づく動的負荷分散機構を備えた共有ストレージ型並列データベースエンジンの試作実装を示し，パブリッククラウドをを用いて構成した実験システムにおいて行った評価実験によって，クエリの実行に用いる演算資源の処理能力が均質でない場合には，演算資源の間で負荷がの偏りが生じてクエリの実行時間が長くなるという問題を示すとともに，提案手法によって演算資源の間で負荷を均衡化し，クエリの実行時間が長くなることを軽減することが可能であることを示す．さらに，クエリ実行時に演算資源の処理能力が変動する状況を模擬した評価実験を示し，提案手法によって動的に生じ得る資源の非均質性による影響を軽減することが可能であることを明らかにする

1.2　本論文の構成

本論文の構成は以下の通りである．

第 2 章では，データベースエンジンにおける資源調整ならびに負荷分散に関して，提案されている手法の特徴を纏めるとともに，本研究との関連について論じる．第 3 章では，共有ストレージ型並列データベースエンジンにおける資源伸縮性について論じ，クエリ実行時の演算資源伸縮性を実現する動的演算資源調整手法を示すとともに，インデックス結合，ハッシュ結合を対象として，当該ソフトウェアに基づくソフトウェアの構成手法を示す．第 4 章では，3 章で提案する動的演算資源調整手法の有効性を検証するために，当該手法に基づいた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を示し，パブリッククラウド環境を用いて構成した実験システムにおいて行った評価実験を示し，提案手法の有効性を明らかにする．第 5 章では，共有ストレージ型並列データベースエンジンにおいて起こり得る負荷の偏りについて論じ，当該偏りに起因するクエリ実行性能の低下を緩和する動的負荷分散手法を示すとともに，インデックス結合，ハッシュ結合およびハッシュ集約を対象として当該手法に基づくソフトウェアの構成手法を示す．第 6 章では，5 章で提案する動的負荷分散手法の有効性を検証するために，当該手法に基づいた動的負荷分散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を示すとともに，パブリッククラウドを用いて構成した実験システムにおいて行った評価実験を示し，提案手法によって非均質な演算資源の影響を軽減し，演算資源の利用効率を向上することが可能であることを明らかにする．第 7 章において本論文をまとめ今後の課題について述べる．

第 2 章

関連研究

2.1 並列データベースエンジンにおける資源調整に関する研究

本論文では共有ストレージ型アーキテクチャの並列データベースエンジンを対象とし，当該データベースエンジンにおける動的演算資源調整手法を提案しており，当該手法の有効性を示すために行った試作実装ではキーバリューストアを共有ストレージとして用いた．現在事業者によって提供されているパブリッククラウド環境の多くでは様々な資源がサービスとして提供されており，演算資源として仮想化されたサーバであるインスタンスを利用可能である他に，オブジェクトストレージやキーバリューストアなどのストレージ資源が提供されている．当該ストレージ資源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近年盛んに行われており [9–13]．キーバリューストアをストレージとして用いた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 [14] や，ストレージ入出力での競合を削減し共有ストレージ型データベースエンジンにおいて高速にトランザクションを実行する方法 [15] などが提案されている．

並列データベースエンジンにおける代表的なアーキテクチャとして，本論文で対象とした共有ストレージ型の他，無共有型の並列データベースエンジンが広く用いられている．無共有型データベースエンジンは演算資源においてストレージを共有せずに，演算資源毎にデータを分割して配置し，演算資源でのストレージ入出力の競合をなくすことで，演算資源に対する高いスケーラビリティが得られる [16][17]．無共有型データベースエンジンは，Teradata などのデータウェアハウスや Hadoop[18] を始めとする並列データ処理系などに採用されており．学術界においても，演算資源内でクエリ処理を完結できるようにクエリ実行の履歴に応じてデータの配置を調整する手法 [19–21] や，演算資源間のネット

ワーク通信を削減してクエリ処理を高速化する手法 [22, 23] が提案されている．

Volcano[24] では，exchange operator によって実行計画を複数のグループへと分割し，グループ毎に異なるプロセスで実行することでクエリを並列に実行する．本論文におけるエクスチェンジは Volcano における exchange operator と同等のものであるが，本論文はエクスチェンジによって分割された実行計画の各グループの実行に用いているプロセス数を，クエリの実行中に調整する動的演算資源調整の手法を提案しており，動的な資源調整手法という点において本論文は Volcano とは本質的に異なる．

処理の実行中に当該処理に割当てる演算資源を調整する手法として，SAN によってディスクを共有したクラスタにおいて実行するデータマイニング処理に対して動的資源調整を行う研究や [25–27] や，MapReduce[28] を対象としたものが提案されている．また分散キーバリューストア [29–31] では，キーバリューストアを起動したまま利用する演算資源の量を増減する機能が提供されていることが多い．本論文で提案する動的演算資源調整手法は，関係データベースエンジンにおいてクエリの実行中での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とする．

データベースエンジンにおける複数クエリの実行において，複数クエリ間で重複する演算の重複実行を削減する手法 [32–34] が古くから提案されている他，演算子の単位でクエリ間の共通する処理を検出しクエリ間で演算結果を共有する手法や [35]，集約を含むクエリにおいて共通した集約キーを利用して集約演算を削減するする手法 [36] が提案されている．これらの研究はデータベースエンジンに複数のクエリが同時に実行される状況でクエリの実行を高速化することを目的とする．一方，本論文で提案する手法では，データベースエンジンにおいて複数のクエリに対して，ユーザの要求に基づくクエリ毎の優先度に応じた演算資源割当ての調整を目的としており，クエリ実行の高速化を目的とする研究とはこの点で異なる．

2.2 並列データベースエンジンにおける負荷分散に関する研究

並列データベースエンジンにおいて，データの分布に偏りが存在すると，結合演算において演算資源毎の負荷に偏りが生じてクエリの実行速度が低下することが広く知られており，データの分布の偏りに起因する実行速度の緩和を目的とする研究が盛んに行われてきた [37–42]．本論文は，データの分布の偏りに加えて，演算資源の処理能力の偏りに起因する負荷の偏りに対する負荷分散を対象としている．

パブリッククラウド環境においては異なる処理能力を持つ非均質な演算資源を組み合わせて利用する必要が必然的に発生する [5–7]．文献 [43, 44] ではヘテロジニアスなクラスタにおいて MapReduce を高速に実行する手法を提案している．文献 [45] では，予め各ノードの処理能力を測定しておき，ハッシュ結合において最適なパーティショニングを線形計画法によって決定する Resrouce Bricolage と呼ばれる手法を提案している．本論文では並列関係データベースエンジンにおいて，ハッシュ結合とインデックス結合の両方を対象とするとともに，クエリの実行時に得られた情報を用いて負荷の均衡化を行う動的負荷分散手法を提案している．

文献 [26, 46] では，データのあるキーの値に基づいて送信先ノードを決定し，当該ノードにおいてハッシュテーブルを用いた数え上げを行う，並列相関ルールマイニングにおいて，多くのデータが送られて高負荷となっているノードから低負荷のノードへとハッシュテーブルのバケットを移すことで負荷の均衡化を行う負荷分散手法が提案するとともに，データの分布の偏り，演算資源の処理能力の偏りについて評価を行い有効性を示している．[47] では，並列関係データベースエンジンのハッシュ結合において，同様にバケットの移行による動的負荷分散手法を提案しており，データの分布の偏りを対象とした評価を行っている．本論文における提案手法では，ハッシュ結合に対する動的負荷分散についてはこれらの手法を拡張し，クエリの実行に用いている演算資源の処理能力をクエリ実行時に推定することによって，動的な演算資源性能の変動についてその影響を軽減することを可能としている．

無共有型の並列データベースエンジンにおいてパーティションの配置を自動的に最適化する手法が多く提案されている [19–21, 48, 49]．これらの研究では，OLTP のワークロードを対象とし，複数のノード間をまたいだ分散トランザクションの数を減らすようにパーティションの配置を調整し，トランザクションのスループットを向上させる．本論文では，OLAP ワークロードのクエリを対象とし，クエリの実行中に割り当てられた演算資源間の負荷の偏りを検出し均衡化する動的負荷分散手法を提案している．

第 3 章

共有ストレージ型並列データベースエンジンにおける動的演算資源調整手法

プロセッサの動作周波数の向上は 2000 年台後半から停滞しており [50]，情報システムの性能向上のためには，複数のプロセッサコアを活用すること，更には複数の演算資源をネットワークを介して接続し単一のシステムとして用いることが重要となってきている．データベースエンジンにおいても，複数の演算資源からなるシステムにおいて並列に処理を行うデータベースエンジンが，学術界，産業界の両方から多く提案されている [18, 51–54]．

共有ストレージ型データベースエンジンは複数の演算資源がストレージを共有するアーキテクチャであり，演算資源の故障に対する高い可用性が期待される．共有ストレージ型データベースエンジンは，盛んに研究の対象となっているのみではなく [9–11, 13]，産業界においても商用実装されて広く用いられている [12, 55, 56]．データベースエンジンは，一般に多数のユーザから多様なクエリを受け付け，この際のクエリ毎のユーザの性能要求も様々である．例えば，データ分析を対話的に行う場合には可能な限り短時間でクエリの結果を得たいが，定期的な帳票作成のための集計処理では所定の時刻までにクエリが完了すればよい．この様なユーザからの多様な性能要求を満たすためには，データベースエンジンではクエリ毎にユーザからの性能要求に基づく優先度にもとづき，データベースエンジンが利用可能な演算資源をクエリ実行に割当てることが望ましい．しかしながら，現状のデータベースエンジンでは，クエリの実行に演算資源を割当てる機会はクエリ実行開始時点に限られている．その為，クエリの実行開始後に生じた演算資源の利用状況の変化

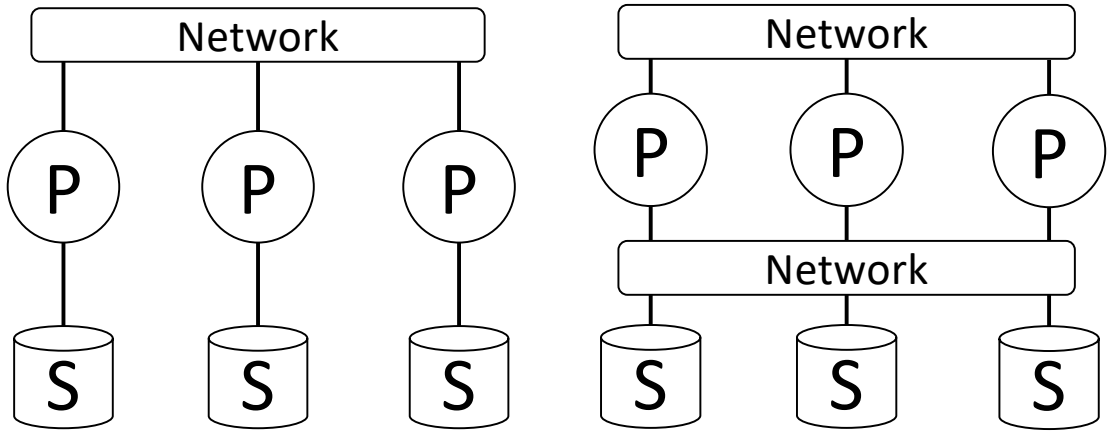
や，新たに実行を要求されたクエリの優先度などに応じて，既に実行中であるクエリの演算資源の割当てを変更することはできない．

本章では，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割当てる演算資源を動的に調整する手法を提案する．例えば，クエリの実行中に新たに演算資源が利用可能となった場合において，当該演算資源を実行中クエリに追加して割当てることにより，当該クエリの実行を加速することがすることが可能となる．また，演算資源が全て実行中の他のクエリに割当てられているために利用可能な演算資源が残っていない状況で，高優先度のクエリ実行が要求された場合において，低優先度の実行中クエリに対する演算資源の割当てを減らすことにより，高優先度クエリの実行を開始することが可能となる．本章の提案手法によって，クエリ実行開始後の演算資源の状況や，新たに実行を要求されたクエリの優先度に基づいて，実行中クエリに割当てる演算資源を動的に調整することにより，従来より効率よく演算資源を利用し，よりクエリ毎の優先度に応じた実行が期待される．著者らの知る限り，同様の研究は他に見当たらない．

3.1 並列データベースエンジン

関係データベースエンジンは，理論的基礎となる関係モデルが 1970 年に E.F.Codd によって提案されて以降 [57]，System R[58] や Ingres[59] を始めとした数多くの研究がなされ，現在では Oracle[60] や IBM DB2[61] などの商用データベースエンジンや，PostgreSQL[62] や MySQL[63]，Firebird[64] などのオープンソースソフトウェアのデータベースエンジンなど数多くの製品がが開発され，IT システムにおいてデータ管理・検索の中心的な役割を果たすソフトウェアとして広く利用されている．関係データベースエンジンでは，データはリレーションと称する表の形式によってモデル化され，データの検索はユーザが結果として要求するリレーションを関係代数演算と称する演算によって宣言的に記述することによって行われる．関係代数演算を記述する言語として一般に SQL と称する言語が用いられる．

並列データベースエンジンは，複数のプロセッサを用いてクエリの実行を高速化する関係データベースエンジンである．並列データベースエンジンの研究は，1970 年代に行われたデータベース専用の並列計算機であるデータベースマシンの研究に端を発する [65–68]．データベースマシンは専用にカスタマイズされたプロセッサやストレージを用いるためにコストに見合う性能を得ることが難しく，研究・開発の対象は汎用のハードウェア上で動作する動作する並列データベースエンジンへと移行していき，産業界からも 80 年代に Teradata[69] や Tandem[70, 71] といった製品が開発されてから，2000 年代に入っても



(a) 無共有型　　(b) 共有ストレージ型

図 3.1: 並列データベースエンジンの代表的なアーキテクチャ．P はプロセッサ，S はストレージを表す．

Netezza[72] や Vertica[73] などが開発されるなど，企業においても広く利用されている．

3.1.1　並列データベースエンジンのアーキテクチャ

並列データベースエンジンに用いられるアーキテクチャの主なアーキテクチャとして，無共有型と共有ストレージ型が挙げられる．無共有型は個々のプロセッサがストレージを共有せず，データセットを予め分割して各プロセッサに接続されたストレージに配置するため，各プロセッサは当該プロセッサに割当てられたデータセットにのみアクセス可能である．共有ストレージ型は複数のプロセッサがネットワークを介して一つのストレージを共有するため，全てのプロセッサはデータベース内の全てのデータセットにアクセスが可能である．

無共有型と共有ストレージ型のアーキテクチャについては多くの比較がなされており [16, 17]，多くの違いが挙げられているが，クエリの実行性能に関する違いとしては大きく二つの要因が挙げられる．一つ目の要因としてはデータの事前分割によるものが挙げられる．無共有型は事前にデータを分割して各プロセッサに配置を行い，ユーザからクエリを与えられた時に，クエリ内でのタスクの分割と事前のデータ分割とを合わせることが可能である場合には，クエリの実行時にプロセッサ間での通信を削減し，クエリの実行を高速化することが可能となる．タスクの分割をデータ分割に合わせることが可能なクエリについては無共有型が優位となるが，タスクの分割とデータ分割を一致させることが不可能

なクエリについては，無共有型と共有ストレージ型との間でデータの事前分割による性能差はなくなるものと考えられる．

二つ目の要因として各々のアーキテクチャを構築するためのコストが挙げられる．無共有型では個々のプロセッサとストレージとをバスで接続するのに対し，共有ストレージ型では全プロセッサとストレージとをネットワークで接続することが必要となるため，無共有型と比較して共有ストレージ型でシステムを構築するコストは大幅に高く．仮に共有ストレージ型におけるネットワークにおいて，無共有型のバスと同じ帯域幅を確保することができた場合には，共有ストレージ型と無共有型の間でストレージ入出力性能による差はなくなるものと考えられるが，実際にはシステムの構築コストが要因となって，共有ストレージ型での入出力帯域は無共有型と比べ小さくなる場合が多い．

本論文では，データベースエンジンのアーキテクチャとして共有ストレージ型データベースエンジンを対象とし，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中に演算資源割当ての調整を行う手法を提案する．

3.1.2 並列データベースエンジンの結合演算方式

関係データベースエンジンにおける代表的な関係代数演算の一つとして結合演算と称する演算が挙げられる．結合演算は複数の表を入力として受け取り，各々の表を構成するタプルについて，クエリによって指定された条件[*1]を満たすタプル同士を結合したものを結果のタプルとし，当該結果タプルから構成される表を結果として返す演算である．結合演算を実行する主な方式として，ネステッドループ結合（ならびにネステッドループ結合においてインデックスを用いたインデックス結合），ハッシュ結合，ソートマージ結合が挙げられる．本論文では，並列データベースエンジンにおいて主に用いられる，インデックス結合ならびにハッシュ結合を対象とする．

まずインデックス結合について述べる．関係データベースエンジンの多くでは，表の検索効率をを向上させるために，インデックスと称するデータを作成することが可能である．インデックスは表のあるキー値を指定し，当該キーの値からタプル全体を取得するための効率のよいデータ構造を予め作成することによって，当該キーの値によって表を探索する際の効率を向上させるものである．インデックスに用いられるデータ構造としては，B+ 木 [74] やハッシュテーブルなどが主に用いられる．インデックス結合は，ネステッドループ結合においてタプルの探索時にインデックスを用いる結合方式である．ネステッド

[*1] 結合条件について，各々の入力表のタプルにおける特定のキーの値の一致を条件とする等価結合と，キーの値の範囲などを条件とする非等価結合が存在する．

ループ結合では入力として受け取る二つの表を外表，内表と称する．ネステッドループ結合では，まず外表について一つのタプルを取得し，内表のタプルを全て走査することによって当該タプルとの結合条件を満たす内表のタプルを探索し，両タプルを結合することによって結果のタプルとする．当該走査を外表の全てのタプルについて行うことにより，結合タプルから構成される結果表を生成する．即ち，ネステッドループ結合は外表のタプルを取得するループ処理と，当該ループ処理により得られた外表の各々のタプルについて結合条件を満たす内表のタプルを探索するループ処理から構成される二重のループ処理によって実行される．インデックス結合は，ネステッドループ結合において内表の結合条件のキーについて予めインデックスが作成されている場合に用いることが可能な結合方式であり，結合条件を満たす内表タプルの探索時にインデックスを用いることによって実行される．インデックス結合は，外表の一タプルごとに内表の探索が行われる．即ち，外表に対して選択条件が指定されて，結合のための内表の探索が少なくなる場合に，実行効率がよくなるという特徴がある．ネステッドループ結合ならびにインデックス結合の並列化は外表のループ処理，即ち外表の各タプルを取得する処理を複数のプロセッサにおいて並列に実行し，その後に各プロセッサにおいて結合条件を満たす内表のタプルを探索することによって行われる．共有ストレージ型では各プロセッサは全てのプロセッサが内表の全てのデータにアクセス可能であるため，結合条件を満たす内表タプルの探索は一つのプロセッサで実行することが可能であるが，無共有型では内表のデータは各プロセッサに分割して配置されているため，外表のタプルを取得した後に当該タプルを全てのプロセッサに対して分配し，全てのプロセッサにおいて結合条件を満たす内表タプルの探索が行われる．

次にハッシュ結合について述べる．ハッシュ結合はビルドフェーズとプローブフェーズという 2 つのフェーズを順に実行する．ビルドフェーズで対象とする表をビルド表，プローブフェーズで対象とする表をプローブ表と称する．まず，ビルドフェーズではビルド表の全てのタプルを取得し，ビルド表の各々のタプルについて結合条件で指定されたキーの値をキーとし，タプル全体を値としたハッシュ表を作成する．プローブフェーズではプローブ表を全てのタプルを取得し，各々のタプルについて検索条件で指定されたキーの値を用いてビルドフェーズで作成したハッシュ表を探索し，一致するタプルがある場合には両タプルを結合することによって，結果表を生成する．ハッシュ結合では，両方の入力表について常に全てのタプルを選択するため，選択条件による実行効率への影響は小さい．ハッシュ結合の並列化手法は文献 [75, 76] によって提案されており，まずビルドフェーズにおいて，ビルド表の各々のタプルについて結合キーの値のハッシュ値を計算し，当該ハッシュ値に基づいてタプルを各プロセッサに分配した後に，各プロセッサにおいて分配

されたタプルを用いてハッシュテーブル構築する．次にプローブフェーズにおいて，ビルドフェーズと同様に結合キーの値のハッシュ値に基づいてタプルを各プロセッサに分配した後に，各プロセッサにおいてハッシュテーブルを参照して結合処理を実行する．並列ハッシュ結合では結合キーの値のハッシュ値によってタプルの分配を行うため，各入力表において結合キーの値の分布に偏りが存在する場合に各プロセッサにおける負荷に偏りが発生して各プロセッサの利用効率が低下する問題が知られている．当該問題に対して各プロセッサの負荷の均衡化を行う負荷分散の研究が広く行われており，当該問題については 5 章において論じる．

3.2　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

3.2.1　動的演算資源調整

従来の共有ストレージ型データベースにおける静的な演算資源割当てではクエリ開始時に演算資源の割当てが決定され，クエリ実行中には演算資源の割当ては変更できないのに対し，本章で導入する動的演算資源調整では実行中クエリに対する演算資源の調整を可能とする．動的演算資源調整によって，随時変化する演算資源の利用状況やユーザからの多様なクエリの実行要求に基いて，実行中クエリへの演算資源の割当てを調整し，より効率のよい演算資源を利用や，ユーザの要求に基づくクエリ毎の優先度に応じたクエリ実行が可能となる．

実際のデータベースエンジンの利用においては異なる負荷特性やフットプリントのクエリが随時実行されるため，あるクエリの実行中に他のクエリが完了することにより，新たに演算資源が利用可能となる．従来の共有ストレージ型データベースエンジンにおける静的な演算資源割当てでは，実行開始時に利用可能であった演算資源のみを用いることができるが，動的演算資源調整を行うことで実行中のクエリに追加の演算資源として割当てることにより，実行中クエリを加速することが可能となる．また，データベースエンジンが利用可能な演算資源が実行中のクエリに全て割当てられている状況で，高優先度のクエリの実行が要求された場合を考える．従来の静的な演算資源割当てでは，実行が要求されたクエリの優先度にかかわらず，いずれかの実行中クエリが完了し当該実行中クエリに割当てられた演算資源が解放されるまで，新たにクエリを開始することはできない．実行を要求したクエリより低い優先度のクエリが実行中であれば，動的演算資源調整によって当該実行中クエリの演算資源割当てを減らし，当該演算資源を割当てることで高優先度のクエ

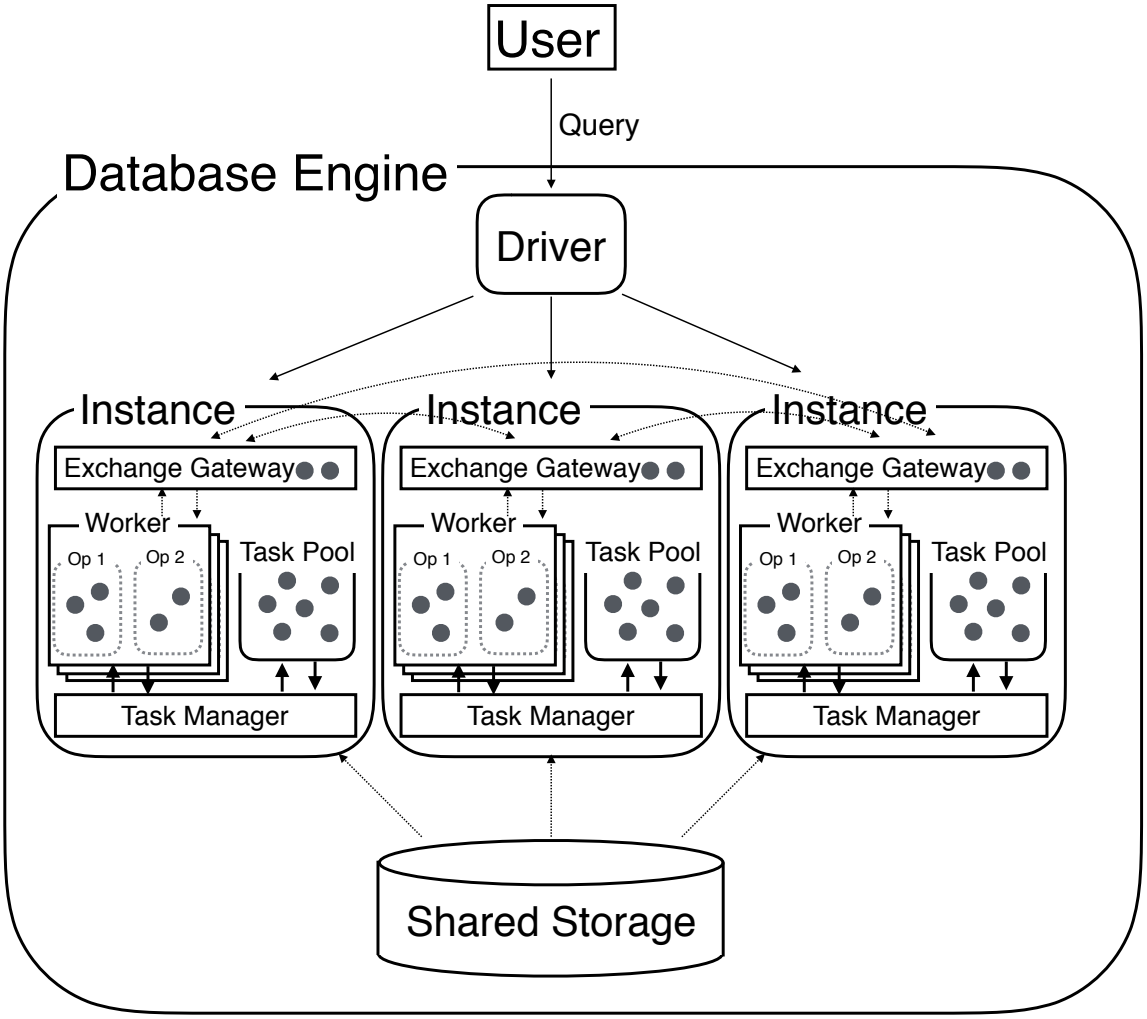


図 3.2: 動的演算資源調整を可能とする共有ストレージ型データベースエンジンの概要

リの実行を即座に開始することが可能となる．

3.2.2 動的演算資源調整を可能とする共有ストレージ型データベースエンジンの概要

図 3.2 は動的演算資源調整を可能とするデータベースエンジンの概要である．図 3.2 のデータベースエンジンは，複数の演算資源（インスタンス）が一つのストレージを共有する共有ストレージ型のアーキテクチャである．当該データベースエンジンにおけるクエリの実行は，ドライバと称す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．ドライバはユーザからクエリを受け取ると，受け取ったクエリに基いて実行計画を作成

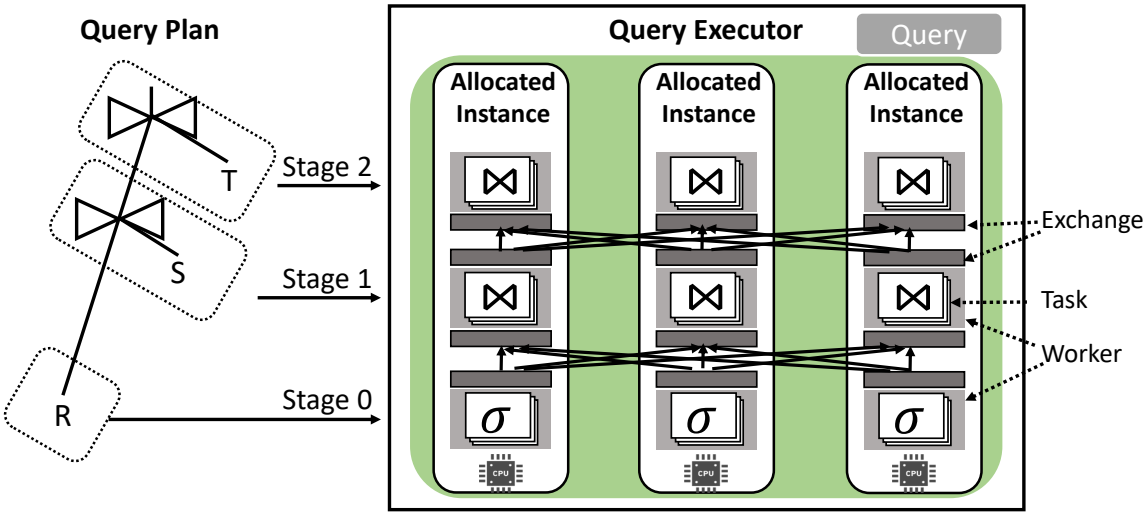


図 3.3: クエリ（$\sigma(R) \bowtie S \bowtie T$）の実行計画とクエリ演算の実行方法の例

し，当該実行計画に含まれる演算を複数のグループに分割する．本章において当該グループを段と称し，各段の実行はそれぞれ異なるプロセスによって行われる．ドライバは段を実行するためのワーカと称するプロセスを各インスタンスにおいて起動し，各ワーカにいずれかの段の実行を割当てる．

実行計画を複数の段に分割したため，ある段の演算の実行に異なる段の演算の結果を必要とする場合がある．その為，クエリ全体の実行には，異なる段の間での演算結果の送受信，即ちワーカ間でのタプルの送受信が必要となるが，当該送受信はエクスチェンジと称する処理によって行われる．例えば，スキャン演算および選択演算を含む段を割当てられたワーカにおいては，共有ストレージからのスキャン入力と，当該入力から得られたタプルに対する選択を行った後に，後段のワーカが存在する場合には，後段のワーカへエクスチェンジを介してタプルを送信する．

ワーカにおける段の実行は，入力データであるタプルについて，1 つのタプルと当該タプルに関する演算の実行状態を紐づけたものを単位として行われる．本章において当該実行単位をタスクと称する．前段のワーカからエクスチェンジによってタプルを受け取ったワーカは，当該タプルの演算実行のためのタスクを生成し，タスクプールへと挿入する．タスクプールのサイズは有限であり，タスクプールが満杯で新たなタスクを挿入不可能である間にはタプルの受取りを停止する．タスクマネージャはタスクプールに滞留するタスクを，ワーカ毎の同時実行タスク数，およびワーカ全体の合計実行タスク数に対して設定された上限を超えないようにタスクを選択し，当該タスクをインスタンスの有する複数の

演算資源に振り分けて実行する，という処理を繰り返す．

ワーカ間においてエクスチェンジを介してデータが送受信されるため，クエリ実行全体では複数のエクスチェンジによって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各々のワーカがインスタンスの演算資源を利用して実行計画に基づく演算を実行する．図 3.3 に，$\sigma(R) \bowtie S \bowtie T$ なるクエリについて，クエリの実行計画とクエリ実行におけるパイプラインの例を示す．当該クエリの実行計画は 3 段から構成されるものであり，クエリの実行においては，実行計画中の各段に対応するワーカが各インスタンスにおいて起動され，隣接する段のワーカ間ではエクスチェンジを介してデータを送受信することで，3 段のパイプライン構造を構成する．

パイプラインにおける各ワーカでの演算の実行は，前段のワーカから受信したタプルによって駆動される．即ちワーカが前段ワーカからタプルを受け取ると，当該タプルに基づく処理が開始される．その後に当該処理の結果として得られたタプルは即座に送信対象先へと送られる，所謂プッシュ型の方式によって後段ワーカへと送信される．例えば結合演算を実行するワーカにおいては，ワーカが新たなタプルを受信すると，当該タプルの結合対象となるタプルを取得するためのストレージへの入出力が実行される．その後に，受信したタプルと入出力の結果として得られたタプルとの結合が行われ，当該結合処理によって得られたタプルはエクスチェンジを介して即座に後段のワーカへと送信される．実行計画の 1 段目の演算を実行するワーカについては前段のワーカが存在しないため，前段ワーカからのタプルの受信によって処理が駆動されるのではなく，当該 1 段目ワーカの処理は常に駆動され続ける．このように，各ワーカにおいて前段のワーカから受信したタプルによって処理を駆動し，処理の結果として得られたタプルをプッシュ型によって後段ワーカへと送信することによって，クエリは複数のワーカと複数のエクスチェンジから構成されるパイプライン処理によって実行される．

図 3.3 に示した $\sigma(R) \bowtie S \bowtie T$ の例においては，1 段目のワーカでの R からタプルの取得，および取得したタプルに対する選択処理が常に駆動され続け，処理の結果として得られたタプルは 2 段目のワーカへと送信される．2 段目のワーカにおいては，受信した R のタプル毎に処理が駆動され，当該受信タプルの結合対象となる S のタプルを取得して，結合したタプルを 3 段目のワーカへと送信する．3 段目のワーカにおいては，受信したタプル毎に処理が駆動され，当該受信タプルの結合対象となる T のタプルを取得することで，クエリの結果の一部となるタプルが得られる．$\sigma(R) \bowtie S \bowtie T$ のクエリはこの 3 段のパイプラインによって実行され，1 段目のワーカが全てのタプルを取得し，当該タプルによって駆動される 2，3 段目のワーカの処理が完了することによって，パイプライン全体の処理も完了しクエリの結果が得られる．

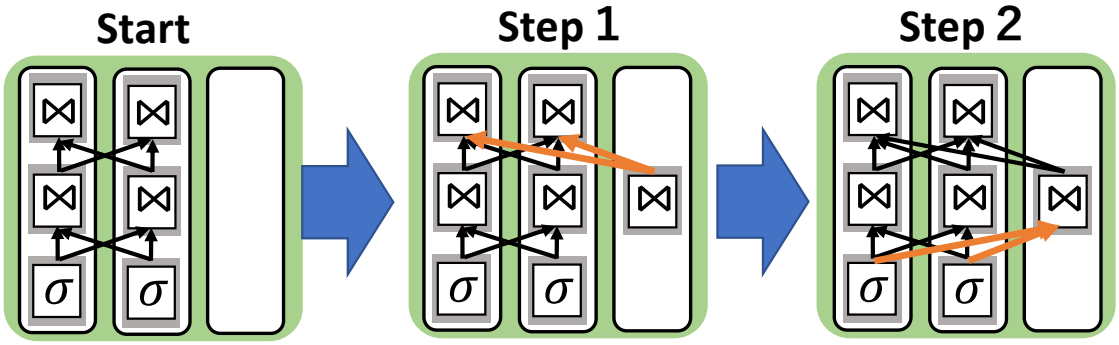


図 3.4: ワーカ起動による実行中クエリへの演算資源追加

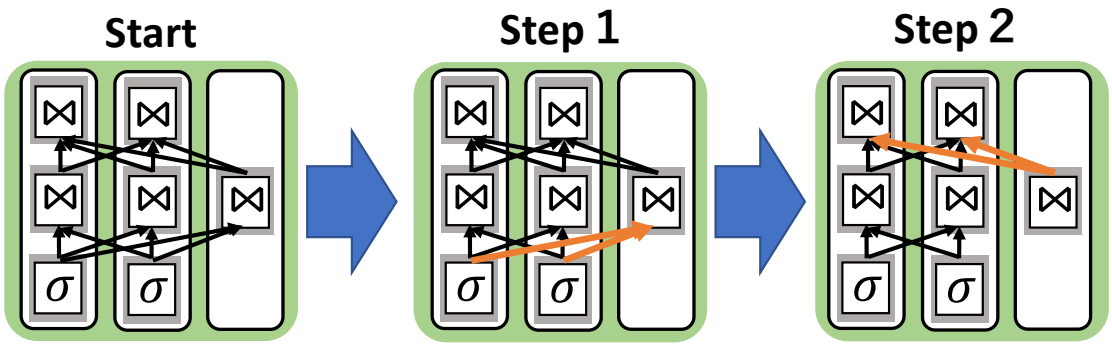


図 3.5: ワーカ停止による実行中クエリからの演算資源削除

3.3 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

3.3.1 ステートレスな演算に対する動的演算資源調整手法

前節に示した共有ストレージ型データベースエンジンにおいて，状態を持たない演算から構成されるクエリに対しては，新たなインスタンスでワーカを起動する，あるいはワーカが起動中のインスタンスにおいてワーカを停止することによって，クエリ実行に割当てる演算資源を追加，あるいは削除することが可能になる（図 3.4，3.5）．しかしながら，クエリの実行中においては，各ワーカには前段のワーカからエクスチェンジを介して随時データが送られてきており，クエリの実行を正しく行うためには，当該ワーカに送られたデータは実行計画に記述された演算に基いて正しく処理された後に，後段のワーカへと全て送らなければならない．即ち，クエリの実行結果を正しく保ったまま動的演算資源調整

Algorithm 1 i 段目のワーカを追加する方法

Require: i: 起動するワーカの段
Require: num: 起動するワーカ数

```
1: function AddWorkers(i, num)
2:     targets ← choose num from available instances
3:     prevWorkers ← workers of stage i − 1
4:     nextWorkers ← workers of stage i + 1
5:     for t ← targets do
6:         start worker of stage i on t
7:         open connections from worker on t to nextWorkers
8:     end for
9:     newWorkers ← workers on targets
10:    for p ← prevWorkers do
11:        open connections from p to newWorkers
12:        start to send data from p to newWorkers
13:    end for
14: end function
```

を行うためには，ワーカの起動・停止とともにワーカに送られた全てのデータを正しく処理し後段のワーカへと送る必要がある．

アルゴリズム 1 に演算資源の追加，即ち新たなインスタンスにおいてワーカを起動する手法を示す．演算資源の追加においては，ワーカを起動するとともに，当該ワーカを前段のワーカのデータ送信先として追加する．しかし，新規ワーカにおいてデータを受信可能となる前に，前段ワーカから新規ワーカへのデータ送信が行われると，新規ワーカにおいて当該データを受信できず正しく処理することはできない． アルゴリズム 1 に示したワーカ起動手法において，新規ワーカを前段ワーカのデータ送信先として追加するタイミングは，新規ワーカにおいて正しくデータを受け取り処理する準備が整った後となる．新規ワーカの準備が整った後に前段ワーカにおいてデータ送信先として追加しデータの送信を開始することで，新規ワーカにおいてデータを正しく受信し処理可能であることを可能としている．

アルゴリズム 2 に演算資源の削除，即ちワーカを実行中のインスタンスにおいてワーカを停止する手法を示す．演算資源の削除においては，ワーカの停止を行うとともに，前

Algorithm 2 i段目のワーカを停止する方法

Require: i: 停止するワーカの段
Require: num: 停止するワーカ数

```
1: function RemoveWorkers(i, num)
2:     targets ← choose num from workers of stage i
3:     prevWorkers ← workers of stage i − 1
4:     nextWorkers ← workers of stage i + 1
5:     for p ← prevWorkers do
6:         stop sending data from p to targets
7:         close connections from p to targets
8:     end for
9:     for t ← targets do
10:        receive all data from prevInstances on t
11:        process all data on t
12:        close connections from t to nextWorkers
13:        stop worker on t
14:    end for
15: end function
```

段ワーカから当該停止ワーカへ送信済みのデータを正しく処理した後に，後段ワーカへ送信する必要がある．アルゴリズム 2 に示したワーカ停止手法において，停止ワーカを前段ワーカのデータ送信先から削除するタイミングは，ワーカ停止の一連の手続きの一番初めとなる．まず前段ワーカにおいてデータ送信先から停止ワーカを削除し，前段ワーカから停止ワーカへのデータ送信を止める．その後，当該停止ワーカにおいて前段ワーカからのデータを全て受信するまで待ち，受信した全てのデータを実行計画に基いて処理し後段ワーカに送信した後に，当該ワーカを停止する．以上の手続きにより，停止ワーカにおいて全てのデータを正しく処理することを可能としている．

本節で示した演算資源の追加・削除の手法は，起動・停止を行うワーカと前段ワーカとが協調して手順を進める必要がある．提案する共有ストレージ型データベースエンジンにおいては，ドライバが演算資源の追加・削除を管理し，ドライバが各ワーカに対して前述の手続きに則って指示を行うことで，演算資源の追加・削除の手順を進める．即ち，演算資源の追加においては，ドライバがワーカの起動を行うインスタンスを決定し，当該イン

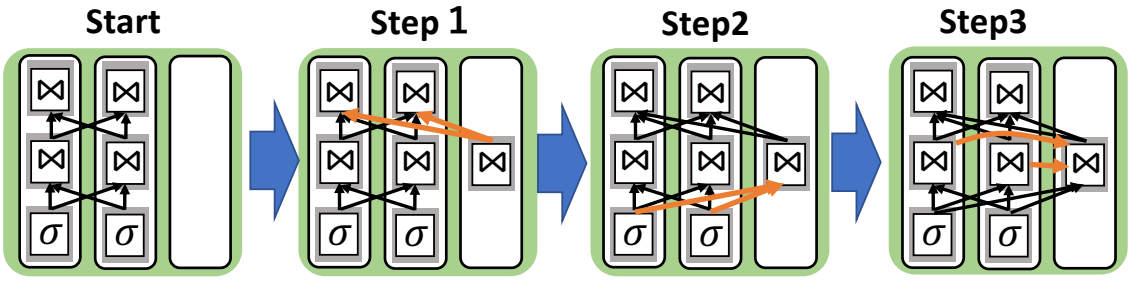


図 3.6: ステートフルな演算を含む実行中クエリへの演算資源追加

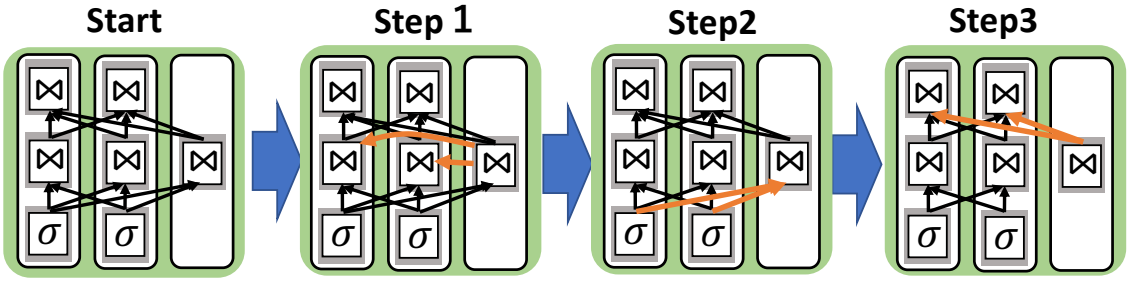


図 3.7: ステートフルな演算を含む実行中クエリからの演算資源削除

スタンスにおいてワーカ起動を指示した後に，当該起動ワーカをデータの送信先として追加するよう前段ワーカに指示する．演算資源の削除においては，ドライバがワーカの停止を行うインスタンスを決定し，当該停止ワーカをデータ送信先から削除するように前段ワーカに指示した後に，当該ワーカに停止の指示を行う．また，提案手法ではワーカの起動・停止を実行する際に，前段および後段ワーカとの接続を変更するため，新規ワーカの起動・停止中に新たに前段および後段のワーカを起動・停止することはできない，即ち隣接する複数の段のワーカに対して同時に起動を実行することはできない．複数の段のワーカの起動・停止順序において他の制約はなく，任意の段のワーカから起動・停止することが可能であり，また隣接しない段のワーカであれば複数の段のワーカの同時起動も許容される．

3.3.2　ステートフルな演算に対する動的演算資源調整手法

クエリを構成する各々の演算が実行状態を持たない場合には，前節で示したワーカ起動・停止手法によって，実行中クエリへの資源割当ての調整が可能となる．一方，実行状態を持つ演算を含むクエリの資源調整においては，ワーカの起動・停止のみではなく，インスタンスの追加する場合には起動中ワーカから新規ワーカへ，インスタンス削除の場合

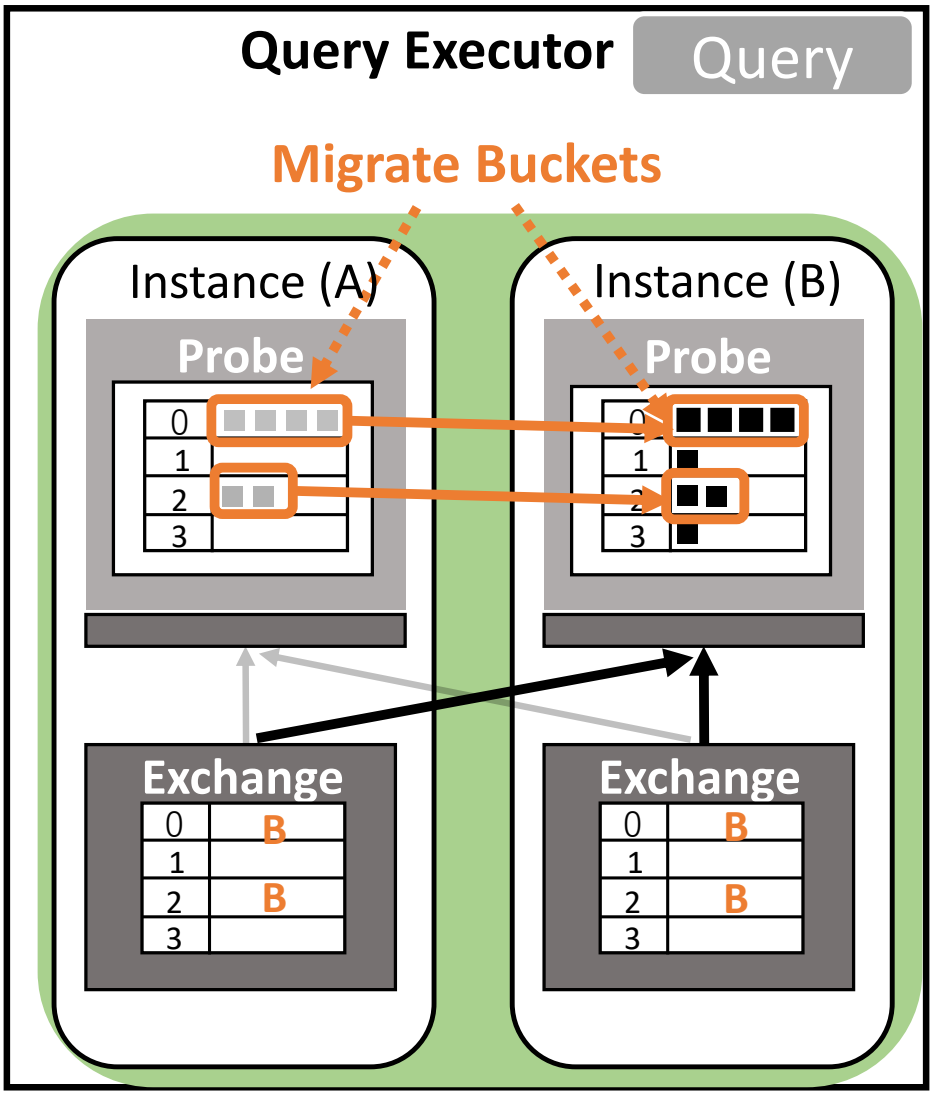


図 3.8: ハッシュテーブルのバケット移送

には停止ワーカから残余ワーカへと実行状態を移送する必要がある（図 3.6，3.7）．

並列データベースにおいて実行状態を持つ代表的な演算として，ハッシュ結合が挙げられる．並列データベースエンジンにおけるハッシュ結合では，ハッシュテーブルを構築するビルドフェーズにおいて，エクスチェンジを介してタプルを送信する際に，タプルに含まれる値に基づいてタプルの送信先を決定することによって，各計算機にハッシュテーブルを分割し，構築されたハッシュテーブルを用いて結合処理を行うプローブフェーズにおいても，同様の送信先決定を行うことによって，プローブ処理を計算機間に分散する．ハッシュ結合に対して演算資源調整を行う場合，演算資源の追加において単にワーカを起

動しただけでは当該ワーカにはハッシュテーブルが存在しないため，ビルドフェーズにおけるハッシュテーブルの構築，ならびにプローブフェーズにおける結合処理を実行されず，当該追加計算機の処理能力は利用されない．演算資源削除の場合にはワーカを停止すると，当該ワーカにおいて構築中，または構築済みであるハッシュテーブルが失われてハッシュ結合を正しく継続することが不可能になる．その為，ハッシュ結合に対する演算資源調整において，演算資源を追加する場合には，新規ワーカ起動後に他の演算資源から当該新規ワーカへとハッシュテーブルの一部を移送する必要があり，演算資源を削除する場合には，ワーカの停止の前に当該ワーカから他のワーカへとハッシュテーブルの一部を移送する必要がある．提案手法においては，バケットを単位としてインスタンス間でハッシュ結合の状態を移送する手法を用いている．当該手法によって，あるワーカから他のワーカへとバケットを移送し，当該バケットに紐づくタプルの送信先を切り替えることによって，ハッシュ結合に対する動的演算資源調整を実現している．

3.4 動的演算資源伸縮を用いた資源割当てのスケジューリング

本論文において提案する動的演算資源伸縮によって，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当該クエリの実行に割当てる演算資源の調整が可能となる，即ち，データベースエンジン全体としては実行中クエリを含めたプリエンプティブな資源割当てのスケジューリングが可能となる．データベースエンジンにおけるクエリへの資源割当てのプリエンプティブなスケジューリングは従来のデータベースエンジンでは不可能だったものであり，プリエンプティブなスケジューリングを前提とした資源割当てのポリシーによって，従来よりユーザの要求に応じたデータベースエンジンにおける資源利用が可能になる．

各クエリへの資源割当てを決定するポリシーについて，まずユーザがクエリ毎の優先度を指定する状況において，各々のクエリに設定された優先度に応じて当該クエリへの資源割当てを決定する方法が考えられる．クエリの優先度に応じた資源割当てのスケジューリングとして，ユーザがクエリ i について，優先度 p_i を指定する時，各々のクエリに割当てるインスタンス数 N_i の比が，優先度 p_i の比に一致するように N_i を決定する．この場合，N_i は次式によって求められる．

$$N_i = \frac{p_i}{\sum p_i} N_{total} \tag{3.1}$$

ここで，N_{total} はスケジューリング時点において利用可能なインスタンス数の合計を表す．

クエリ毎の優先度によって資源割当てを決定するポリシーの他，ユーザがクエリ毎にデッドライン，即ちクエリ毎に当該クエリの実行完了までに超過しないことを要求する時刻を指定する状況において，各々のクエリに設定されたデッドラインに応じて当該クエリへの資源割当てを決定する方法が考えられる．当該状況において，全てのクエリのデッドライン超過時間を可能な限り小さくすることが望ましい．この為には，スケジューリング時点においてデッドラインまでの時間が最も近いクエリを可能な限り早く完了させる，即ち当該クエリに可能な限りの最大限のインスタンス数を割当てる．クエリ i に割当てるインスタンス数 N_i は次式によって決定される．

$$N_i = \begin{cases} N_{total} - Q_N * N_{min} & (i = argmin(d_i)) \\ N_{min} & (otherwise) \end{cases} \tag{3.2}$$

ここで，N_{total} はスケジューリング時点において利用可能なインスタンス数の合計，N_{min} はユーザによって指定されたインスタンス割当て数の下限値，Q_N は実行中のクエリ数を表す．

動的演算資源伸縮を用いたクエリスケジューリングでは，データベースエンジンにおいて実行中のあるクエリが完了した時点，もしくは新たなクエリの実行が要求された時点のいずれかにおいてスケジューリングを開始し，ユーザによって指定されたポリシーに従って，各クエリへの割当てインスタンス数を決定する．各クエリについて，決定された割当てインスタンス数と現在の割当てインスタンス数が異なる場合，まずは現在の割当てインスタンス数が超過しているクエリに対して，演算資源調整によって資源削除を行い，次に割当てインスタンス数が不足しているクエリに対して演算資源調整に資源追加を行うことによって，全てのクエリに割当てインスタンス数とポリシーによって決定されたインスタンス数とを一致させる．

第 4 章

演算資源調整機構を備えた共有ストレージ型並列データベースエンジンの試作実装と評価

本論文で提案する動的演算資源調整手法は，従来不可能であった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その実行に割当てる資源の調整を可能とする点において特徴的である．本章では，動的演算資源調整手法の有効性を検証するために，当該手法に基づいた動的演算資源調整機構を有する共有ストレージ型並列データベースエンジンの試作実装を示し，パブリッククラウドを用いた実験システムにおいて行った評価実験を示すとともに，データベースエンジンの利用状況を模擬したケーススタディによって，提案手法の有効性を明らかにする．

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

前章で示した動的演算資源調整手法の有効性を実験によって評価するために，当該手法に基づいた共有ストレージ型データベースエンジンを試作した．当該試作のシステム構成図を図 4.1 に示す．

当該試作におけるクエリの実行方式は，アウトオブオーダ型実行方式 [77, 78] に基づいたものを用いた．当該実行方式は，クエリの実行時に入出力処理に基づいて動的にタスク分解を行い，当該タスクを複数の演算資源へと振り分けて並列に実行する．また，データ

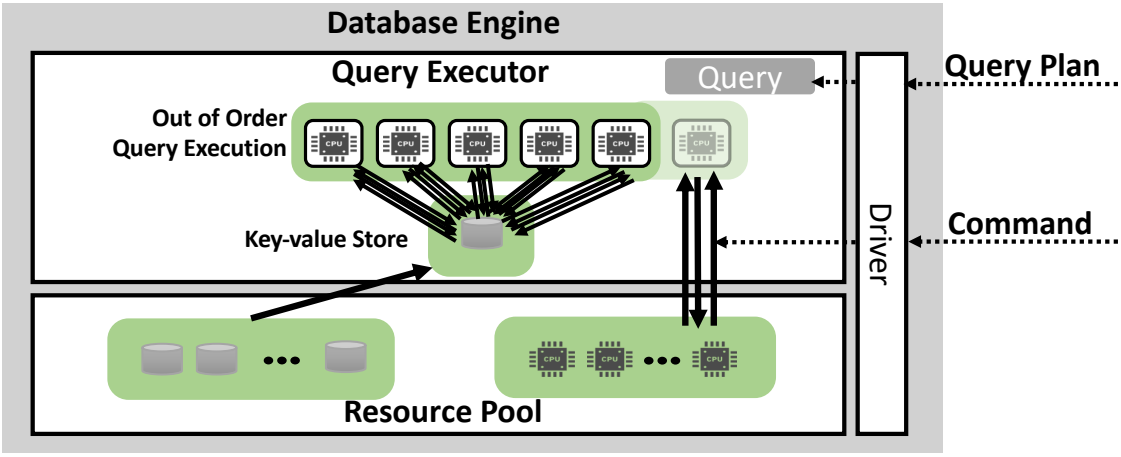


図 4.1: 動的演算資源調整機構を有する共有ストレージ型並列データベースエンジンの試作実装

ベースエンジンにおける共有ストレージとキーバリューストアをを用いるようにした．当該試作の共有ストレージとして用いるための要件として，キーバリューストアでは主キーによる値の取得に加えて，全件の取得，及び値に対して二次索引を作成し当該二次索引による値の取得が可能であることを必要とする．

当該試作では，ドライバに演算資源の追加・削除を開始するためのコマンドを実装し，ユーザから当該コマンドを受け取ることによって，演算資源の追加・削除を開始する．ドライバに対し演算資源の追加・削除コマンドを発行すると，演算資源においては追加するインスタンスにおいて全ての段のワーカを起動し，演算資源の削除においては停止するインスタンスにおける実行中の全ての段のワーカを停止することによって，インスタンスを単位として演算資源の追加・削除を実行する．

当該試作においては，ドライバがユーザからクエリの実行計画を受け取ることによってクエリの実行が開始される．当該試作においてクエリの実行を開始する際には，各インスタンスおいて当該実行計画中の各段に対応するワーカが丁度 1 つずつ実行されるようにワーカを配置する．また，各インスタンスは 1 つのクエリの実行にのみ用いられるようにしたため，各インスタンスでクエリの実行計画における段数と等しいワーカが起動され，全てのインスタンスで等しい処理が実行される．インスタンスの追加を実行する際についても，実行計画中の各段に対応するワーカが 1 つずつ実行されるようにワーカを起動し，クエリの実行中にインスタンスの追加・削除を実行する際に，どのインスタンスを追加・削除しても等価な処理となるようにした．例えば，図 4.4 に示した Q1 のクエリは 3 段の実行計画となるため，Q1 の実行に用いたインスタンスにおいては，1 つのインスタンス

表 4.1: AWS（N. Virginia region）実験環境諸元

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

で起動されるワーカ数は 3 となり，Q1 実行中に新たに起動されたインスタンスについても同様に 3 つのワーカが起動される．

複数の段のワーカを起動・停止する際の順序については，本試作においては実装の簡潔さから後段のワーカから一つずつ順に起動・停止する方式を採用した．

本試作においては，インスタンスの追加・削除はユーザからの明示的なコマンドのみを契機とする．ドライバはユーザからコマンドを受け取ると当該コマンドに基いて各ワーカへ指示を行うのみであり，ドライバにおける負荷は十分に低く，ドライバの負荷によってインスタンスの追加・削除が影響を受けることはない．その為，以下の実験においてはワーカの実行に割当てるインスタンスの負荷についてのみ議論を行う．

4.2 実験環境

パブリッククラウド環境である Amazon Web Service（AWS）を用いて実験システムを構築した，演算資源として EC2 を，共有ストレージとして DynamoDB をそれぞれ用いた．実験環境の諸元を表 4.1 に示す．

評価実験用のデータセットとして，TPC-H[79] の dbgen を用いて，$ScaleFactor = 100$ および $ScaleFactor = 1000$ で生成したデータを DynamoDB のテーブルとしてロードした．TPC-H の仕様に定められた各テーブルの外部キーにおいて二次索引を作成した．

4.3 インデックス結合クエリに対する動的演算資源調整

まず，インデックス結合から構成されるクエリに対しての動的演算資源調整の評価を行った．実験に用いたクエリは，customer，orders，lineitem の 3 表の結合演算を行うク

```
SELECT o_orderkey, l_linenumber, o_totalprice
FROM customer
JOIN orders ON c_custkey = o.custkey
JOIN lineitem ON o_orderkey = l_orderkey
WHERE c_custkey BETWEEN X AND Y
```

図 4.2: 評価実験に用いたクエリ: Q1

```
SELECT l_orderkey, p_partkey, p_retailprice
FROM part
JOIN lineitem ON p_partkey = l_partkey
WHERE p_partkey BETWEEN X AND Y
```

図 4.3: 評価実験に用いたクエリ: Q2

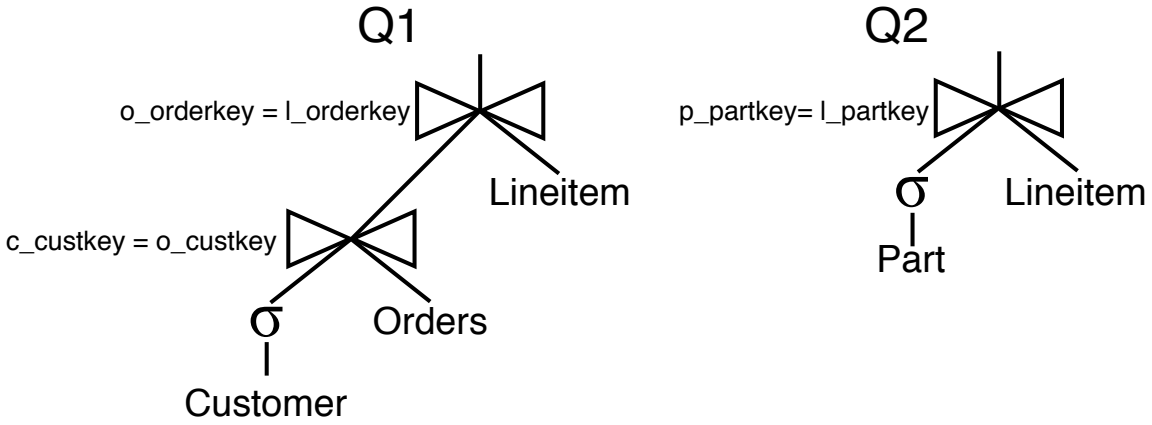
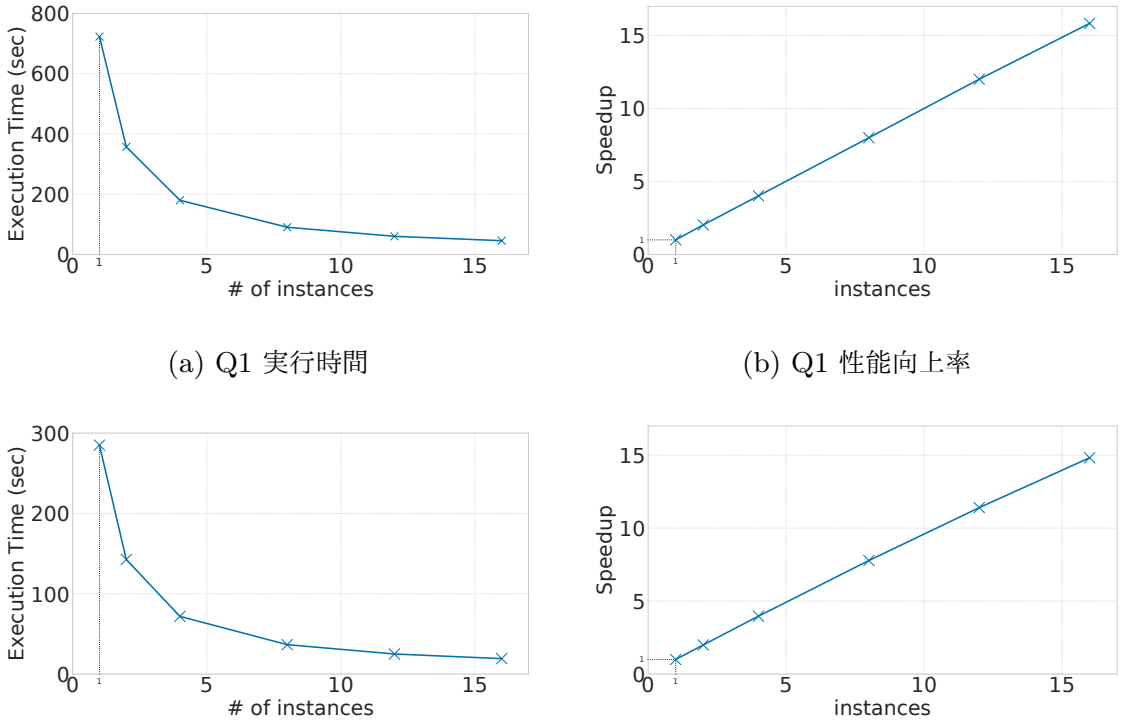


図 4.4: 評価実験に用いたクエリのプランツリー

エリ Q1（図 4.2）と，part，lineitem の 2 表の結合演算を行うクエリ Q2（図 4.3）の，2 つのクエリを用いた．Q1，Q2 のプランツリーを図 4.4 に示す．Q1 では customer テーブルの c_custkey，Q2 では part テーブルの p_partkey に対する選択を行い，それぞれ全体の 10%,5% が選択される範囲を選択条件として用いた．クエリ実行において結合を行う手法は複数存在するが，本実験で用いた選択率においては Index Join が優位となるため，Q1,Q2 の結合演算には Index Join を用いた．

4.3.1 演算資源量に対する静的スケーラビリティ

本実験では，Q1，Q2 の実行において，演算資源に対して実行速度の向上が得られることを示すために，Q1，Q2 のそれぞれについて実行に割当てられたインスタンス数に対す



(a) Q1 実行時間

(b) Q1 性能向上率

(c) Q2 実行時間

(d) Q2 性能向上率

図 4.5: クエリ Q2 の静的スケーラビリティ

る実行時間を計測した．各々のクエリの実行時間，およびインスタンス数 1 の時の実行時間を 1 とした時の性能向上率を図 4.5 に示す．インスタンス数 16 の時にインスタンス数 1 の場合と比較して，Q1 では 15.83 倍，Q2 では 14.82 倍の性能向上率が得られた．この結果から，Q1，Q2 ともにインスタンス数に応じた性能向上が得られることが確認された．

4.3.2　1 クエリでの動的演算資源量調整

本実験では，提案する動的演算資源調整手法が，クエリの実行中に割当てる演算資源を調整可能であることを示すために，Q1 の実行中において提案手法によって利用するインスタンス数の追加・削除を行い，1 秒ごとにその時点における利用インスタンス数，全てのインスタンスでの合計 CPU 使用率，クエリ全体でのストレージへの毎秒リクエスト回数 (IOPS) を計測した．インスタンス数の追加・削除操作としては，インスタンス数 1 でクエリの実行を開始し，15 秒毎に毎にインスタンス数を 2,4,8,16 に増やす操作を行った．その後，インスタンス数 16 から 15 秒ごとにインスタンス数を 8,4,2,1 まで減らす操作を

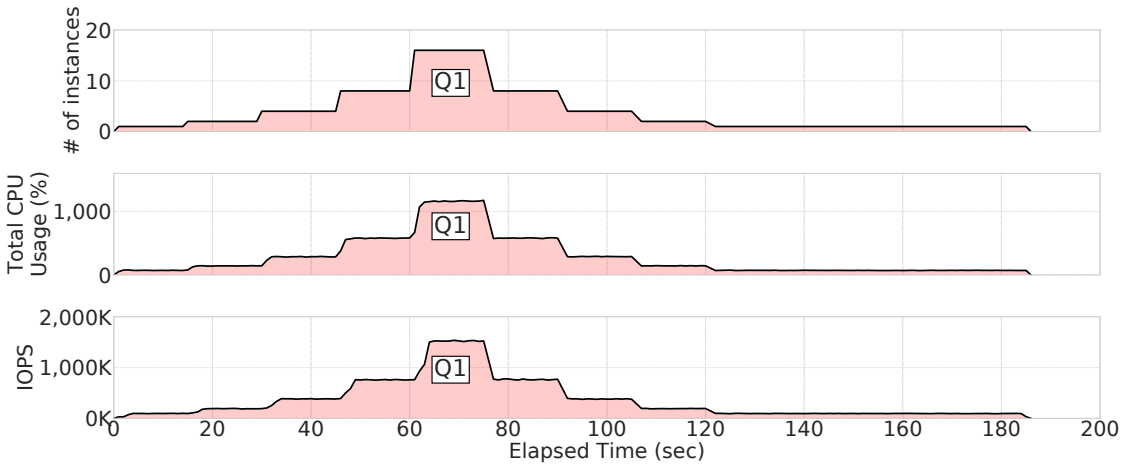


図 4.6: クエリ Q1 に対する動的演算資源調整

表 4.2: クエリ Q1 で動的演算資源調整を行った時の IOPS 性能向上率

インスタンス数	性能向上率
1	1
2	1.97
4	3.94
8	7.94
16	15.15

行った．結果を図 4.6 に示す．インスタンスの追加・削除によってクエリ実行に割当てる演算資源を調整し，それによってクエリ全体での IOPS を調整することができた．また，クエリ実行中での各インスタンス数における平均 IOPS を計算し，インスタンス数を 1 の時の平均 IOPS を 1 とした場合の，各インスタンス数における性能向上率を表 4.2 に示す．インスタンス数 8 まではほぼインスタンス数に対して線形に平均 IOPS が向上した．インスタンス数 16 では性能向上率は 15.15 倍であり，インスタンス数に対して性能向上率が僅かに下回った．また，4.2 節で示した Q1 のインスタンス数 16 における実行時間の性能向上率 15.83 倍より，本実験で得られた IOPS の性能向上率 15.15 倍が下回っているが，これは動的演算資源調整を行ってから IOPS が向上するまでのオーバーヘッドによるものと考えられる．図 4.6 ではインスタンス数を 8 から 16 に調整した際に，インスタンス数の増加から IOPS の向上まで約 2 秒を要した．本実験により，提案手法によって実行

中のクエリに割当てる演算資源の調整を行うことが可能であること，また演算資源の調整によってクエリの IOPS を調整することが可能であることを示した．

4.4 動的演算資源調整を用いたケーススタディ

本論文で提案する動的演算資源調整手法は，従来不可能であった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その実行に割当てる資源の調整を可能とするものであり，データベースエンジンにおける実際に利用状況を模擬したケーススタディによって，当該手法の有用性を明らかにする．

4.4.1 ケーススタディ 1: 余剰演算資源を用いた実行中クエリの加速

実際のデータベースエンジンにおいては，多様な優先度のクエリが同時，あるいは随時に実行を要求される．提案手法によって，異なる優先度の複数のクエリが実行されている状況においても，優先度に基づいた演算資源の調整を実行中のクエリを含めて行うことが可能となり，より優先度に応じた演算資源の割当てが可能となることをケーススタディによって示す．本章では各クエリに正の実数値として優先度を設定することが可能であるとし，複数のクエリが同時に実行されている際には，各クエリの優先度の比率が，各クエリ実行に割当てられるインスタンス数の比率と一致するように動的演算資源調整を行うものとする．

一つ目のデータベースエンジンのケーススタディとして，優先度の異なる 2 つのクエリが同時に実行を開始した場合を想定する．クエリ実行開始時に利用可能な演算資源が優先度に応じてそれぞれのクエリに割当てられ，演算資源が多く割当てられた高優先度のクエリが先に完了したとする．動的演算資源調整を行わない場合には，当該状況において高優先度クエリに割当てられていた演算資源は当該クエリの実行完了によって解放され他のクエリから利用可能となるが，既に実行を開始している低優先度クエリは実行に割当てる演算資源を変更することはできず当該演算資源を利用することはできない．

本章で提案する動的演算資源調整手法を用いることにより，高優先度のクエリが完了した時点で，新たに利用可能になった演算資源を，低優先度の実行中クエリに対して割当てることが可能となり，当該クエリの実行を加速することが期待される．

本実験では，Q1 から c_custkey に対する選択条件を異なるものとしたクエリを Q1’ として用い，Q1’ の選択率は 10% となるように設定した．Q1 および Q1’ の優先度をそれぞれ 1 と 7 に設定して，同時にクエリの実行を開始した．クエリの実行中に 1 秒ごとに，

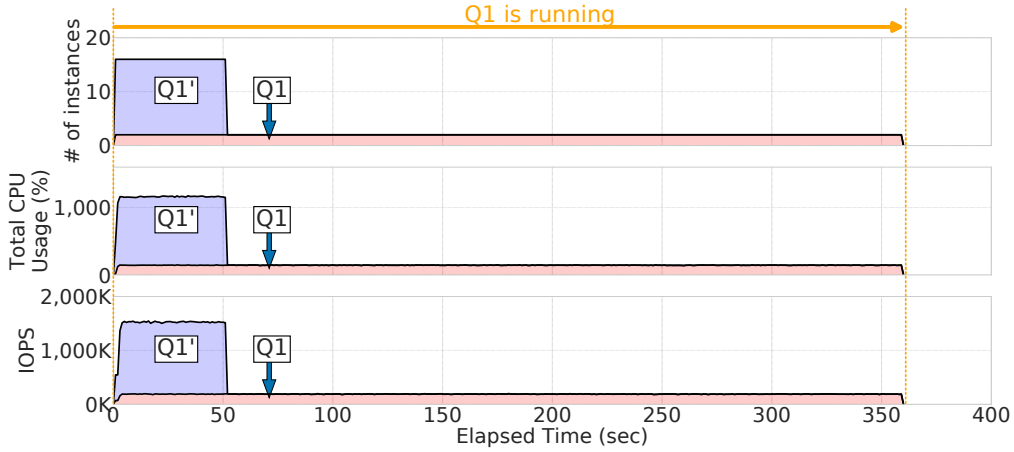


図 4.7: 余剰演算資源を用いた実行中クエリの加速: 動的演算資源調整なし

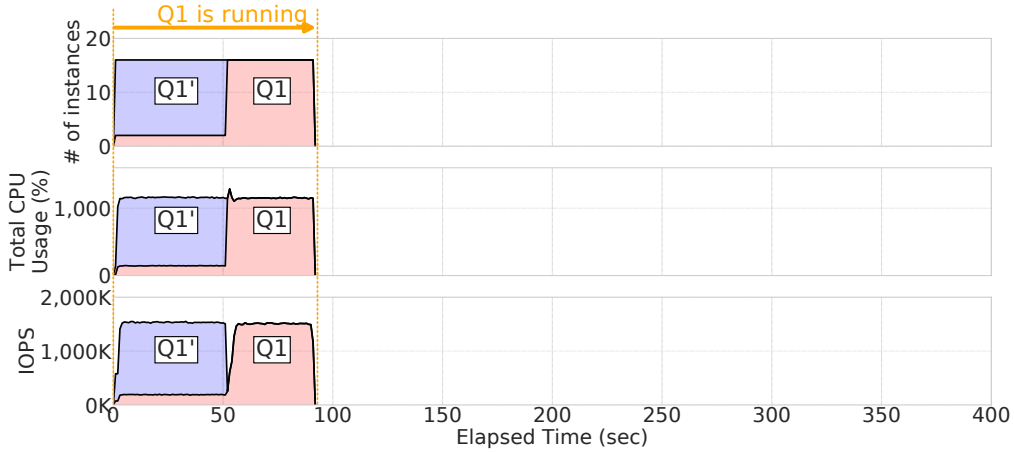


図 4.8: 余剰演算資源を用いた実行中クエリの加速: 動的演算資源調整あり

その時点での利用しているインスタンス数，全ての利用インスタンスでの合計 CPU 使用率，合計 IOPS を，Q1，Q1’ のそれぞれについて記録した．

クエリ開始時点で割当てられたインスタンス数は，クエリに設定された優先度に基づき，Q1，Q1’ のそれぞれについて 2，14 となった．動的演算資源調整を用いずクエリ完了まで同じインスタンス数を用いた場合の結果を図 4.7 に，動的演算資源調整を用いて Q1’ が完了した時点で新たに利用可能となったインスタンスを Q1 に割当てた場合の結果を図 4.8 に示す．動的演算資源調整を用いなかった場合，Q1 はクエリが完了するまで実行開始時点のインスタンス数のみを利用し，クエリ完了まで 360 秒要した．それに対し，動的演算資源調整を用いた場合，Q1 が完了すると実行されているクエリは Q1’ のみとなり動

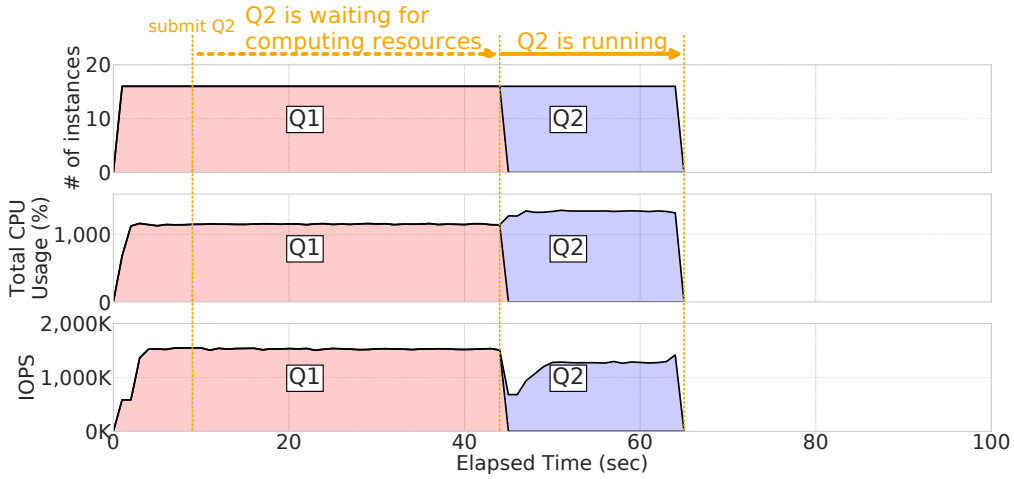


図 4.9: 高優先度クエリの割り込み: 動的演算資源調整なし

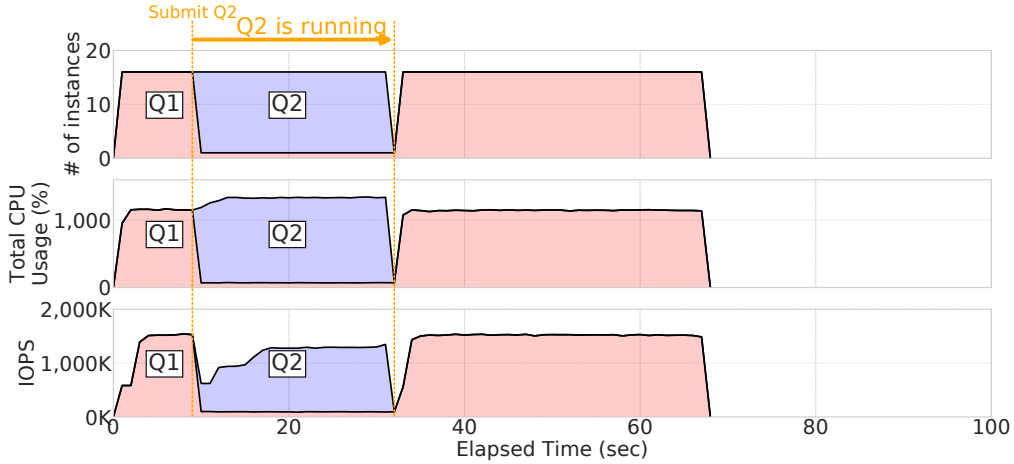


図 4.10: 高優先度クエリの割り込み: 動的演算資源調整あり

的演算資源調整によって Q1' に全てのインスタンスが割当てられるため，Q1 が完了した 52 秒の時点で，動的演算資源調整によって Q1' に 16 インスタンスが割当てられた．Q1' の実行は 92 秒で完了し，提案手法を用いることによって Q1 の実行時間は 268 秒短縮された．本実験によって，クエリの実行中に新たに演算資源が利用可能となった場合に，提案手法を用いて当該演算資源を実行中クエリに対して割当てることにより，当該実行中クエリを加速することが可能であることが確認された．

4.4.2　ケーススタディ 2: 高優先度クエリの割り込み

前節で示したものとは異なるケーススタディとして，利用可能な演算資源がない状態で高優先度のクエリの実行が要求された場合を想定する．この状況において，動的演算資源調整を行わない場合には，実行中のいずれかのクエリが完了し新たに演算資源が利用可能になるまで高優先度のクエリの実行開始が遅延される．本章で提案する動的演算資源調整手法を用いた場合，低優先度のクエリの利用演算資源を減少させ，それによって解放された演算資源を利用することで，高優先度のクエリを即座に開始することが可能となる．これにより，高優先度クエリは他のクエリの完了を待つことなく実行を開始し，従来より実行の要求から短時間で当該クエリの実行が完了することが期待される．

本実験では，Q1 を優先度 1 に設定して実行を開始し，Q1 実行開始の 10 秒後に優先度を 15 に設定した Q2 の実行を要求した．クエリの実行中に，1 秒ごとにその時点での利用しているインスタンス数，全ての利用インスタンスでの合計 CPU 使用率，合計 IOPS を，Q1，Q2 のそれぞれについて記録した．提案手法を用いなかった場合の結果を図 4.9 に，提案手法を用いて，Q2 のクエリ実行開始を要求した時に Q1 から Q2 への演算資源の割当てを行い Q2 を即座に実行した場合の結果を図 4.10 に示す．Q1 開始時点では他に実行されているクエリは存在しないため，Q1 は 16 インスタンスを割当てられて実行を開始した．提案手法を用いなかった場合，10 秒の時点で Q2 の実行を要求されたにも関わらず，Q1 が完了する 45 秒時点まで Q2 の開始が遅延されるため，Q2 が完了するのは Q2 の実行を要求してから 55 秒後となった．提案手法を用いた場合，Q2 の実行を要求すると，Q1，Q2 の割当てインスタンス数はそれぞれのクエリに設定された優先度に基づいて調整され，Q1，Q2 の割当てインスタンス数はそれぞれ 1，15 となった．これによって Q2 は実行を即座に開始することが可能となり，Q2 の実行は 22 秒で完了した．即ち，提案手法を用いることで，Q2 の実行を要求してから完了するまでの時間を 33 秒短縮することができた．一方，両方のクエリが完了するまでに要した時間は，提案手法を用いた場合，用いなかった場合と比べて 4 秒遅くなっていた．これは提案手法によって演算資源の調整を行った際に，割当てられた演算資源によってクエリの IOPS が増減するまでに遅延が存在するために，演算資源の調整を行うことで低優先度のクエリの実行に遅れが生じたものと考えられる．

本実験によって，提案手法を用いることにより演算資源の空きがない状況においても高優先度のクエリを割り込んで開始することができ，実行の要求から完了までの時間を短縮する効果が得られることを明らかにした．

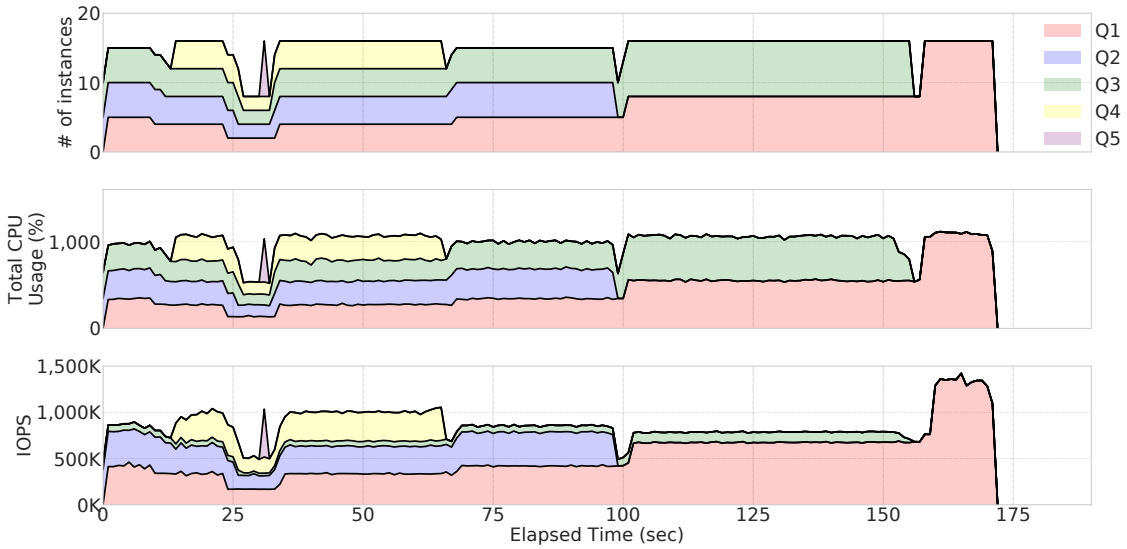


図 4.11: 多数のクエリでの動的演算資源調整

4.4.3 ケーススタディ 3: 多数のクエリでの動的演算資源調整

より実環境に近いケーススタディとして，異なる優先度を持つ多数のクエリが同時，あるいは随時実行される状況を想定し，当該状況においても，提案手法によって実行中クエリに割当てる演算資源を調整し，例えば優先度の高いクエリは多くの資源を割当てて即座に実行を開始しつつも，高優先度クエリの実行が完了し当該クエリの実行に割当てられていた演算資源が他のクエリに割当て可能になり次第，優先度に基づき各クエリへの演算資源の割当てを再度調整することが可能になるなど，よりクエリ毎の優先度に基づいた演算資源の調整が可能となることを示す．

ケーススタディ 3 では，第 4.1 節で示した Q1，Q2 に加えて，σ(orders) $\bowtie$ lineitem を実行する Q3，σ(customer) $\bowtie$ orders $\bowtie$ lineitem $\bowtie$ supplier を実行する Q4，σ(supplier) $\bowtie$ partsupp を実行する Q5 の 5 クエリを用い，Q3，Q4，Q5 の選択率は 1% となるように設定した．まず Q1，Q2，Q3 を優先度 1 に設定して実行を開始し，開始から 10 秒後に優先度 1 で Q4 の実行を要求した．さらに Q4 開始から 10 秒後に Q5 を優先度 4 に設定して実行を要求した．クエリの実行中に，1 秒ごとにその時点での利用しているインスタンス数，全ての利用インスタンスでの合計 CPU 使用率，合計 IOPS を，各クエリについて記録した．

結果を図 4.11 に示す．Q1，Q2，Q3 に設定された優先度は全て 1 であるため，各クエ

リは 5 インスタンスずつ割当てられて実行を開始した．Q4 の実行要求時には，Q4 に設定された優先度は Q1，Q2，Q3 と同じく 1 であるため，提案手法によって Q1，Q2，Q3 に割当てられたインスタンス数が 4 になるように調整され，Q4 は 4 インスタンスを割当てられて実行を開始した．Q5 の実行要求時には，Q5 に設定された優先度は 4 であるため，Q1，Q2，Q3，Q4 の割当てインスタンス数が 2 に調整され，Q5 は 8 インスタンスを割当てられて実行を開始した．その後，Q5，Q4，Q2，Q3，Q1 の順にクエリが完了し，各クエリの完了時点において，残りのクエリの割当てインスタンス数の比率が各クエリの優先度の比率と等しくなるように調整された．Q5 は他のクエリと比べて高い優先度を設定されていたが，提案手法によって Q5 の実行を優先度に応じた演算資源を割当てて開始することができた．また，IOPS についても，クエリ毎に演算資源あたりの得られる IOPS に差があるために全クエリでの合計 IOPS は一定とならないもの，クエリ毎の IOPS については割当てられた演算資源に応じた値を得ることができた．

しかしながら，いくつかの時間帯において，クエリに割当て可能であるがどのクエリにも割当てられていないインスタンスが存在した．例えば，高優先度に設定された Q5 について，Q5 の実行要求時点における実行中クエリの割当て演算資源を調整することによって，Q5 の実行を多くの資源を割当てつつ開始することができたものの，実行中クエリに対する演算資源調整に 5 秒程度の時間を要したために，Q5 の実行要求から開始までに 7 秒程度の遅延を要した．当該遅延が発生している間に，クエリに割当て可能なインスタンスで，どのクエリにも割当てられていないものが最大 8 インスタンス存在した．演算資源の調整に伴う遅延を削減することによって，提案手法による資源調整はより効率よく資源を利用可能になるものと考えられる．本実験によって，優先度の異なる多数のクエリが実行される状況においても，提案手法によって実行中のクエリの割当て演算資源をクエリの優先度に基いて調整可能であることを示した．

4.4.4 ケーススタディ 4: 資源解放によるインスタンス利用コスト削減

これまでのケーススタディは全体での利用可能資源量が一定であるという状況を想定していたが，クラウド環境は迅速な資源の確保・解放が可能であるという特徴を有しており，当該特徴をデータベースエンジンにおける動的演算資源伸縮に活用するためには，利用可能な資源量に変動がある場合に，各クエリの実行に割当てる資源の調整を可能とすることが求められる．

提案する動的演算資源調整は利用可能資源量に変動がある場合においても資源調整が可能であることをケーススタディによって示す．本ケーススタディでは，利用可能なインス

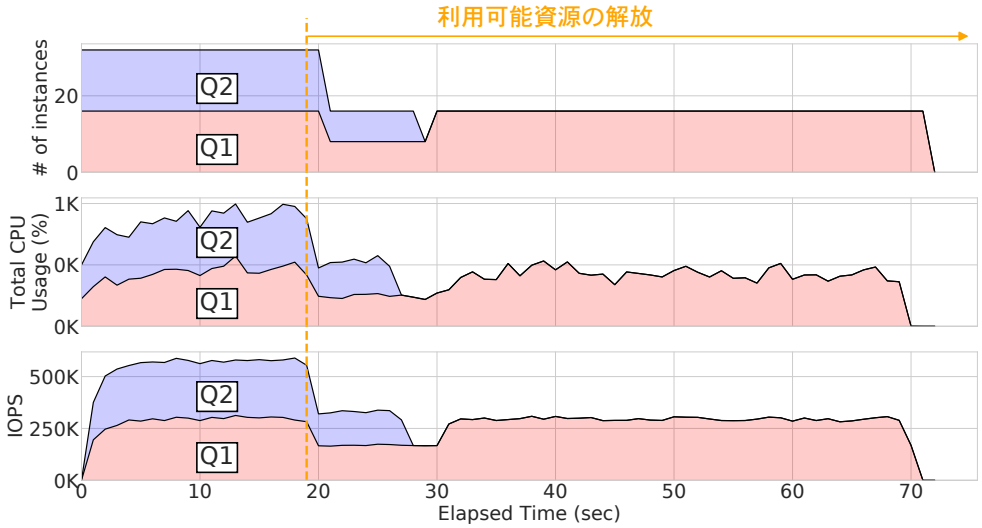


図 4.12: 資源解放によるインスタンス利用コストの削減

タンス数が 32 である状況において，Q1 と Q2 をそれぞれ優先度 1 を設定して実行開始し，クエリの実行中に利用可能資源を解放して 16 まで減らした．クエリの実行中に，1 秒ごとにその時点での利用しているインスタンス数，全ての利用インスタンスでの合計 CPU 使用率，合計 IOPS を，各クエリについて記録した．

結果を図 4.12 に示す．クエリ実行開始時における Q1，Q2 への割当てインスタンス数は，優先度に従ってそれぞれ 16 となった．利用可能資源解放後には，Q1，Q2 へのインスタンスの割当ては優先度に従ってそれぞれ 8 となった．Q2 完了後には，Q2 に割当てられていたインスタンスが Q1 へと割当てられて，Q1 の割当てインスタンス数は 16 となった．本実験により，クエリの実行中に利用可能資源の量が減少する状況においても，提案手法によって優先度に基づいた資源割当てが可能であることを示した．

4.4.5 ケーススタディ 5: 資源確保による需要増への対応

本ケーススタディでは，一時的に資源需要が増加する状況において，新たに利用可能資源を確保し提案手法によって当該資源の割当てを行うことによって，一時的な需要増に即応可能であることを示す．

本ケーススタディでは利用可能インスタンス数が 16 である状況において Q1 を優先度 1 に設定して開始した．Q1 開始の 10 秒後に Q2 を優先度 1 で開始した．Q2 の開始十秒後に資源の追加確保を行い，利用可能インスタンス数を 32 とした．Q2 の実行完了後に

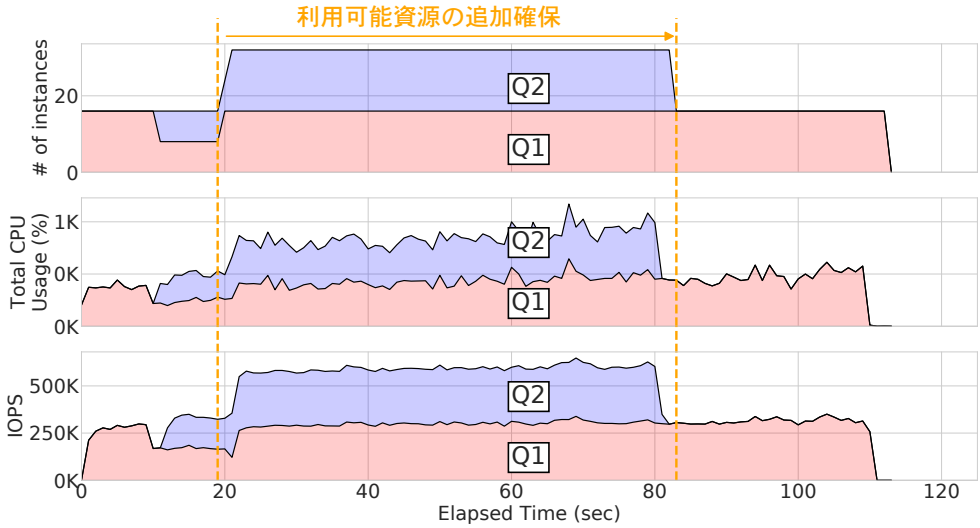


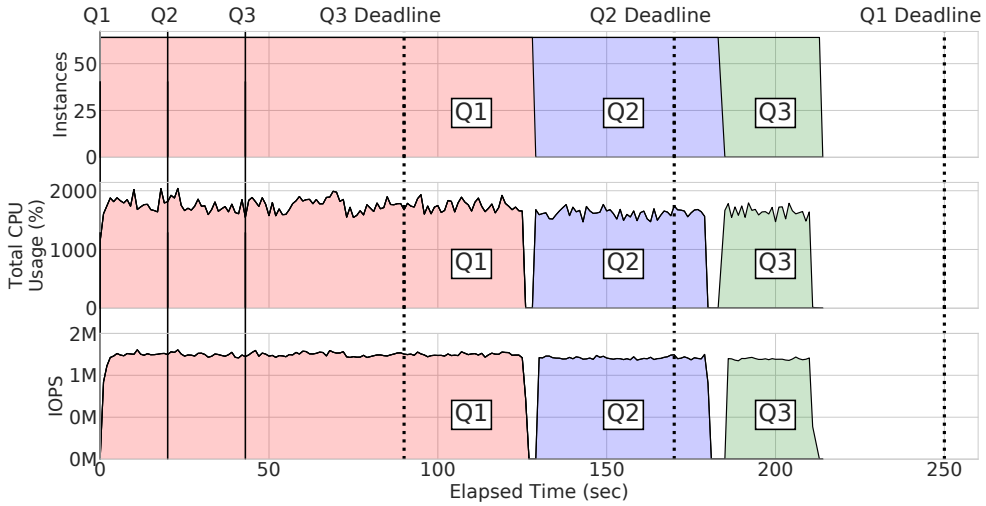
図 4.13: 資源確保による需要増への対応

資源の解放を行い，利用可能資源を 16 とした．クエリの実行中に，1 秒ごとにその時点での利用しているインスタンス数，全ての利用インスタンスでの合計 CPU 使用率，合計 IOPS を，各クエリについて記録した．

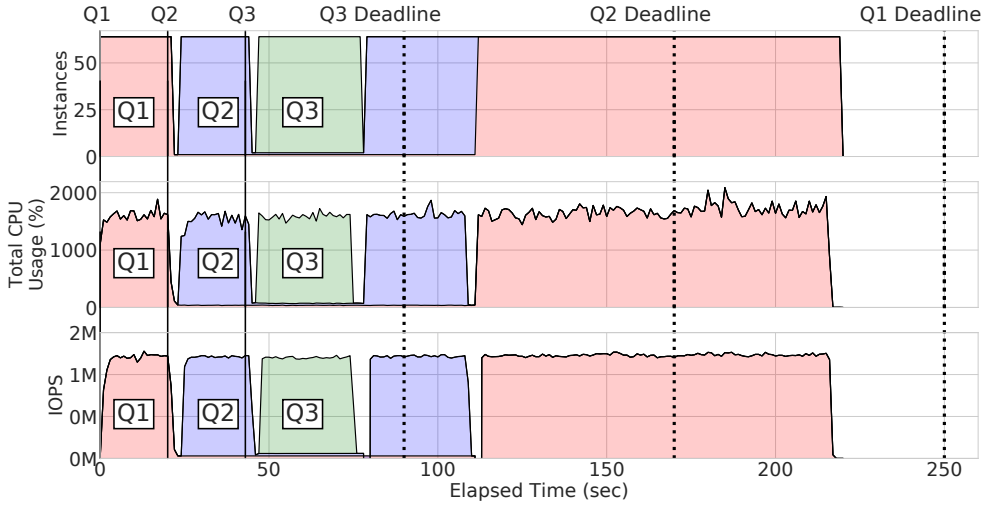
結果を図 4.13 に示す．Q1 は割当てインスタンス数 16 で開始し，Q2 の実行開始時には優先度に基づく割込みが行われ，Q1，Q2 の割当てインスタンス数はそれぞれ 8 となった．資源の追加確保実行後には Q1，Q2 の割当てインスタンス数はそれぞれ 8 となった．本実験により，一時的にデータベースエンジン全体での資源要求が増加した状況においても，利用可能資源の追加確保を行い当該資源を提案手法によって割当てることによって，一時的な需要増に即応が可能となることを明らかにした．

4.4.6 ケーススタディ 6: 動的演算資源調整によるデッドラインを考慮した資源割当て

これまでのケーススタディでは各クエリごとに設定された優先度に基づき，各クエリへの割当てインスタンス数の比がクエリの優先度の比と等しくなるように動的演算資源調整を行った．各クエリへの割当て資源量の決定には，各々のクエリの優先度の他，当該クエリに設定されたデッドライン，即ちユーザが当該クエリの完了時点において超過することを望まない時刻に基づいた方法が考えられる．本ケーススタディでは，本論文によって提案する動的演算資源調整によって，クエリの実行が要求された時に，当該クエリのデッド



(a) 動的演算資源調整なし



(b) 動的演算資源調整あり

図 4.14: デッドラインを考慮した動的演算資源調整

ラインまでの時間が実行中のクエリのデッドラインまでの時間よりも短い場合には，本論文において提案する動的演算資源調整によって当該新規クエリに可能な限りの多数の資源を割当てることによって，各クエリでのデッドラインの超過を回避することが可能となることを示す．

本ケーススタディでは，Q1，Q2，Q3 のそれぞれについて，0 秒，20 秒，40 秒時点に

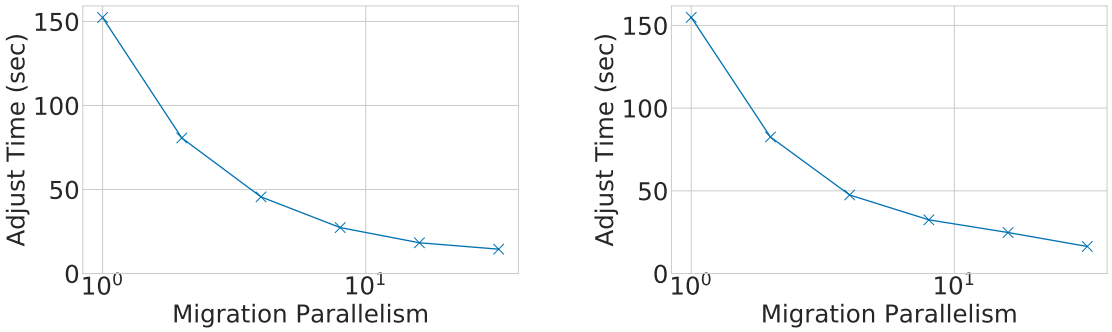
おいて実行要求し，各々のクエリのデッドラインを 250 秒，170 秒，90 秒時点にに設定した．動的演算資源調整を行わない場合には，各々のクエリを要求が行われた順に実行した．動的演算資源調整を行う場合には，クエリの実行要求が行われた際に，当該クエリのデッドラインまでの時間が，実行中クエリのデッドラインまでの時間よりも短い場合には，当該新規クエリに可能な限りの多数のインスタンスを割当てて実行を開始した．クエリの実行中に各クエリへの割当てインスタンス数と IOPS を 1 秒ごとに計測した．得られた結果を図 4.14 に示す．動的演算資源調整を行わない場合には，Q1，Q2，Q3 の順でクエリが実行され，Q2 と Q3 でデッドラインの超過がが発生した．デッドラインの超過時間は Q2，Q3 のそれぞれについて 14 秒，124 秒であった．動的演算資源調整を行う場合には，Q2 の実行が要求された際には，Q2 のデッドラインまでの時間がが Q1 のデッドラインの時間よりも短いため，Q1 の割当てインスタンス数は 1 まで減らされ，Q2 は 63 インスタンスを割当てて実行が開始された．Q3 の実行開始時も同様に 62 インスタンスを割当てて実行が開始された．Q3 の完了時には，Q3 に割当てられていたインスタンスは，次にデッドラインまでの時間が短い Q2 に割当てられたために，Q3 完了後の Q2 の割当てインスタンス数は 63 となり，Q2 の完了後も同様に Q1 の割当てインスタンス数が 64 となった．動的演算資源伸縮を用いた資源割当てにおいては，全てのクエリをデッドラインまでに完了することができ，クエリのデッドラインの合計時間は 0 秒であった．本実験によって，複数のクエリが実行されている状況において，本論文において提案する動的演算資源調整手法を用いて各々のクエリのデッドラインを考慮した資源割当てを行うことによって，各々のクエリのデッドライン超過時間を削減することが可能であることを明らかにした．

4.5　ハッシュ結合クエリに対する動的演算資源調整

前節では，インデックス結合から構成されるクエリに対する動的演算資源調整についての評価実験を示した．本節では，ハッシュ結合から構成されるクエリを対象とする．

4.5.1　バケット移送の並列化による性能向上

第 3 章において論じたように，ハッシュ結合の状態移送はインスタンス間においてハッシュテーブルのバケットを移送することによって行われるが，演算資源の追加においては実行中インスタンスから追加インスタンスへ，演算資源調整の削除においては停止インスタンスから残余インスタンスへと状態を移送することとなり，状態移送後の各インスタン



(a) 実行時間　　(b) 性能向上率

図 4.15: 状態移送の並列度と実行時間

スの負荷を均衡化するためには，当該状態移送を多数のインスタンスの組で実行する必要があるため，複数のインスタンスの組で並列して状態移送を実行することによって，演算資源調整の実行時間を高速化することが期待される．32 インスタンスを割当てて実行中のハッシュ結合クエリに 32 インスタンスを追加する，ならびに 64 インスタンスを割当てて実行中のハッシュ結合クエリから 32 インスタンスを削除する演算資源調整において，状態移送の並列度を変えて資源調整の実行時間を計測した．結果を図 4.15 に示す．状態移送を並列に行うことによって資源調整の実行時間を短縮し，32 並列によって資源追加では 10.5 倍，資源削除では 9.7 倍の高速化が得られることを示した．

4.5.2　1 クエリでの動的演算資源量調整

本実験では，提案する動的演算資源調整手法によって，ハッシュ結合クエリの実行に割当てる演算資源を調整可能であることを示すために，ハッシュ結合クエリの実行中において，提案手法によって利用するインスタンス数の追加・削除を行い，1 秒ごとにその時点における利用インスタンス数，ならびに入出力スループットを計測した．インスタンス数の追加・削除操作としては，インスタンス数 8 でクエリの実行を開始し，インスタンス数を 32,64 まで増やす操作を行い，その後にインスタンス数を 32,8 まで減らす操作を行った．結果を図 4.16 に示す．提案手法によって，ハッシュ結合クエリに対してクエリ実行に割当てる演算資源を調整し，それによってクエリ全体での入出力スループットを調整可能となることが確認された．資源調整のオーバーヘッドとして，資源追加・削除ともに，資源調整から完了までに 10 秒程度の時間を要した．ハッシュ結合に対する演算資源調整

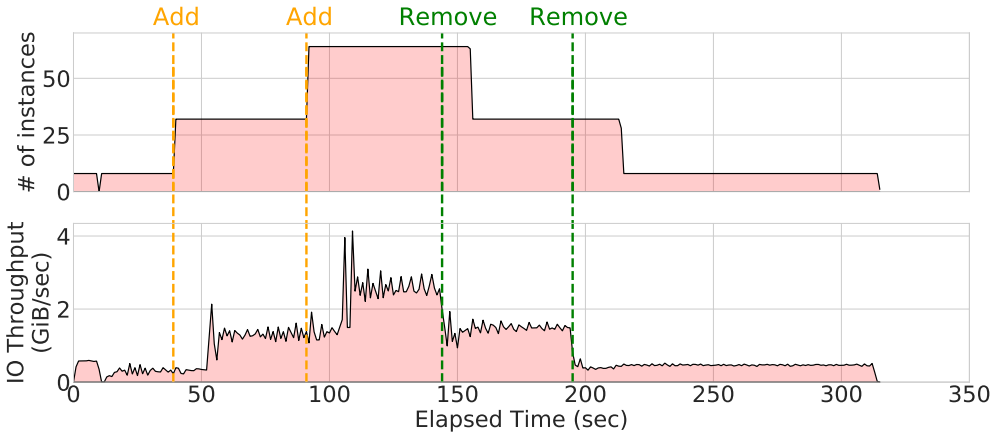


図 4.16: ハッシュ結合クエリに対する動的演算資源調整

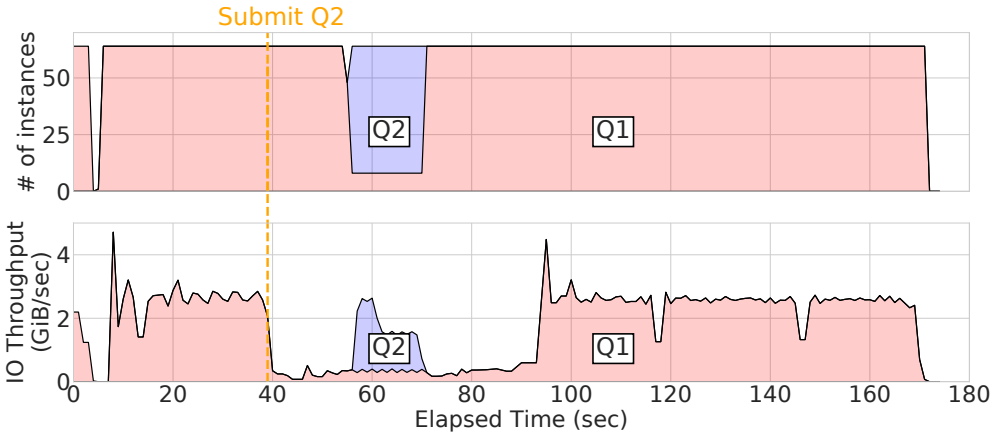


図 4.17: 異なる結合方式の複数クエリに対する資源調整

では，インデックス結合に対する演算資源調整と異なり，バケットの移送による状態移送が必要となるため，より資源調整のオーバーヘッドが大きくなったものと考えられる．本実験による，提案手法にハッシュ結合から構成されるクエリについて動的資源伸縮が可能となることを示した．

4.5.3 異なる結合方式の複数クエリに対する資源調整

並列データベースエンジンにおける結合演算の実行方式はハッシュ結合が広く用いられているが，ハッシュ結合はデータセットを全件走査する手法であるため，クエリの選択率の影響は小さい．一方，インデックス結合は，インデックススキャンを用いてネステッド

ループ結合を行う結合方式であり，インデックス結合では選択率が小さい場合に大きく実行速度が向上するため，低選択率のクエリではインデックス結合による実行方式がより高速にクエリを実行可能であることが知られている．ハッシュ結合クエリとインデックス結合クエリの両方が実行されている状況において，両実行形式のクエリに対して動的資源調整を実現することにより，より広範なクエリに対して動的資源調整が可能となったといえる．

ハッシュ結合クエリとインデックス結合クエリの両方が実行されている状況において，提案手法によって演算資源調整が可能となることを示すために，ハッシュ結合クエリの実行中に，インデックス結合クエリの割込み実行を行う評価実験を行った．本実験では選択率 50% としたハッシュ結合クエリの実行を行い，当該クエリの実行開始 40 秒後にインデックス結合クエリの開始を要求した．ハッシュ結合クエリ，インデックス結合クエリに設定する優先度はそれぞれ 1，7 とし，各々のクエリに割当てる割当てるインスタンス数を優先度の比と等しくなるようにした．クエリの実行中に，各々のクエリについて実行に割当てているインスタンス数と入出力処理のスループットを 1 秒ごとに計測した．得られた結果を図 4.17 に示す．実行中のハッシュ結合クエリの割当てインスタンスを減らすことによって，インデックス結合の割込実行が可能となることを確認した．本実験によって，ハッシュ結合とインデックス結合の両方を含む複数のクエリに対して，提案手法によって動的演算資源調整が可能となることを示した．

4.6　大規模環境における動的演算資源調整

4.6.1　演算資源量に対する静的スケーラビリティ

本実験では，256 インスタンス規模の環境におけるクエリ実行について，演算資源に対して実行速度の向上が得られることを示すために，図 4.18 に示す Q1，Q2，Q3 のそれぞれについて実行に割当てられたインスタンス数に対する実行時間ならびに平均 IOPS を計測した．各々のクエリについて，インスタンス数 8 の時の実行時間を 8 とした時の性能向上率ならびに平均 IOPS を図 4.19 に示す．インスタンス数 256 の時に，Q1 では 242.3 倍，Q2 では 229.6 倍，Q3 では 229.9 倍の性能向上が得られた．また，平均 IOPS は Q1，Q2，Q3 のそれぞれについて，8.7M，9.4M，8.6M となった．この結果から，提案するデータベースエンジンにおいて 256 インスタンスまでインスタンス数に応じた性能向上が得られることが確認された．

```
select p_brand,
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from part
join lineitem on part.p_partkey = lineitem.l_partkey
where [part selection condition]
group by p_brand
```

(a) Q1

```
select o_orderpriority,
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from customer
join orders on customer.c_custkey = orders.o_custkey
join lineitem on o_orderkey = l_orderkey
where [customer selection condition]
group by o_orderpriority
```

(b) Q2

```
select n_name,
       sum(ps_supplycost),
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from customer
join orders on customer.c_custkey = orders.o_custkey
join lineitem on o_orderkey = l_orderkey
join partsupp on l_partkey = ps_partkey and l_suppkey = ps_suppkey
join supplier on s_suppkey = ps_suppkey
join nation on s_nationkey = n_nationkey
where [customer selection condition]
group by n_name
```

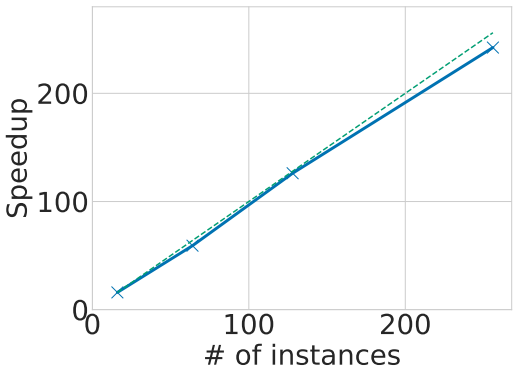
(c) Q2

図 4.18: 大規模環境における評価実験に用いたクエリ

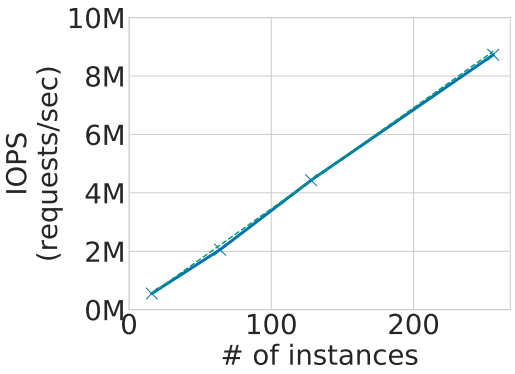
4.6.2 1 クエリでの動的演算資源量調整

本実験では，提案する動的演算資源調整手法が，256 インスタンスの規模において，実行に割当てる演算資源を調整可能であることを示すために，インデックス結合クエリの実行中において，提案手法によって利用するインスタンス数の追加・削除を行い，1 秒ごとにその時点における利用インスタンス数，ならびに入出力スループットを計測した．インスタンス数の追加・削除操作としては，インスタンス数 64 でクエリの実行を開始し，インスタンス数を 128,256 で増やす操作を行い，その後にインスタンス数を 128,64 まで減

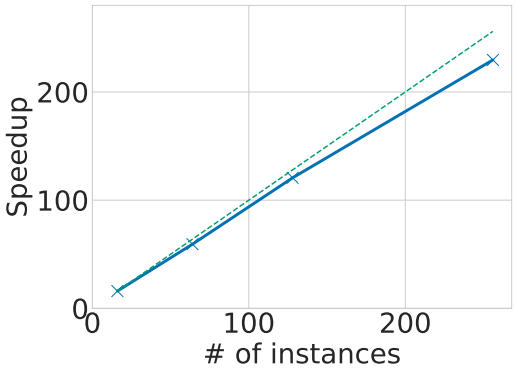
らす操作を行った．結果を図 4.21 に示す．256 インスタンスの規模においても演算資源を調整し，それによってクエリ全体での IOPS を調整可能となることが確認された．256 インスタンスを割当てている間は平均 7.8 IOPS の処理性能が得られた．演算資源の追加は数秒で完了したものの，演算資源の削除については，256 インスタンスから 128 インスタンスへの削除の完了までに 20 秒程度の時間を要した．これは，資源削除の際に，前段から停止インスタンスへと送信済みであるデータの処理が完了するまで待つ必要があるため，演算資源削除の対象インスタンス数が増えるに従って，演算資源削除の実行時間が延びることの起因するものと考えられる．本実験によって，提案手法によって 256 インスタンスの規模における動的演算資源調整が可能となることを示した．



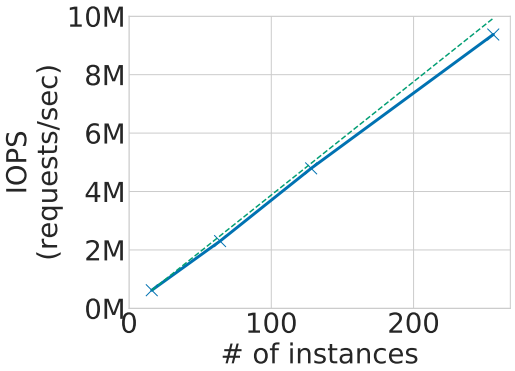
(a) Q1 性能向上率



(b) Q1 IOPS

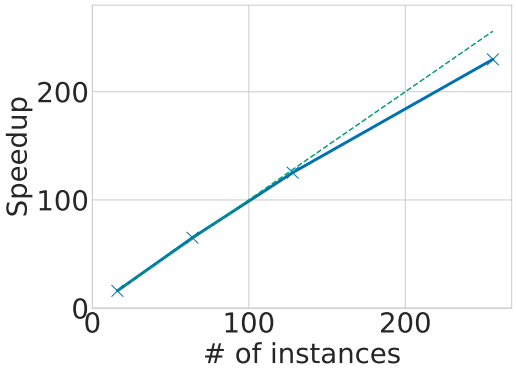


(c) Q2 性能向上率

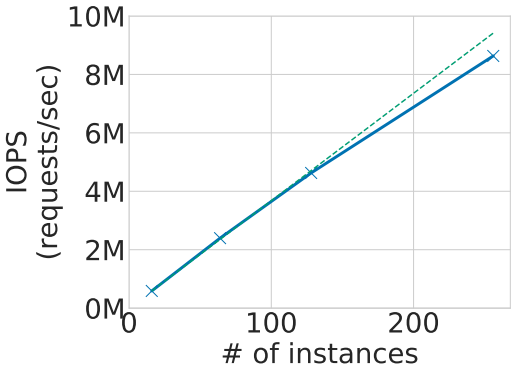


(d) Q2 IOPS

図 4.19: クエリ Q2 の静的スケーラビリティ



(a) Q3 性能向上率



(b) Q3 IOPS

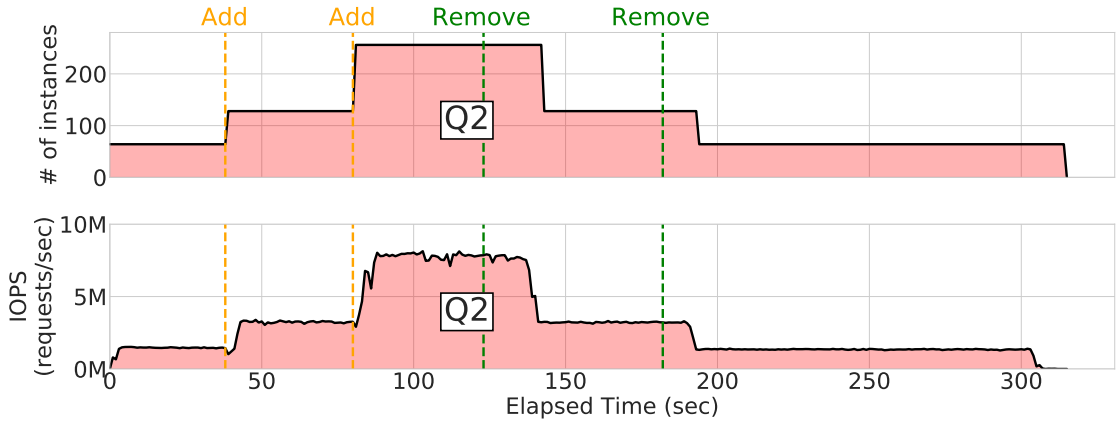


図 4.21: ハッシュ結合クエリに対する動的演算資源調整

第 5 章

共有ストレージ型並列データベースエンジンにおける動的負荷分散手法

並列データベースエンジンは複数のプロセッサを用いて演算を並列化することによってクエリ実行を高速化するデータベースエンジンであり，学術界，産業界の双方から多くの提案がなされている [18, 51, 52, 54]．並列データベースエンジンにおいてクエリ実行に用いているプロセッサ間の負荷に偏りが生じた場合，高負荷のプロセッサによってクエリ全体の実行が律速されて低負荷のプロセッサの利用効率が低下し，結果としてクエリの実行時間が長くなることが知られており，プロセッサ間の負荷を均衡化して偏りの影響を軽減する負荷分散の研究が多くなされている [37–42]．

従来の並列データベースエンジンの多くは，処理能力の等しい均質なプロセッサを用いてクエリが実行されることを前提としており，各プロセッサの負荷が等しい場合には各プロセッサにおける処理時間が等しくなることを想定する．クエリ実行に用いるプロセッサの処理能力が等しくない場合，各プロセッサに等しい負荷を割り当てると，低い能力のプロセッサの処理時間が相対的に長くなり，クエリ全体の実行速度が当該プロセッサの処理速度によって律速され，高性能のプロセッサの利用効率が低下する．プロセッサの非均質性に起因するプロセッサの利用効率への影響を軽減するためには，各プロセッサの能力に応じて負荷を分散し，各プロセッサにおける処理時間を均衡化する手法が必要となる．

ユーザがネットワークを介して仮想化された演算・記憶資源を利用するパブリッククラウド環境は，Amazon，Google などの事業者によって一般に提供されており，ユーザが自らハードウェアを所有しデータセンタを構築するオンプレミス環境に加えて，近年広く利用されるようになってきている．パブリッククラウド環境の多くは，事業者が所有するデータセンタに収められたサーバを仮想化しインスタンスとして提供しているが，実際に

は事業者がユーザに提供可能なインスタンスには限りがある．その為，ユーザは当該ユーザが要求した能力を持つインスタンスを常に確保できるとは限らず，当該状況においてはユーザは異なる能力のインスタンスを利用する必要がある．また，パブリッククラウド環境が提供するインスタンスは仮想化されたものである為，同じ能力とされるインスタンスにおいても実際に有する能力は異なる可能性がある [5–7]．パブリッククラウド環境のインスタンスを利用する際には，インスタンス毎のプロセッサの処理能力が異なる，即ち非均質な演算資源を用いる状況を考慮する必要があり，均質な演算資源を前提とする並列データベースエンジンではパブリッククラウド環境の演算資源を効率よく利用することはできない．

並列データベースエンジンにおける結合演算の実行方式はハッシュ結合が広く用いられているが，ハッシュ結合はデータセットを全件走査する手法であるため，クエリの選択率の影響は小さい．一方，インデックス結合は，インデックススキャンを用いてネステッドループ結合を行う結合方式であり，インデックス結合では選択率が小さい場合に大きく実行速度が向上するため，低選択率のクエリではインデックス結合による実行方式がより高速にクエリを実行可能であることが知られている [8]．従来のインデックス結合はプロセッサの利用効率が低く，プロセッサ間の処理能力の偏りによる影響が小さいために，並列データベースエンジンにおける負荷分散の対象としては主にハッシュ結合が扱われてきた．しかし，アウトオブオーダ型実行方式 [77, 78] を用いることによって，インデックス結合におけるプロセッサ利用効率を大幅に向上させることが可能であり，当該実行方式を用いた場合はインデックス結合においてもプロセッサ処理能力の偏りの影響を大きく受けることとなる．

並列データベースエンジンにおいて演算資源の非均質性の影響を受けてプロセッサの利用効率が低下するという問題に対しては，これまで主にハッシュ結合に対して負荷分散によって当該影響を軽減する研究が行われてきた [45]．しかしながら，インデックス結合，とりわけアウトオブオーダ実行方式を用いた場合の演算資源非均質性の影響について取り組んだ研究は，著者らの知る限り見当たらない．本論文は，ハッシュ結合とインデックス結合の両結合方式に対して負荷分散手法を示すことにより，広範な選択率のクエリに対して優位な結合方式を選択しつつ，演算資源非均質性の影響を軽減しより高い性能向上が得られることを示す点が特徴的である．

5.1 共有ストレージ型並列データベースエンジンにおけるプロセッサ間の処理時間の偏り

並列データベースエンジンは複数のプロセッサを用いてクエリ実行における演算を並列化する．並列データベースエンジンでのクエリ実行においてプロセッサ間の処理時間に偏りが発生した場合，最も遅いプロセッサによってクエリ全体の実行時間が律速される．クエリ実行中にプロセッサ間の処理時間の偏りが生じる原因としては様々なものが議論されている，並列データベースエンジンにおいて各プロセッサが一つのストレージを共有するアーキテクチャである共有ストレージ型においては，プロセッサの能力の偏り，データの分布の偏りの二つが主な原因として挙げられる [37].

5.1.1 プロセッサの能力の偏り

プロセッサの能力の偏りについて，並列データベースエンジンの多くは，各プロセッサの能力が等しいことを前提としており，各プロセッサに等しい負荷を与えた場合には等しい処理時間となることを期待する．クエリの実行に割り当てたプロセッサの能力に偏りが存在する場合，各プロセッサに対して等しい量の負荷を与えたとしても，各プロセッサの処理時間は異なることとなる．並列データベースエンジンを用いたシステムを運用する際には，多くの場合，等しい能力を持つプロセッサを用いるが，システムの運用中に新たにノードを追加する場合や故障したノードを交換する場合などに，従来と同じ能力のプロセッサを用意できずに，結果として複数の異なる能力を持つプロセッサが混在する状況になることが考えられる．また，近年利用が広まっているパブリッククラウド環境を用いて並列データベースエンジンのシステムを運用する場合，等しいな能力のプロセッサを持つ均質なインスタンスのみではユーザが要求した量を確保できず，複数の能力のプロセッサを持つインスタンスを組み合わせなければならない状況が起こりうる．また，仮想化されたサーバを提供するパブリッククラウド環境では，等しい能力のプロセッサを持つとされるインスタンスにおいても，実際に得られる能力が異なる場合があることが知られている [5–7].

5.1.2　データの分布の偏り

次に，データの分布の偏り，即ちテーブル内の各列の値の分布が一様でなく偏りがある場合について議論する．並列データベースエンジンにおいては，ノード間でネットワークを介してタプルを分配する際に，タプルの値に基づいて当該タプルの送信先を決定する操作が必要となる場合がある．例として，並列データベースエンジンにおけるハッシュ結合では，各タプルにおける結合キーのハッシュ値に基づいて，当該タプルの送信先を決定し各ノードに分配する操作が行われる．当該操作において，結合キーの値の分布に偏りが存在すると，特定のノードに多くのタプルが送信され，当該ノードのプロセッサの負荷が高まることとなる．さらに，データの分布に偏りが存在すると，結合処理を行う際のタプル毎の結合率にも偏りが生じる．即ち，ハッシュ結合において，ビルド側テーブルの 1 タプルに対して結合するプローブ側テーブルのタプル数が結合キー毎に異なることとなる．結合率の高いタプルを含むバケットを割当てられたノードにおいて，多くのプローブ側タプルの結合処理を行う必要がある為，当該ノードのプロセッサの負荷が高くなる．これらの原因によって特定のノードのプロセッサに負荷が偏ると，当該プロセッサにおける処理時間が延びて，プロセッサ間の処理時間に偏りが生じる．データの分布の偏りに起因する負荷の偏りが発生するクエリの具体例としては，少数の特定の商品が特に多く売れた状況において，商品マスタテーブルと購入履歴テーブルを結合して顧客毎の売上を計算するクエリなどが考えられる．並列データベースエンジンにおける結合の実行方式としては，ハッシュ結合の他にインデックス結合が挙げられるが，インデックス結合ではタプルの送信先は任意のノードを選択してよいため，データの分布の偏りがある場合においてもノード毎の送信タプル数を等しくすることは容易であり，データの分布の偏りの影響は受けにくい．

本論文では，本章で述べたプロセッサの能力の偏り，及びデータの分布の偏りの両方を対象とし，当該偏りに起因するプロセッサ間の処理時間を均衡化する手法を示す．

5.2　共有ストレージ型データベースエンジンにおける動的負荷分散手法

前章で議論した原因によってプロセッサ間の処理時間に偏りが生じた場合，最も遅いプロセッサの処理時間にクエリ全体の実行時間が律速されて，全体としてプロセッサの利用効率が落ちる．プロセッサ間の処理時間の偏りによるクエリ実行時間への影響は，プロ

セッサ間の処理時間がが均衡化されるように負荷を分散することによって軽減が可能である．負荷分散には，クエリの実行前に得られる情報のみを利用して各プロセッサの処理時間が等しくなるように実行プランを作成する静的な手法と，クエリの実行中に得られた情報を利用して各プロセッサの処理時間を均衡化する動的な手法が考えられる．クエリ実行中の結合演算におけるデータの分布は，当該演算の前に実行される選択演算での条件によって容易に変わりうるため，結合演算において各プロセッサにどの程度の負荷の偏りが生じるかをクエリ実行前に予測することは困難である．また，CPU のコア数や周波数などのプロセッサの仕様を事前に得ることは可能だが，クエリ実行において各プロセッサが当該仕様によりどの程度の処理時間となるかを予測することは難しい．即ち，より広範な環境およびクエリに対してプロセッサ間の処理時間の均衡化を行うためには，実行時に得られる情報を用いて各プロセッサに負荷を分散し，各プロセッサの処理時間を均衡化する動的な負荷分散手法が必要となる．本章では並列データベースエンジンの結合方式として広く用いられているハッシュ結合に加えて，低い選択率においてハッシュ結合より優位となるインデックス結合の，2 つの結合方式を含むクエリについて動的負荷分散手法を示す．

5.2.1　ハッシュ結合における動的負荷分散

並列データベースエンジンにおけるハッシュ結合は，一方のテーブルのタプルを結合キー値に基づいてタプルを各ノードへと分配し，各ノードにおいてタプルの結合キー値をキーとしたハッシュテーブルを構築するビルドフェーズと，もう一方のテーブルのタプルを結合キー値に基づいて各ノードへと分配し，各ノードにおいてタプルの結合キー値によってハッシュテーブルを検索し当該処理によって得られたタプルとの結合処理を行うプローブフェーズによって行われる．本節では各フェーズにおいてハッシュテーブルのバケットの粒度での負荷モデルを示し，高負荷のノードから低負荷ノードへとバケットを移行する手法 [26, 46, 47] を用いて，負荷の均衡化を行う手法を示す．

■ビルドフェーズにおける負荷モデル　ハッシュ結合のビルドフェーズにおける負荷モデルについて述べる．ビルドフェーズでは，各ノードにおいてビルド側のテーブルをフルスキャンし，得られたタプルを結合キー値に基づいて各ノードへと分配した後に，各ノードにおいてタプルの結合キー値をキーとしたハッシュテーブルの構築を行う．

各ノードにおける，テーブルのフルスキャンおよびタプルの送信処理について，1. フルスキャンにおいて処理中のページから次のタプルを取得（処理中のページにタプルが残っていない場合には入出力によって次ページを取得），2. 得られたタプルから送信先の決定，

3. タプルの送信の処理を行う．ノード i において，入出力によるフルスキャンの次ページの取得に要する CPU 時間を $C_{scan,i}$，1 タプルの取得，送信先決定，タプル送信を行うのに要する CPU 時間を $C_{scansend}$ とする．ある時間内におけるフルスキャンのページ取得回数，タプルの送信回数をそれぞれ $\Delta N_{scan,i}$，$\Delta N_{scansend,i}$ とすると，当該時間内における合計の CPU 時間 $\Delta CPU_{scan,i}$ は，次式で表すことができる．

$$\begin{aligned}\Delta CPU_{scan,i} = \Delta N_{scan,i} C_{scan,i} + \\ \Delta N_{scansend,i} C_{scansend,i}\end{aligned} \tag{5.1}$$

ハッシュ結合のビルドフェーズにおけるハッシュテーブルの構築では，ビルド側テーブル 1 タプルについて，1. タプルの受信，2. 結合キーのハッシュ値からバケットの決定，3. バケット内での結合キーによる探索，4. タプルの挿入，の順に処理を行う．1 と 2 は受信したタプル毎に行うため，ノード i において 1 と 2 の処理に要する CPU 時間を $C_{brecv,i}$，3 の処理に要する CPU 時間を $C_{bsearch,i}$，4 の処理に要する CPU 時間を $C_{binsert,i}$ とする．ノード i のあるバケット j について，ある時間内での当該バケットにおけるタプル受信回数，バケット内でのタプル探索回数，タプル挿入回数を $\Delta N_{brecv,ij}$，$\Delta N_{bsearch,ij}$，$\Delta N_{binsert,ij}$ とする時，当該時間内においてバケットのビルド処理に要する CPU 時間 $\Delta C_{buildbucket,ij}$ は次のように表される．

$$\begin{aligned}\Delta CPU_{buildbucket,ij} = \Delta N_{brecv,ij} C_{brecv,i} + \\ \Delta N_{bsearch,ij} C_{bsearch,i} + \\ \Delta N_{binsert,ij} C_{binsert,i}\end{aligned} \tag{5.2}$$

ビルド処理全体に要した CPU 時間は，当該ビルド処理においてノードに割り当てられた全バケットの CPU 時間の合計となる．バケットの総数を N としたときの，あるノード i のビルド処理の CPU 時間 $C_{build,i}$ は，式 5.2 を用いて，次のように表すことができる．

$$\Delta CPU_{build,i} = \sum_{j \in N} \Delta CPU_{buildbucket,ij} \tag{5.3}$$

ノード i のビルドフェーズの処理に要する CPU 時間を ΔCPU_i は，式 5.1，式 5.3 を用いて次のように表すことができる

$$\Delta CPU_i = \Delta CPU_{scan,i} + \Delta CPU_{build,i} \tag{5.4}$$

CPU_i に対してノード i の CPU 仕様に基づく係数 α_i を掛けることによって，正規化された負荷 ΔL_i を定義する．

$$\Delta L_i = \alpha_i \Delta CPU_i \tag{5.5}$$

■プローブフェーズにおける負荷モデル　次に，ハッシュ結合のプローブフェーズにおける負荷モデルについて述べる．プローブフェーズでは，各ノードにおいてプローブ側のテーブルをフルスキャンし，得られたタプルを結合キー値に基づいて各ノードへと分配した後に，各ノードでビルドフェーズで構築したハッシュテーブルを用いたプローブ処理を行う．ノード i における，プローブ側テーブルのフルスキャンに要する CPU 時間は，ビルドフェーズにおけるフルスキャンと同様に式 5.1 で表すことができる．

プローブフェーズにおけるプローブ処理では，プローブ側テーブルの 1 タプルについて，1. タプルの受信，2. 結合キーのハッシュ値からバケットの決定，3. バケット内での結合キーによる探索，4. 取得したビルド側テーブルのタプルとの結合，5. 結合結果タプルの送信，の順に処理を行う．1 と 2 は受信したタプル毎に，4 と 5 は結合した得られたタプル毎に行うため，ノード i において 1 と 2 の処理に要する CPU 時間を $C_{precv,i}$，3 の処理に要する CPU 時間を $C_{psearch,i}$，4 と 5 の処理に要する CPU 時間を $C_{psend,i}$ とする．ノード i のあるバケット j について，ある時間内での当該バケットにおけるタプル受信回数，バケット内でのタプル探索回数，タプル送信回数を $\Delta N_{precv,ij}$，$\Delta N_{psearch,ij}$，$\Delta N_{psend,ij}$ とする時，当該時間内においてバケットのプローブ処理に要する CPU 時間 $\Delta CPU_{probebucket,ij}$ は次のように表される [47]．

$$\begin{aligned}\Delta CPU_{probebucket,ij} = {} & \Delta N_{precv,ij} C_{precv,i} + \\ & \Delta N_{psearch,ij} C_{psearch,i} + \\ & \Delta N_{psend,ij} C_{psend,i}\end{aligned} \tag{5.6}$$

ノード i におけるプローブ処理全体に要する CPU 時間は，当該プローブ処理においてノードに割り当てられた全バケットの CPU 時間の合計となる．バケットの総数を N としたときの，あるノード i においてプローブ処理に要する CPU 時間 $CPU_{probe,i}$ は，式 5.6 を用いて，次のように表すことができる．

$$\Delta CPU_{probe,i} = \sum_{j \in N} \Delta CPU_{bucket,ij} \tag{5.7}$$

ノード i における処理が，フルテーブルスキャンおよび一段のプローブ処理から構成される場合，ノード i のプローブフェーズの処理に要する CPU 時間を ΔCPU_i は，式 5.1，式 5.7 を用いて次のように表すことができる．

$$\Delta CPU_i = \Delta CPU_{scan,i} + \Delta CPU_{probe,i} \tag{5.8}$$

式 5.8 と CPU 仕様に基づく係数 α_i を用いて，式 5.5 と同様に正規化された負荷 ΔL_i を定義する．

■バケット移行を用いたハッシュ結合の動的負荷分散手法　ビルドフェーズ，プローブフェーズの各フェーズの処理における負荷モデルを示した．当該負荷モデルではバケットごとの負荷を求めることが可能であるため，ノード間で負荷の偏りが存在する場合には，高負荷のノードから低負荷のノードへとバケットを移行し，当該バケットに紐づくタプルの送信先を低負荷のノードへと切り替えることによって，負荷の均衡化を行う．[26, 46, 47].

まず，ビルドフェーズ，プローブフェーズにおいて，ノード i での各回数 ΔN，および処理全体の CPU 時間 ΔCPU_i を複数回計測して，式 5.4，5.8 を用いて連立方程式を立式し，当該連立方程式を解くことによって各処理に要する CPU 時間の係数を決定する．当該係数を用いることによって，計測された ΔN から，各ノードの正規化負荷 L_i を求めることが可能になる．

各ノードの正規化負荷 L_i が等しくなるようにバケット移行することで負荷の均衡下を行う．バケットの移行を用いた動的負荷分散のアルゴリズムは，ビルドフェーズとプローブフェーズについて同じ手法を用いることが可能であるが，例としてビルドフェーズにおける動的負荷分散のアルゴリズムを Algorithm3 に示す．

各ノードは式 5.8 および式 5.1 において用いる各回数をカウントしておき，一定時間ごとに当該回数を回収する．回収された回数から式 5.5 によって用いて各ノードの負荷を計算する．計算によって得られた各ノードの負荷について，最大値の最小値の差であるレンジを平均値で正規化した値を不均衡の度合いとして用い，当該値が一定のしきい値を上回った場合に，ノード間のバケットを移行し負荷を均衡化する．移行するバケットの決定は Algorightm4 に示す手法によって行う．高負荷ノードのバケットを負荷が高い順に選択し，当該バケットを高負荷ノードから低負荷ノードに移行したものとして負荷を求め，当該移行後負荷が最小となるようなバケットの集合を移行対象とする．Algorightm4 によって決定されたバケットを移行した場合にまだ不均衡の度合いがしきい値を上回る場合

は，最大負荷ノードと最小負荷ノードを求めて Algorightm4 を適用することを不均衡の度合いがしきい値を下回るまで繰り返すことによって，移行するバケットを決定する．

5.2.2 インデックス結合における動的負荷分散

インデックス結合では，ネステッドループ結合において内表からのタプルの取得を外表の結合キー値を用いたインデックススキャンによって行う．従来のインデックス結合はプロセッサの利用効率が低いため，並列データベースエンジンにおける負荷分散は主にハッシュ結合を対象として行われてきた．しかし，アウトオブオーダ型実行方式 [77, 78] を用いることで，インデックス結合のプロセッサの利用効率を大幅に向上させることが可能となり，当該手法を用いた場合にはプロセッサ間の処理時間を均衡化する手法を用いることで，プロセッサ間の能力の偏りに起因するクエリ実行速度の低下を軽減することが可能で

Algorithm 3 ハッシュ結合のビルドフェーズにおける動的負荷分散

Require: $threshold$: threshold of load imbalance

function HashJoinLoadBalance($threshold$)
  for $i \leftarrow nodes$ **do**
    for $j \leftarrow buckets(i)$ **do**
      $\Delta CPU_{buildbucket,ij} = calcBucketLoad(i, j)$
    end for
    $CPU_{build,i} = \sum_{i \in N} CPU_{buildbucket,ij}$
    $\Delta C_{scan,i} = calcScanLoad(i)$
    $CPU_i = CPU_{scan,i} + CPU_{probe,i}$
    $L_i = \alpha_i(CPU_{scan,i} + CPU_{probe,i})$
  end for
  $h \leftarrow$ node with the highest L_i
  $l \leftarrow$ node with the lowest L_i
  $ImbalanceFactor = \frac{\Delta L_h - \Delta L_l}{avg(\Delta L_i)}$
  if $ImbalanceFactor > threshold$ **then**
    $buckets = DecideBucketsToMigrate(h, l)$
    Migrate $buckets$ from h to l
  end if
end function

ある．インデックス結合ではハッシュ結合と異なり，タプルをノード間で分配する処理において任意のノードにタプルを送信することが可能である．本論文では，各ノードごとに送信先として決定される重みを定義し，当該重みを用いてインデックス結合の負荷モデルを示し，重みを調整することによって各ノードのプロセッサの処理時間を均衡化する手法を示す．

■インデックス結合における負荷モデル　インデックス結合は，外表のスキャンによるタプルの取得，外表タプルの結合キー値による内表のインデックスの探索，得られたインデックスエントリからの内表タプルの取得と外表タプルとの結合の順に処理を行う．処理の起点となる外表のスキャンの負荷はスキャンの手法やインデックスの種類によって異なるが，本論文では広く用いられている B+ 木のインデックススキャンの場合について議論する．B+ 木を用いた外表のスキャンの処理では，B+ 木内部の探索，B+ 木のリー

Algorithm 4 移行するバケットの決定

Require: h: node with the maximum load
Require: l: node with the minimum load
function DecideBucketsToMigrate(h, i)
    $buckets \leftarrow$ sort buckets of h by its load in descending order
    $bucketsToMigrate \leftarrow$ empty list
    $rem = L_h - L_l$
    for $j \leftarrow buckets$ **do**
        append j to $bucketsToMigrate$
        $L'_h \leftarrow$ calcluate load of h without $bucketsToMigrate$
        $L'_l \leftarrow$ calcluate load of l with $bucketsToMigrate$
        $newRem = |L_h - L_l|$
        if $newRem > rem$ **then**
            remove j from $bucketsToMigrate$
            break
        end if
        $rem = newRem$
    end for
    return $bucketsToMigrate$
end function

フページの取得，タプルの送信を行う．ノード i において，ある時間内における木内部の探索，リーフページの取得，タプルの送信のそれぞれの回数を，$\Delta N_{search,i}$，$\Delta N_{page,i}$，$\Delta N_{send,i}$ とし，それぞれの処理に要する CPU 時間を C_{search}，C_{page}，C_{send} とすると，外表のスキャン処理に要する CPU 時間は次式で表すことができる．

$$\begin{aligned}\Delta C_{scan,i} = &\Delta N_{search,i} C_{search,i} + \\ &\Delta N_{page,i} C_{page,i} + \\ &\Delta N_{send,i} C_{send,i}\end{aligned} \tag{5.9}$$

次に，外表タプルの結合キー値による内表インデックス探索処理の負荷モデルについて述べる．内表インデックス探索処理では，1 タプルの受信毎に，1. 外表のタプルを受信，2. 受信タプルの結合キーの値を用いて内表のインデックスをスキャン，3. 得られたインデックスエントリと外表のタプルを送信，の処理を順に行う．ノード i における 1 および 2 に要する CPU 時間を $C_{irecv,i}$，3 に要する CPU 時間を $C_{isend,i}$ とした時，インデックススキャンによって得られたインデックスエントリの数を j とすると，当該タプルのインデックス探索処理に要する CPU 時間は $C_{irecv,i} + j * C_{isend,i}$ となる．外表と内表の平均結合率を J とすると，得られるインデックスエントリの数 j はインデックス探索処理が十分な回数実行された場合には J に近づくと考えられる．ノード i において，ある時間内におけるインデックス探索のタプル受信回数を，$\Delta N_{index,i}$ とすると，$\Delta N_{index,i}$ が十分に大きい時，当該時間内におけるノード i でのインデックス探索処理に要する合計 CPU 時間 $C_{index,i}$ は次のように表すことができる．

$$\begin{aligned}\Delta C_{index,i} &= \Delta N_{index,i}(C_{irecv,i} + J * C_{isend,i}) \\ &= \Delta N_{index,i} C_{itotal,i}\end{aligned} \tag{5.10}$$

ただし，$C_{itotal,i} = C_{irecv,i} + J * C_{isend,i}$ である．

ここで，各ノード間でのタプルの分配におけるノード i への送信の重みを w_i とする．全ノードの数を N とし，$S = \{1, 2, ..., N\}$ とすると，タプル分配におけるノード i への送信割合は $w_i / \sum_{i \in S} w_i$ となるが，簡単のために $\sum_{i \in S} w_i = 1$ が成り立つように w_i を定めるものとする．

この時，ノード i のインデックス探索処理に対して送信されたタプルの数を $\Delta N_{iprevsend,i}$ とすると，ノード i においてインデックス探索処理が受信したタプル数 $N_{index,i}$ は次のように表すことができる．

$$\begin{aligned}\Delta N_{index,i} &= \sum_{i \in S} w_i \Delta N_{iprevsend,i} \\ &= w_i \sum_{i \in S} \Delta N_{iprevsend,i}\end{aligned} \tag{5.11}$$

式 5.11 を用いると，式 5.10 は次のようになる．

$$\Delta C_{index,i} = w_i C_{itotal,i} \sum_{i \in S} \Delta N_{iprevsend,i} \tag{5.12}$$

インデックスエントリを用いて内表タプルの取得し外表タプルとの結合を行う処理の負荷モデルについて述べる．当該処理では，1 つのインデックスエントリの受信毎に，1. インデックスエントリと外表のタプルを受信，2. インデックスエントリから内表のタプルを取得，3. 外表のタプルと内表のタプルを結合，4. 結合タプルの送信，の処理を順に行う．1,2,3,4 の処理はインデックスエントリの受信毎に常に一定の処理となるため，ノード i における 1,2,3,4 の処理に要する合計の CPU 時間を C_{jtotal} とする．ノード i において，ある時間内におけるインデックスエントリの受信数を，$\Delta N_{join,i}$ とすると，ノード i での内表タプル取得および結合の処理に要する合計 CPU 時間 $CPU_{join,i}$ は次のように表すことができる．

$$\Delta CPU_{join,i} = \Delta N_{join,i} C_{jtotal,i} \tag{5.13}$$

ノード i において内表タプル取得処理に送信されたタプルの数を $\Delta N_{jprevsend,i}$ とすると，式 5.13 は次のように表すことができる．

$$\Delta CPU_{join,i} = w_i C_{jtotal,i} \sum_{i \in S} \Delta N_{jprevsend,i} \tag{5.14}$$

ノード i におけるインデックス結合が，外表のインデックススキャンおよび一段のインデックス結合から構成される場合，ノード i の処理に要する CPU 時間 ΔCPU_i は式 5.12，式 5.13，式 5.9 を用いて次のように表すことができる．

$$\Delta CPU_i = \Delta C_{scan,i} + \Delta CPU_{index,i} + \Delta CPU_{join,i} \tag{5.15}$$

ハッシュ結合と同じく，式 5.15 とノード i の CPU 仕様に基づく係数 α_i を用いて，ノード i の正規化負荷 L_i を次のように定義する．

$$\Delta L_i = \alpha_i \Delta CPU_i \tag{5.16}$$

5.2.3 送信先決定の重みを用いたインデックス結合の動的負荷分散手法

式 5.16 において，各ノードへの送信先決定の重み w_i を用いた，各ノードのインデックス結合の負荷モデルを示した．w_i を調整する事によって，インデックス結合における各ノードの負荷を均衡化する手法を示す．

まず，ノード i において，各回数 ΔN，および全体の CPU 時間 ΔC_i を複数回計測して，式 5.15 を用いて連立方程式を立式し，当該連立方程式を解くことによって各処理に要する CPU 時間の係数を決定する．

次に，ノード i への送信の重みを w'_i に調整する時，調整後のノードの負荷 L'_i をは次のようになる．

$$\begin{aligned} CPU'_i = {} & w'_i C_{itotal,i} \sum_{i \in S} \Delta N_{iprevsend,i} + \\ & w'_i C_{ttotal,i} \sum_{i \in S} \Delta N_{tprevsend,i} + \\ & \Delta C_{scan,i} \\ \Delta L'_i = {} & \alpha_i \Delta CPU'_i \end{aligned} \tag{5.17}$$

全てのノードで L'_i が等しいとして連立方程式を立式し，当該連立方程式を解いて全ノードの負荷が全て等しくなるような w'_i を求める．

$$L'_1 = L'_2 = ... = L'_N \tag{5.18}$$

求めた調整後の送信の重み w'_i を全ノードに送信し，各ノードにおいては受信した重みに従ってタプル毎の送信先を決定することにってノード間の負荷を均衡化する．

第 6 章

動的負荷分散機構を備えた共有ストレージ型並列データベースエンジンの試作実装と評価

6.1 動的負荷分散機構を備えた共有ストレージがデータベースエンジンの試作実装

前章で示した動的負荷分散手法の有効性を実験によって評価するために，当該手法をに基づいた共有ストレージ型データベースエンジンを試作した．当該試作のシステム構成図を図 6.1 に示す．当該試作では，クエリ実行に割り当てるノードに加えてクエリの実行全体を管理するドライバと称するノードを用いるが，提案手法での一定時間ごとの各ノードの計測値の収集，負荷の不均衡の判定，負荷分散手法の実行はこのドライバノードにおいて行うようにした．各ノードにおける負荷の計算において，CPU 仕様に基づく係数 α_i として CPU コア数の逆数を用いた．

当該試作におけるクエリの実行方式は，アウトオブオーダ型実行方式 [77, 78] に基づいたものを用いた．当該実行方式は，クエリの実行時に入出力処理に基づいて動的にタスク分解を行い，当該タスクを複数の演算資源へと振り分けて並列に実行する．また，データベースエンジンにおける共有ストレージとキーバリューストアをを用いるようにした．当該試作の共有ストレージとして用いるための要件として，キーバリューストアでは主キーによる値の取得に加えて，全件の取得，及び値に対して二次索引を作成し当該二次索引による値の取得が可能であることを必要とする．

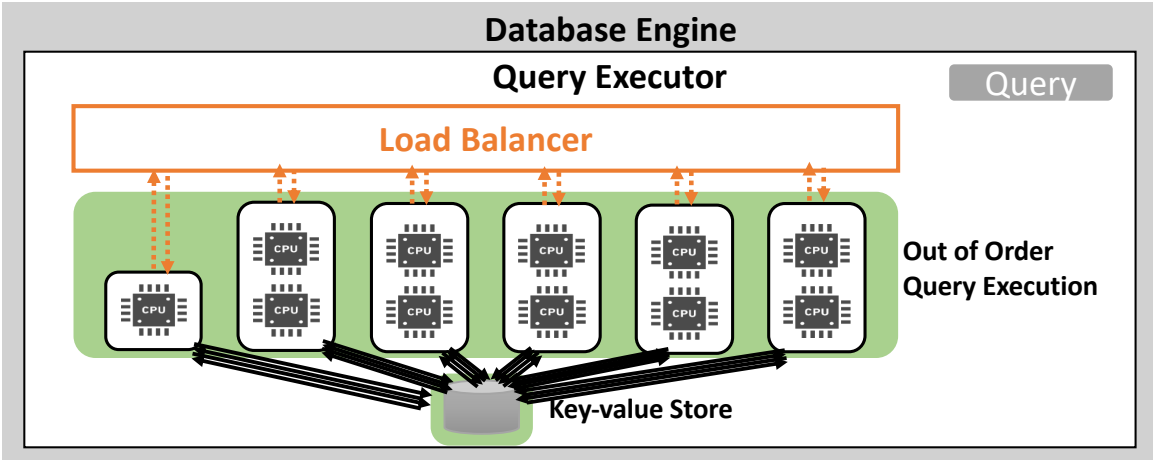


図 6.1: 動的負荷分散機構を有する共有ストレージ型並列データベースエンジンの試作実装

6.2 実験環境

パブリッククラウド環境の Amazon Web Service（AWS) を用いて実験システムを構築した．計算資源として EC2 の c4.8xlarge タイプのインスタンスと，c4.2xlarge タイプのインスタンスを用いた．共有ストレージとしては DynamoDB を用いた．表 6.1 に実験環境の諸元を示す．

データセットとしては，TPC-H ベンチマークの dbgen を用い，Scale Factor = 100 で生成したデータを用いた．さらに，データの偏りによるクエリ実行速度への影響を評価するために，dbgen が生成するデータに対して orders の o_custkey，lineitem の l_partkey の値の分布を偏らせたものを作成した．o_custkey，l_partkey の分布は zipf 分布を用いた．zipf 分布は，全要素の数 N における順位 k の頻度として表される．

$$f(k;s,N) = \frac{\frac{1}{k^s}}{\sum_{n \in N} \frac{1}{n^s}} \tag{6.1}$$

s が 0 で一様分布となり，大きくなるにつれて偏りの大きい分布となる．元の dbgen 用いて生成した一様分布のデータセット，および $s = 1$，$s = 2$ を用いて生成した偏りのあるデータセットをそれぞれ，Uniform，Moderate，High と称することとする．作成した各データセットは Dynamodb にテーブルとしてロードし，各テーブルの外部キーには TPCH の仕様に基づいた 2 次索引を作成した．

表 6.1: AWS（N. Virginia region）実験環境諸元

EC2 c4.8xlarge	
CPU	36 vCPU
Memory	60 GiB
OS	Amazon Linux 64bit (hvm)
Hardware	Shared (Default Tenancy)
Network Bandwidth	10Gbps
Network Location	Colocated (Placement Group)
EC2 c4.2xlarge	
CPU	8 vCPU
Memory	15 GiB
OS	Amazon Linux 64bit (hvm)
Hardware	Shared (Default Tenancy)
Network Bandwidth	High
Network Location	Colocated (Placement Group)

```
select p_brand,
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from part
join lineitem on part.p_partkey = lineitem.l_partkey
where [part selection condition]
group by p_brand
```

図 6.2: 評価実験に用いたクエリ: Q1

クエリとしては図 6.2，6.3，6.4 に示す 3 つのクエリを用いた．各クエリは結合を含むものであり，結合方式としてはハッシュ結合およびインデックス結合をクエリの選択率に応じて決定した．

```
select o_orderpriority,
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from customer
join orders on customer.c_custkey = orders.o_custkey
join lineitem on o_orderkey = l_orderkey
where [customer selection condition]
group by o_orderpriority
```

図 6.3: 評価実験に用いたクエリ: Q2

```
select n_name,
       sum(ps_supplycost),
       sum(l_quantity)
       sum(l_extendedprice * (1 - l_discount) * (1 + l_tax))
from customer
join orders on customer.c_custkey = orders.o_custkey
join lineitem on o_orderkey = l_orderkey
join partsupp on l_partkey = ps_partkey and l_suppkey = ps_suppkey
join supplier on s_suppkey = ps_suppkey
join nation on s_nationkey = n_nationkey
where [customer selection condition]
group by n_name
```

図 6.4: 評価実験に用いたクエリ: Q3

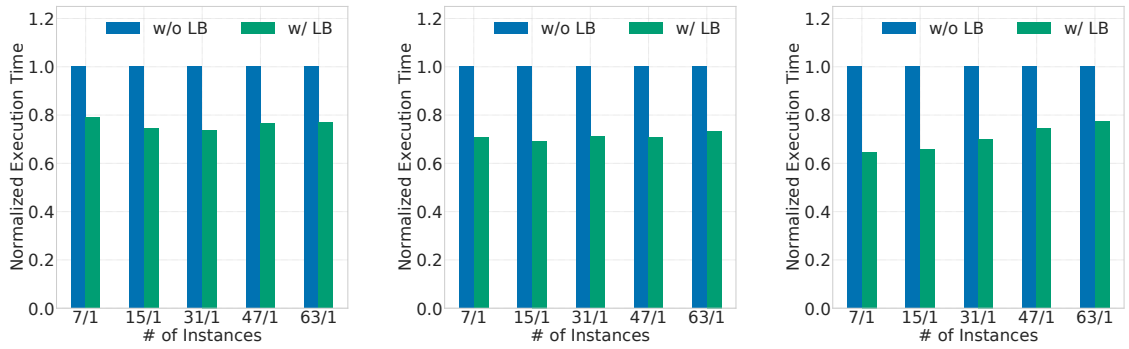
6.3 プロセッサの処理能力の偏りに対する負荷分散

6.3.1 プロセッサ処理性能の動的推定

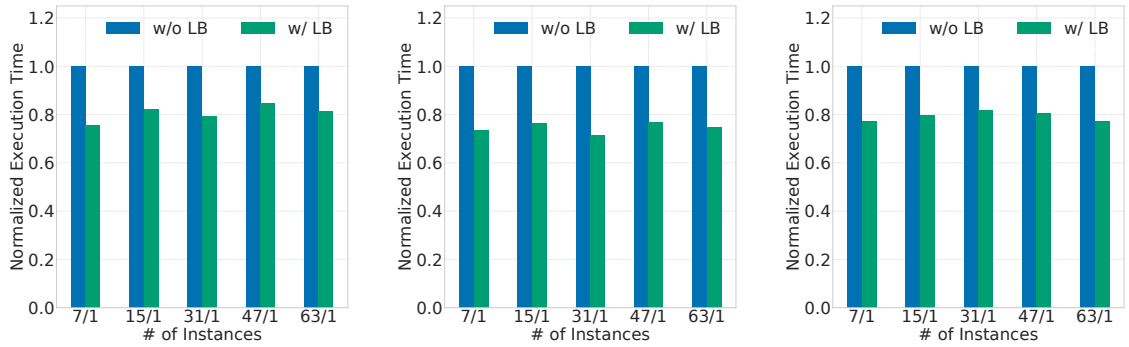
並列データベースエンジンでは複数のプロセッサを用いてクエリを実行するが，プロセッサごとの能力が等しくない場合，各プロセッサに等しい仕事を割当てると，プロセッサ間の処理時間に偏りが生じ，最も遅いプロセッサの処理時間によってクエリ全体の実行時間が律速される．非均質な演算資源を用いてクエリを実行する場合に，プロセッサの利用効率を高めるためには，プロセッサ毎の能力に基づいて負荷を分散する手法が求められる．

6.3.2 静的な資源の非均質性に対する負荷分散

能力の低いプロセッサが 1 つ混在している状況における提案手法の有効性を示す．偏りのない Uniform データセットに対し，Q1，Q2，Q3 の各クエリについて，選択率は 10%，



(a) インデックス結合 クエリ 1　(b) インデックス結合 クエリ 2　(c) インデックス結合 クエリ 3

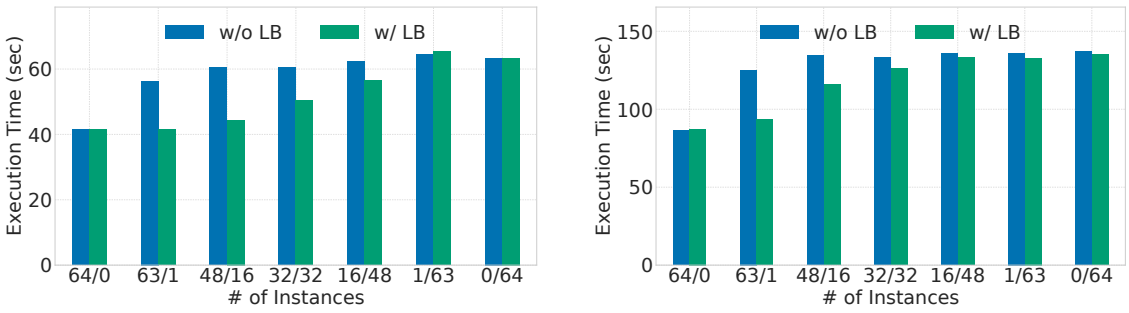


(d) ハッシュ結合 クエリ 1　(e) ハッシュ結合 クエリ 2　(f) ハッシュ結合 クエリ 3

図 6.5: 低性能なインスタンスが 1 つ混在している状況下での動的負荷分散を用いたクエリ実行時間．x 軸の左の値は高性能な c4.8xlarge のインスタンス数，x 軸の右の値は低性能な c4.2xlarge のインスタンス数，y 軸は動的負荷分散を用いなかった場合を 1 とした正規化実行時間を表す．

25% とし，10% の場合はインデックス結合を，25% の場合はハッシュ結合を用いて実行時間を計測した．インスタンスのタイプは，c4.8xlarge，c4.2xlarge を用い，インスタンス数として c4.8xlarge は 7,15,31,47,63 インスタンスを，c4.2xlarge は常に 1 インスタンスを用いた．これらのクエリについて動的負荷分散を無効にした場合と有効にした場合の 2 通りで計測を行った．動的負荷分散を用いない場合の実行時間を 1 とした正規化実行時間を図 6.5 に示す．正規化実行時間は動的負荷分散による実行速度の変化を表しており，正規化実行時間が短いほど動的負荷分散によって得られた実行速度の向上が大きいことを表す．

インデックス結合，ハッシュ結合の両方について，動的負荷分散を用いた場合に実行時間の短縮が得られた．クエリ Q1，Q2，Q3 のそれぞれについて，インデックス結合では



(a) インデックス結合 実行時間　　(b) ハッシュ結合 実行時間

図 6.6: 低性能インスタンスの割合を変えた場合のクエリ実行時間の比較．x 軸の左の値は高性能な c4.8xlarge のインスタンス数，x 軸の右の値は低性能な c4.2xlarge のインスタンス数を表す．

最短で 0.738，0.693，0.646 の，ハッシュ結合では最短で 0.755，0.713，0.773 の正規化実行時間が得られた．本実験により，提案手法はインデックス結合，ハッシュ結合の両方について，能力の低いプロセッサ一つ混在する影響を軽減可能であることを示した．

能力が異なるプロセッサの割合による影響を確認するため，Uniform データセットに対して，c4.8xlarge と c4.2xlarge を合わせて 64 インスタンスを用いて Q1,Q2,Q2 を実行し，実行時間を計測した．64 インスタンスのうち，c4.2xlarge のインスタンス数は 0，1，16，32，48，63，64 とした．Q1,Q2,Q3 の選択率は 10% と 25% を用い，10% のときはインデックス結合，25% のときはハッシュ結合を用いた．計測した実行時間を図 6.6 に示す．インデックス結合，ハッシュ結合ともに能力が低い c4.2xlarge インスタンスが 1 つ混ざっているだけで実行時間が長くなった．一方，能力が異なる割合を変えた際の動的負荷分散による実行速度の向上は，均質なインスタンスを用いた場合を除き，能力の低いインスタンス数の割合が小さいほど大きかった．これは，低性能のインスタンスが 1 つでも存在すると当該インスタンスによって全体の実行速度が律速されるが，低性能インスタンスの割合が低い場合は低負荷となっている高性能インスタンスの数が多いために，動的負荷分散による性能向上の余地が大きくなるためと考えられる．パブリッククラウド環境においては，能力の異なるインスタンスを用いなければならない状況が起こりうるため，パブリッククラウド環境を用いた並列データベースエンジンにおいて動的負荷分散機構を備えることがより重要であると言える．本実験により，低い能力のプロセッサが少数混在する場合に実行速度が大きく低下すること，また提案手法によって当該影響を軽減可能であることを明らかにした．

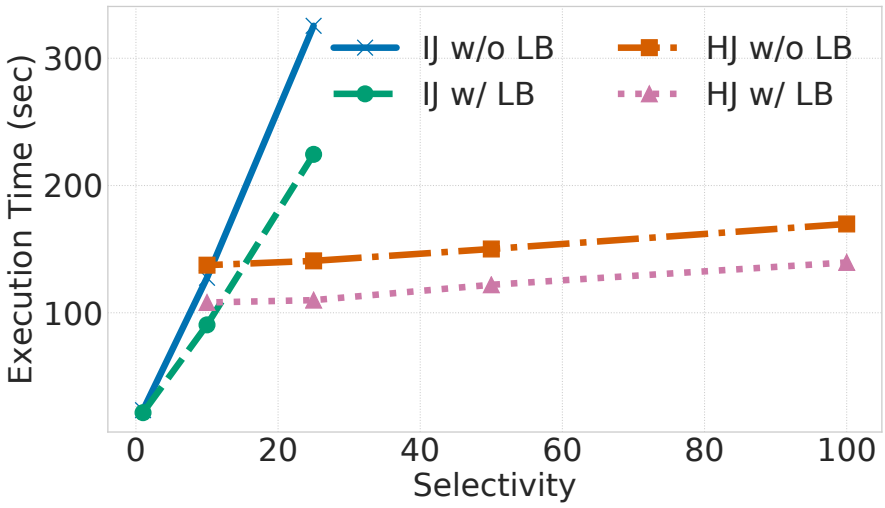


図 6.7: 同一クエリを異なる選択率において実行した場合の実行時間．ハッシュ結合，インデックス結合のそれぞれについて動的負荷分散を用いなかった場合と用いた場合の実行時間を表す．

また，高性能の c4.8xlarge を 64 インスタンス，低性能の c4.2xlarge を 1 インスタンス用いて，Q3 を異なる選択率において実行し，負荷分散を有効にした場合と無効にした場合のそれぞれについて実行時間を計測した．選択率として，ハッシュ結合を用いた場合の選択率を 100%, 50%, 25%, 10%，インデックス結合を用いた場合の選択率を 25%, 10%, 1% とした．結果を図 6.7 に示す．ハッシュ結合，インスタンス結合ともに動的負荷分散による実行速度の向上が得られた．また，ハッシュ結合では選択率によって実行時間は大きくは変わらない一方で，インデックス結合では選択率が高くなるにつれ実行時間が増加し，選択率 25% では負荷分散を無効にした場合と有効にした場合の両方でハッシュ結合の方が方が短い実行時間となった．即ち，本実験に用いたクエリでは選択率 10% 以下ではインデックス結合が，選択率 25% 以上ではハッシュ結合が優位な結合方式となったが，どの選択率においても優位な結合方式に対して提案手法による実行速度の向上が得られた．本実験により，提案手法によってハッシュ結合とインデックス結合の両方について動的負荷分散を行うことで，広範な選択率においてより優位な結合方式を選択しつつ非均質な演算資源の影響を軽減可能であることを示した．

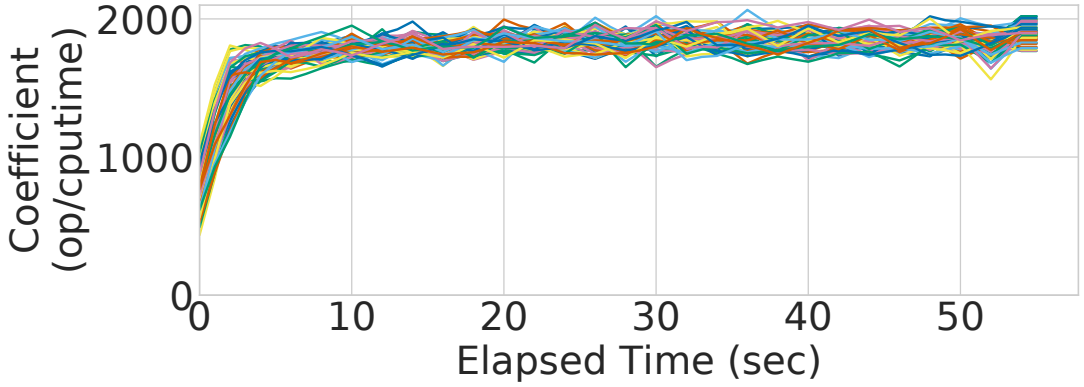


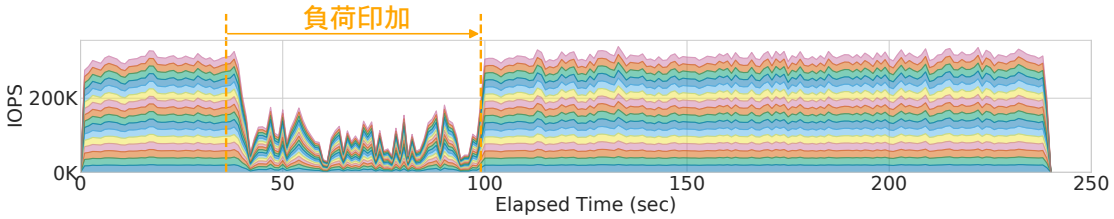
図 6.8: インデックス結合クエリにおいて推定された処理性能の経時変動

6.3.3 動的な資源の非均質性に対する負荷分散

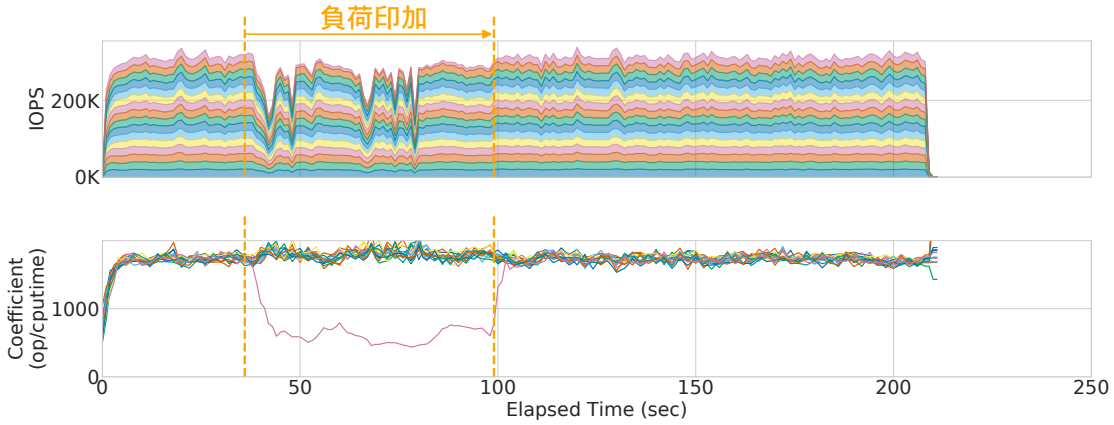
クラウド環境が提供するインスタンスは仮想化されたサーバであるため，同一物理マシン上に割当てられた他のインスタンスが高負荷な処理を行うことによる影響や，別の実行中のインスタンスを別の物理マシンに移送するインスタンスマイグレーションなどによって，インスタンスから実際に得られる処理能力が動的に変動する場合がある．本論文で提案する動的負荷分散手法は，クエリの実行時に得られる情報を用いてクエリ実行に用いているプロセッサの処理能力を動的に推定するため，動的な処理能力の変動に起因する負荷の偏りを均衡化し，処理能力の変動の影響を吸収可能であることを示す．

提案手法によってプロセッサの処理性能が推定可能であることを示すために，c4.8large を 64 インスタンス用いてインデックス結合クエリを実行し，1 秒ごとに推定された性能を記録した．各インスタンにおいて推定された処理性能を図 6.8 に示す．推定された処理性能は各インスタンスで概ね等しい値となった．本実験により，提案手法によってプロセッサの処理能力をクエリの実行中に推定可能であることを示した．

提案手法によって，動的に変動するプロセッサの処理能力の影響を軽減可能であることを示すために，c4.8xlarge を 16 インスタンス用いて，Q1 をインデックス結合によって実行するとともに，クエリ実行開始の 40 秒後から，実行に割当てたインスタンスのうちのある 1 インスタンスにおいて，当該インスタンス 50 において % の負荷となるような別のプロセスを 60 秒間実行した．当該プロセスによって当該インスタンスに負荷を印加することによって，当該プロセスを実行しているインスタンスにおける動的な性能変動を模擬



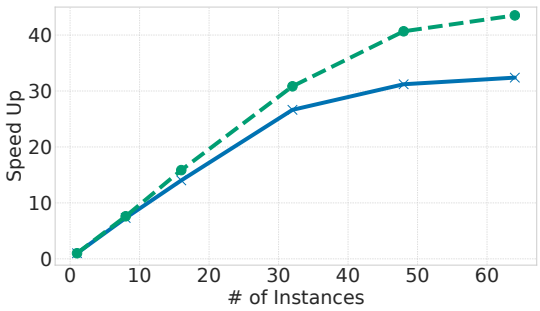
(a) 動的負荷分散なし



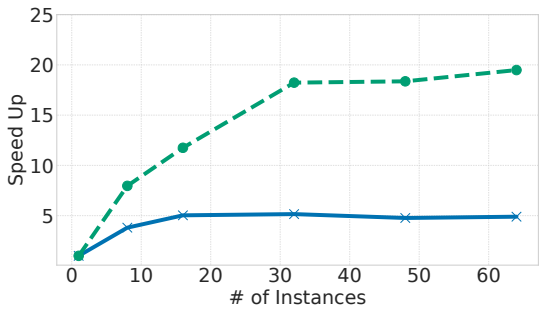
(b) 動的負荷分散あり

図 6.9: 動的な処理能力の変動に対する動的負荷分散

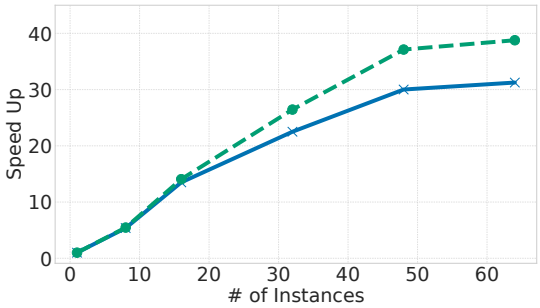
するものである．動的負荷分散を用いた場合，用いなかった場合の両方について 1 秒ごとに各インスタンスにおける IOPS を計測し，動的負荷分散を用いた場合には推定された処理能力を併せて計測した．得られた結果を図 6.9 に示す．動的負荷分散を用いなかった場合には，負荷印加をによってインスタンス間の性能の不均衡が発生し，負荷印加を行ったインスタンスにおける IOPS によって，他のインスタンスにおける IOPS が律速され，クエリ全体の処理速度が大きく低下した．一方，動的負荷分散を用いた場合には，負荷印加を行っているインスタンスの推定処理性能が低下し，当該推定性能に基づく負荷分散によって，他のインスタンスへの処理速度への影響を軽減し，クエリの実行時間を短縮することが可能であることが確認された．本実験によって，クエリ処理に用いる演算資源の処理能力が動的に変動する場合に，提案手法によって処理能力の変動を推定し，推定された処理能力に基づいた負荷分散を行うことによって，動的な動的な演算資源の非均質性の影響を軽減することが可能であることを明らかにした．



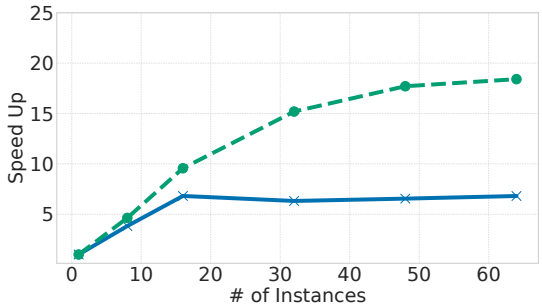
(a) クエリ 1 Moderate



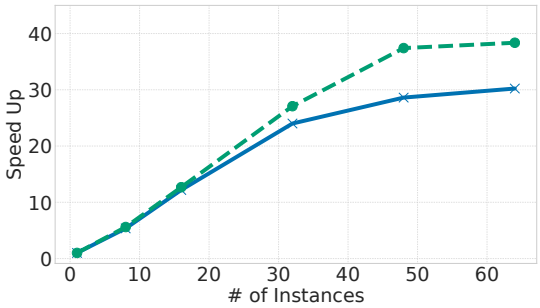
(b) クエリ 1 High



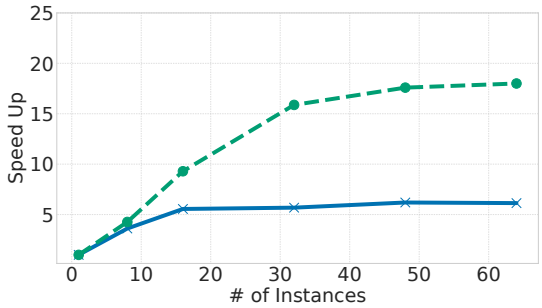
(c) クエリ 2 Moderate



(d) クエリ 2 High



(e) クエリ 3 Moderate



(f) クエリ 3 High

図 6.10: データ分布に偏りのあるデータセットに対するクエリ実行の性能向上率

6.4 データの分布の偏りに対する負荷分散

並列データベースエンジンにおけるハッシュ結合で結合キーの値の分布が一様でない場合，プローブフェーズにおけるタプルの送信先は結合キーによって決定されるため，特定のノードに多くのタプルが送信されノード間の負荷の偏りが発生することが広く知られる

ている．動的負荷分散手法によって，多くタプルが送られて高負荷となっているノードのバケットを他の低負荷なノードに移すことで，ノード間の負荷が均衡化されクエリの実行速度への影響を軽減可能であることを示す．

結合キーの分布に偏らせたデータセットである Moderate, High の各データセットに対して，Q1,Q2,Q3 を動的負荷分散を用いない場合と用いた場合で実行し，それぞれの実行時間を計測した．クエリ実行には c4.8xlarge を 1,8,16,32,48,64 インスタンス用い．選択率を 25% としてハッシュ結合によって実行した．得られた結果について，インスタンス数 1 のときの実行時間に対する性能向上率を図 6.10 に示す．

図 6.10 より，動的負荷分散を用いることで，用いない場合と比べてより高い実行速度の向上が得られた．動的負荷分散を用いない場合と比べて Moderate データセットでは最大 1.34 倍，High データセットでは最大 3.97 倍の実行速度の向上が得られた．High データセットでは，データの分布の偏りがより大きいため，ノード間の負荷の偏りが大きく，動的負荷分散によるプロセッサの利用効率向上の余地が大きい．その為，High データセットにおいてより大きな実行速度の向上が得られたと考えられる．本実験により，提案手法によってプロセッサの能力の偏りのみではなく，データの分布の偏りに起因する負荷の非均衡についても影響を軽減可能であることを示した．

第 7 章

結論

本論文では，共有ストレージ型並列データベースエンジンにおけるクラウド資源の高次利用手法として，データベースエンジンにおいてクエリ実行時の資源伸縮性を実現する動的資源調整手法，ならびに資源非均質性の吸収を実現する動的負荷分散手法を提案した．

動的資源調整手法においては，従来，データベースエンジンは実行中のクエリに対してその実行に割当てる資源の調整を行うことが不可能であったのに対して，共有ストレージ方式の並列データベースエンジンを対象として実行中のクエリに対して演算資源の割当てを動的に変更することを可能とする動的資源調整手法を提案した．提案手法に基づく演算資源調整機構を備えた備えた共有ストレージ型並列データベースエンジンの試作を行い，パブリッククラウド環境を用いて構成した実験システムにおいて評価実験を行ったところ，インデックス結合，ハッシュ結合といった代表的な演算について，提案手法によってクエリ実行時の資源伸縮性が実現可能であることが確認された．また，実際のデータベースの利用状況を模擬したケーススタディを行ったところ，提案する動的演算資源調整手法を用いることによって，優先度に基づいた資源割当てを行い高優先度クエリの割込み実行が可能となるほか，各々のクエリに設定されたデッドラインを考慮した資源割当てを行い各クエリのデッドライン超過時間の低減が可能となること，ならびにデータベースエンジンにおいて利用可能資源に変動がある場合においても資源調整が可能となることを示した．さらに，パブリッククラウド環境の 256 インスタンスを用いて構成した実験システムにおいて性能評価を行ったところ，256 インスタンスで最大 242.3 倍の性能向上，9.4M IOPS の処理性能が得られることが確認された．

動的負荷分散手法においては，並列データベースエンジンにおいてクエリの実行に用いる演算資源の処理能力が均質でない場合に，演算資源の処理能力の利用効率が低下する場合があるという問題を指摘し，当該問題を解決するために，演算資源の処理性能をクエリ

の実行時に動的に推定し，当該推定処理能力に応じて負荷を分散させることにより演算資源の処理能力の利用効率を向上する動的負荷分散手法を提案した．パブリッククラウド環境を構成した実験システムにおいて当該手法の評価実験を行ったところ，提案手法によってインデックス結合では最大 0.693 倍，ハッシュ結合では最大 0.713 倍のクエリ実行時間の短縮が得られることが確認された．低性能の演算資源は 1 つ存在するだけでクエリの実行速度が大きく低下することを示し，提案手法によって当該影響を軽減可能であることを示した．また，演算資源の性能が動的に変動することを模擬した評価実験によって，提案手法によって動的な資源非均質性の影響を軽減可能であることを示した．さらにハッシュ結合について，データの偏りに対する負荷分散の評価実験を行ったところ，最大 3.97 倍の性能向上が得られることが確認された．

本論文では，演算資源を対象とし，クエリ実行時の演算資源伸縮を可能とする動的演算資源調整手法と，演算資源非均質性の吸収を可能とする動的負荷分散手法を示した．当然のことながら，並列データベースエンジンのクエリ実行には，演算資源のみではなく，メモリ資源，ストレージ資源，ネットワーク資源などの他の資源も必要となる．演算資源によって実効性能が律速されているクエリに対しては，本論文で提案する動的演算資源調整手法を用いて，並列データベースエンジンにおいて実行中クエリに割当てる演算資源を調整することにより，クエリ実行時に当該クエリの処理スループットを変動させることが可能であるが,，他の資源によって実行性能が律速されるクエリに対して処理スループットを変動させるためには，当該資源利用を動的に調整する手法が必要となるであろう．

また，本論文では，共有ストレージ型の並列データベースエンジンを前提とし，共有ストレージ型並列データベースエンジンにおける動的演算資源調整手法を提案した．並列データベースエンジンのアーキテクチャとしては並列データベースエンジンの他に無共有型が広く用いられている．無共有型は事前にデータを分割して各計算機に配置し，クエリ実行が要求された場合に，当該クエリの実行におけるタスクの分割と事前のデータ分割とを合わせることが可能である場合には，クエリの実行時に計算機間での通信を削減し，クエリの実行を高速化することが可能となる．無共有型では各々の計算機に分割されたデータの一部が割当てられるため，無共有型に対して動的演算資源調整を行う場合には，計算期間でデータの移送が行うことが必須となる．無共有型へ動的演算資源調整手法を拡張する際には，データ移送による影響を軽減する手法が必要となることが予想される．

IT システムの構築におけるクラウド環境のシェアは今後さらに高まるものと予想されており，またビジネス上のワークロードに基づいたユーザからの IT システムへの要求も更に複雑化していくものと推察される．データベースエンジンは IT システムにおいてデータ管理・取得の重要な役割を担うソフトウェアであり，データベースエンジンにおけ

る資源利用を動的に制御することによって，今後さらに複雑化してゆくユーザの要求に応じた資源利用を可能とすることが求められるようになるであろう．データベースエンジンにおける資源利用を制御する手法の一例として，共有ストレージ型の並列データベースエンジンについて演算資源に対する動的資源伸縮と動的資源非均質性吸収をを実現したという点において，本研究の価値があると言えよう．データベースエンジンにおける動的な資源利用の制御は未だ十分に開拓されていない領域であり，本研究を当該領域の端緒としたい．

謝辞

本研究を遂行し学位論文に纏めるにあたり，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，素晴らしい研究環境を与えて頂き，時に厳しく，時に温かく御指導，御鞭撻頂きました．喜連川教授のご指導の下，本研究を通して得られた経験は大変に得難いものであり，必ずや今後の大きな糧となるものと存じます．心より厚く感謝申し上げます.

学位審査において，主査である東京大学大学院情報理工学系研究の田浦健次朗教授を始め，坂井修一教授，相澤清晴教授，入江英嗣准教授，東京大学生産技術研究所の豊田正史教授には，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.

本研究の遂行にあたって，内閣府最先端研究開発支援プログラム (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」より多大な御支援を頂きました．当該プログラムの支援に深く感謝申し上げます．また，本研究の実験システム構築において，Amazon Web Services, Inc より AWS Cloud Credits for Research Program による御支援の下，実験環境資源を御提供頂きました．ここに深く感謝申し上げます.

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博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・知識について御指導頂き，本研究を進めるにおいての数多くの貴重な御助言を頂いたのみでなく，常に順風とは言えない研究生活において大変辛抱強くご指導いただきました．ここに厚く感謝申し上げます．また，喜連川研究室の方々には日々の研究生活において様々な面で大変お世話になりました．深く感謝申し上げます.

最後になりますが，著者の進む道を応援し，遠い故郷より見守って下さった祖父母，両親に深く感謝します．そして，時に辛い研究生活を長きに渡って温かく支えてくれた妻と二人の子に心から感謝します．本当にありがとうございました.

参考文献

[1] Peter Mell, Tim Grance, et al. “The NIST definition of cloud computing”. In: (2011).

[2] John Gantz and David Reinsel. “The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east”. In: *IDC iView: IDC Analyze the future* 2007.2012 (2012), pp. 1–16.

[3] Benoit Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 215–226. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903741. URL: http://doi.acm.org/10.1145/2882903.2903741.

[4] Anurag Gupta et al. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1917–1923. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742795. URL: http://doi.acm.org/10.1145/2723372.2742795.

[5] Benjamin Farley et al. “More for your money: exploiting performance heterogeneity in public clouds”. In: *ACM Symposium on Cloud Computing, SOCC ’12*. 2012, p. 20. DOI: 10.1145/2391229.2391249. URL: http://doi.acm.org/10.1145/2391229.2391249.

[6] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance”. In: *PVLDB* 3.1 (2010), pp. 460–471.

[7] Guohui Wang and T. S. Eugene Ng. “The Impact of Virtualization on Network Performance of Amazon EC2 Data Center”. In: *INFOCOM 2010. 29th IEEE*

International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 2010, pp. 1163–1171. DOI: 10.1109/INFCOM.2010.5461931. URL: https://doi.org/10.1109/INFCOM.2010.5461931.

[8] 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 1993), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20-23, 1993*. IEEE Computer Society, 1993, pp. 230–242. ISBN: 0-8186-3330-1. DOI: 10.1109/PDIS.1993.253088. URL: https://doi.org/10.1109/PDIS.1993.253088.

[9] Jeff Shute et al. "F1: the fault-tolerant distributed RDBMS supporting google's ad business". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 2012, pp. 777–778.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud". In: *ACM Trans. Database Syst.* 38.1 (2013), 5:1–5:45.

[11] Matthias Brantner et al. "Building a database on S3". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. 2008, pp. 251–264.

[12] Alexandre Verbitski et al. "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases". In: *SIGMOD Conference*. 2017, pp. 1041–1052.

[13] Justin J. Levandoski et al. "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*. 2011, pp. 123–133.

[14] Markus Pilman et al. "Fast Scans on Key-Value Stores". In: *PVLDB* 10.11 (2017), pp. 1526–1537.

[15] Simon Loesing et al. "On the Design and Scalability of Distributed Shared-Data Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 2015, pp. 663–676.

[16] Michael Stonebraker. “The Case for Shared Nothing”. In: *IEEE Database Eng. Bull.* 9.1 (1986), pp. 4–9.

[17] David J. DeWitt and Jim Gray. “Parallel Database Systems: The Future of High Performance Database Systems”. In: *Commun. ACM* 35.6 (1992), pp. 85–98.

[18] Apache Hadoop. *http://hadoop.apache.org/*.

[19] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. “Locality-aware Partitioning in Parallel Database Systems”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 2015, pp. 17–30.

[20] Carlo Curino et al. “Schism: a Workload-Driven Approach to Database Replication and Partitioning”. In: *PVLDB* 3.1 (2010), pp. 48–57.

[21] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. “SWORD: scalable workload-aware data placement for transactional workloads”. In: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. 2013, pp. 430–441.

[22] Shumo Chu, Magdalena Balazinska, and Dan Suciu. “From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 2015, pp. 63–78. DOI: 10.1145/2723372.2750545. URL: http://doi.acm.org/10.1145/2723372.2750545.

[23] Immanuel Trummer and Christoph Koch. “Parallelizing Query Optimization on Shared-Nothing Architectures”. In: *PVLDB* 9.9 (2016), pp. 660–671.

[24] Goetz Graefe and William J. McKenna. “The Volcano Optimizer Generator: Extensibility and Efficient Search”. In: *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*. 1993, pp. 209–218.

[25] 合田 和生 et al. “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”. In: 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* 87.6 (June 2004), pp. 661–674. ISSN: 09151915.

[26] Kazuo Goda et al. “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*. 2002, pp. 182–192.

[27] Masato Oguchi and Masaru Kitsuregawa. “Runtime Data Declustering over SAN-Connected PC Cluster System”. In: *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*. 2002, p. 275.

[28] Sven Groot, Kazuo Goda, and Masaru Kitsuregawa. “Towards improved load balancing for data intensive distributed computing”. In: *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*. 2011, pp. 139–146.

[29] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *SOSP*. 2007, pp. 205–220.

[30] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!)” In: *7th Symposium on Operating Systems Design and Implementation (OSDI ’06), November 6-8, Seattle, WA, USA*. 2006, pp. 205–218.

[31] Apache Cassandra. *http://cassandra.apache.org/*.

[32] Sheldon Finkelstein. “Common Expression Analysis in Database Applications”. In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’82. New York, NY, USA: ACM, 1982, pp. 235–245. ISBN: 0-89791-073-7.

[33] Timos K. Sellis. “Multiple-Query Optimization”. In: *ACM Trans. Database Syst.* 13.1 (1988), pp. 23–52.

[34] Prasan Roy et al. “Efficient and Extensible Algorithms for Multi Query Optimization”. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. 2000, pp. 249–260.

[35] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. “QPipe: A Simultaneously Pipelined Relational Query Engine”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Manage-*

ment of Data. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 383–394. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066201. URL: http://doi.acm.org/10.1145/1066157.1066201.

[36] Duy-Hung Phan and Pietro Michiardi. "A novel, low-latency algorithm for multiple Group-By query optimization". In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 2016, pp. 301–312.

[37] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins". In: *Proc. 17th International Conference on Very Large Data Bases, VLDB 1991*. 1991, pp. 537–548. URL: http://www.vldb.org/conf/1991/P537.PDF.

[38] David J. DeWitt et al. "Practical Skew Handling in Parallel Joins". In: *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.* Ed. by Li-Yan Yuan. Morgan Kaufmann, 1992, pp. 27–40. ISBN: 1-55860-151-1.

[39] Yu Xu and Pekka Kostamaa. "Efficient Outer Join Data Skew Handling in Parallel DBMS". In: *PVLDB* 2.2 (2009), pp. 1390–1396.

[40] Yu Xu et al. "Handling data skew in parallel joins in shared-nothing systems". In: *Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*. 2008, pp. 1043–1052. DOI: 10.1145/1376616.1376720.

[41] Nicolas Bruno, YongChul Kwon, and Ming-Chuan Wu. "Advanced Join Strategies for Large-Scale Distributed Computation". In: *PVLDB* 7.13 (2014), pp. 1484–1495.

[42] Long Cheng et al. "Robust and Skew-resistant Parallel Joins in Shared-Nothing Systems". In: *Proc. of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014*. 2014, pp. 1399–1408. DOI: 10.1145/2661829.2661888.

[43] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*. 2008, pp. 29–42.

[44] Faraz Ahmad et al. "Tarazu: optimizing MapReduce on heterogeneous clusters". In: *Proc. of the 17th International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems, ASPLOS 2012. 2012, pp. 61–74. DOI: 10.1145/2150976.2150984.

[45] Jiexing Li, Jeffrey F. Naughton, and Rimma V. Nehme. "Resource bricolage and resource selection for parallel database systems". In: *VLDB J.* 26.1 (2017), pp. 31–54. DOI: 10.1007/s00778-016-0435-4. URL: https://doi.org/10.1007/s00778-016-0435-4.

[46] Masahisa Tamura and Masaru Kitsuregawa. "Dynamic Load Balancing for Parallel Association Rule Mining on Heterogenous PC Cluster Systems". In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*. 1999, pp. 162–173.

[47] 安井隆宏 et al. "並列 DBMS に於ける動的負荷分散機構の実装". japanese. In: 情報処理学会研究報告データベースシステム (*DBS*) 1999.61 (July 1999), pp. 393–398. ISSN: 09196072. URL: https://ci.nii.ac.jp/naid/110002931146/.

[48] Rimma V. Nehme and Nicolas Bruno. "Automated partitioning design in parallel database systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis et al. ACM, 2011, pp. 1137–1148. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989444. URL: https://doi.org/10.1145/1989323.1989444.

[49] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan et al. ACM, 2012, pp. 61–72. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213844. URL: https://doi.org/10.1145/2213836.2213844.

[50] Christos Kozyrakis et al. "Server Engineering Insights for Large-Scale Online Services". In: *IEEE Micro* 30.4 (July 2010), pp. 8–19. ISSN: 0272-1732.

[51] Presto. Distributed SQL Query Engine for Big Data. *https://prestodb.io/*.

[52] Sergey Melnik et al. "Dremel: Interactive Analysis of Web-Scale Datasets". In: *PVLDB* 3.1 (2010), pp. 330–339.

[53] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX*

Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012. 2012, pp. 15–28.

[54] Anurag Gupta et al. "Amazon Redshift and the Case for Simpler Data Warehouses". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 2015, pp. 1917–1923.

[55] Angelo Pruscino. "Oracle RAC: Architecture and Performance". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. 2003, p. 635.

[56] Jeffrey W. Josten et al. "DB2's Use of the Coupling Facility for Data Sharing". In: *IBM Systems Journal* 36.2 (1997), pp. 327–351.

[57] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685.

[58] Donald D. Chamberlin et al. "A History and Evaluation of System R". In: *Commun. ACM* 24.10 (1981), pp. 632–646. DOI: 10.1145/358769.358784. URL: https://doi.org/10.1145/358769.358784.

[59] Gerald Held, Michael Stonebraker, and Eugene Wong. "INGRES: A Relational Data Base System". In: *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA*. Vol. 44. AFIPS Conference Proceedings. AFIPS Press, 1975, pp. 409–416. DOI: 10.1145/1499949.1500029. URL: https://doi.org/10.1145/1499949.1500029.

[60] *Oracle*. https://www.oracle.com/index.html. (Accessed on 02/07/2019).

[61] James Martin, Kathleen Kavanagh Chapman, and Joe Leben. *DB2: Concepts, Design, and Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN: 0-13-198581-7.

[62] *PostgreSQL*. https://www.postgresql.org/. (Accessed on 02/07/2019).

[63] *MySQL*. https://www.mysql.com/. (Accessed on 02/07/2019).

[64] *Firebird*. https://firebirdsql.org/. (Accessed on 02/07/2019).

[65] George P. Copeland et al. "Data Placement In Bubba". In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*. Ed. by Haran Boral and Per-Åke Lar-

son. ACM Press, 1988, pp. 99–108. DOI: 10.1145/50202.50213. URL: https://doi.org/10.1145/50202.50213.

[66] Haran Boral and Per-Åke Larson, eds. *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*. ACM Press, 1988.

[67] David J. DeWitt et al. "GAMMA - A High Performance Dataflow Database Machine". In: *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.* Ed. by Wesley W. Chu et al. Morgan Kaufmann, 1986, pp. 228–237. ISBN: 0-934613-18-4. URL: http://www.vldb.org/conf/1986/P228.PDF.

[68] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. "An Overview of The System Software of A Parallel Relational Database Machine GRACE". In: *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.* Ed. by Wesley W. Chu et al. Morgan Kaufmann, 1986, pp. 209–219. ISBN: 0-934613-18-4. URL: http://www.vldb.org/conf/1986/P209.PDF.

[69] *Teradata*. https://www.teradata.com/. (Accessed on 02/07/2019).

[70] Dieter Gawlick, Mark N. Haynie, and Andreas Reuter, eds. *High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings.* Vol. 359. Lecture Notes in Computer Science. Springer, 1989. ISBN: 3-540-51085-0. DOI: 10.1007/3-540-51085-0. URL: https://doi.org/10.1007/3-540-51085-0.

[71] "Tandem Database Group - NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL". In: *High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30, 1987, Proceedings.* Ed. by Dieter Gawlick, Mark N. Haynie, and Andreas Reuter. Vol. 359. Lecture Notes in Computer Science. Springer, 1987, pp. 60–104. ISBN: 3-540-51085-0. DOI: 10.1007/3-540-51085-0_43. URL: https://doi.org/10.1007/3-540-51085-0%5C_43.

[72] Malcolm Singh and Ben Leonhardi. "Introduction to the IBM Netezza warehouse appliance". In: *Center for Advanced Studies on Collaborative Research,*

CASCON '11, Toronto, ON, Canada, November 7-10, 2011. Ed. by Joanna W. Ng et al. IBM, 2011, pp. 385–386. URL: http://dl.acm.org/citation.cfm?id=2093965.

[73] Chuck Bear, Andrew Lamb, and Nga Tran. *The vertica database: SQL RDBMS for managing big data*. SQL RDBMS for managing big data. New York, New York, USA: ACM, Sept. 2012.

[74] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.acm.org/10.1145/356770.356776.

[75] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. "Application of Hash to Data Base Machine and Its Architecture". In: *New Generation Comput.* 1.1 (1983), pp. 63–74. DOI: 10.1007/BF03037022. URL: https://doi.org/10.1007/BF03037022.

[76] David J. DeWitt et al. "Implementation Techniques for Main Memory Database Systems". In: *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*. Ed. by Beatrice Yormark. ACM Press, 1984, pp. 1–8. DOI: 10.1145/602259.602261. URL: https://doi.org/10.1145/602259.602261.

[77] 喜連川優, 合田和生. "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". japanese. In: 日本データベース学会論文誌 8.1 (2009), pp. 131–136.

[78] 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". japanese. In: 電子情報通信学会論文誌 *D* J97-D.4 (Apr. 2014), pp. 729–737.

[79] The Transaction Processing Performance Council. *http://www.tpc.org/*.

[80] Wesley W. Chu et al., eds. *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*. Morgan Kaufmann, 1986. ISBN: 0-934613-18-4.

発表文献

査読付き論文誌

- 奥野晃裕，早水悠登，合田和生，喜連川優. 共有ストレージ型データベースエンジンに於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 (TOD78) Vol.11 No.2, 2018 年 7 月
- 奥野晃裕，早水悠登，合田和生，喜連川優. 共有ストレージ型並列データベースエンジンにおける演算資源非均質性の影響を軽減可能とする動的負荷分散手法. 電子情報通信学会論文誌 （投稿済み）

査読付き国内学会

- 奥野晃裕，早水悠登，合田和生，喜連川優. 共有ストレージ型データベースエンジンに於ける動的演算資源調整手法の提案. 第 11 回 Web とデータベースに関するフォー ラム (WebDB Forum 2018)，2018 年 9 月

その他

- 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. 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. 117， No. 374，DE2017-25，pp.7-12，2017 年 12 月
- 奥野晃裕，早水悠登，合田和生，喜連川優. 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. 第 10 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 16 回日本 データベー

ス学会年次大会 (DEIM2018)，C6-2, 2018 年 3 月

- 奥野晃裕，早水悠登，合田和生，喜連川優. パブリッククラウド環境を用いた動的演算資源調整手法の実行性能への影響機序の分析. 第 11 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 16 回日本 データベース学会年次大会 (DEIM2019)，2019 年 3 月