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(TOD), Vol. 11, No. 2, pp.30-48, 2018.

情報処理学会論文誌　データベース　Vol.11　No.2　30–43 (July 2018)

共有ストレージ型データベースエンジンにおける動的演算資源調整手法の提案

奥野 晃裕[1,a)]　早水 悠登[1]　合田 和生[1]　喜連川 優[1,2]

受付日 2017年12月10日, 採録日 2018年4月4日

概要：広く用いられている共有ストレージ型データベースエンジンでは，演算資源が 1 つのストレージを共有することで，演算資源における故障に対するデータベースエンジンの可用性を高めるとともに，演算資源によってクエリのスループットが律速される場合には，クエリ実行開始時に演算資源量を指定することによって当該クエリのスループットを変更することができる．しかしながら，従来の共有ストレージ型データベースエンジンではクエリの実行開始時に利用する演算資源が決定されるため，クエリ実行に割り当てる演算資源の調整の機会はクエリ実行前に限られていた．本論文では，共有ストレージ型データベースエンジンにおいて演算資源の割当てをクエリの実行時に調整する手法を提案する．当該手法を用いることにより，たとえば，あるクエリの実行中に他のクエリの実行が完了して新たに演算資源が利用可能になる場合においても，当該実行中クエリに対して追加して演算資源を割り当てることが可能となるなど，従来手法より効率の良い演算資源の利用を実現することができる．本論文では，パブリッククラウド環境を用いた評価実験により，提案手法の有効性を明らかにする．

キーワード：データベースエンジン，動的資源調整

Dynamic Computing Resource Adjustment in Shared-storage Database Engine

Akihiro Okuno[1,a)]　Yuto Hayamizu[1]　Kazuo Goda[1]　Masaru Kitsuregawa[1,2]

Received: December 10, 2017, Accepted: April 4, 2018

Abstract: Shared-storage database engine, which is one of the major architectures of the database engine, has been widely used. Shared-storage database engine achives high availability against a fault in the computing resources by sharing a storage among computing resources. With shared-storage database engine, for a query where the throughput is limited by the computing resources, a query throughput can be controlled by specifying the amount of computing resources. However, in a conventional shared-storage database engine, the amount of computing resources is determined at the beginning of query execution, thus the opportunity of specifying the amount of computing resources used for query execution is limited before the execution of the query. In this paper, we propose a method to adjust the amount of computing resources to be utilized for query execution during query execution. The proposed method improves the efficiency of computing resource utilization in shared-storage database engine. For example, the proposed method allows the shared-storage database engine to allocate the available resources to the currently executing query even if the resources were not available before the execution of the query. We present experimental evaluations using the public cloud environment and clarify the effectiveness of the proposed method.

Keywords: database engine, dynamic resource adjustment

1. はじめに

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 [1]，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，さら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

[1] 東京大学生産技術研究所
Institute of Industrial Science, the University of Tokyo, Meguro, Tokyo 153–8503, Japan
[2] 国立情報学研究所
National Institute of Informatics, Chiyoda, Tokyo 101–8403, Japan
[a)] okuno@tkl.iis.u-tokyo.ac.jp

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 [2], [3], [4], [5], [6]．

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，さかんに研究の対象となっているだけではなく [7], [8], [9], [10]，産業界においても商用実装されて広く用いられている [11], [12], [13]．

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，この際のクエリごとのユーザの性能要求も様々である．たとえば，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．このよう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ごとにユーザからの性能要求に基づく優先度に基づき，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り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り当てる機会はクエリ実行開始時点に限られている．そのため，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，すで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．

本論文では，共有ストレージ型データベースエンジンにおいて実行中のクエリに割り当てる演算資源を動的に調整する手法を提案する．たとえば，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り当てることにより，当該クエリの実行を加速することが可能となる．また，演算資源がすべて実行中の他のクエリに割り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．本論文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，実行中クエリに割り当てる演算資源を動的に調整することにより，従来より効率良く演算資源を利用し，よりクエリごとの優先度に応じた実行が期待される．著者らの知る限り，同様の研究はほかに見当たらない．著者らは文献 [14] において，動的演算資源調整の基本構想を示し，演算資源の追加ならびに削除のプロトコル，および実験により単一クエリに対して 4 インスタンスまでの動的演算資源調整の性能スケーラビリティを示した．それに対し，本論文では実験の規模を 12 インスタンスまで拡張するとともに，複数のクエリに対して動的演算資源を行うケーススタディの結果を示すことで提案手法の有効性を示す．本論文では，パブリッククラウド環境において提案手法の試作実装を用いた評価実験の結果を示し，提案手法の有効性を明らかにする．

本論文の構成は次のとおりである．2 章ではデータベースエンジンにおける共有ストレージ型のアーキテクチャについて述べる．3 章では共有ストレージ型データベースエンジンにおける動的演算資源調整の手法を示す．4 章ではパブリッククラウド環境において提案手法に基づく評価実験の結果を示し，当該手法の有効性を明らかにする．5 章では関連する研究および本論文との関係を示し，6 章では本論文のまとめ，および今後の展望を述べる．

2. 共有ストレージ型データベースエンジン

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の 1 つに共有ストレージ型データベースエンジンがあげられる．共有ストレージ型データベースエンジンは，複数の演算資源がネットワークを介して 1 つの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データベースエンジンの可用性を高めることができるほか，すべての演算資源が共通のストレージを参照するため，ストレージに対する負荷分散を行いやすいなどの利点がある [15], [16]．共有ストレージ型データベースエンジンは，産業界においても Oracle RAC [11] や IBM DB2 [12] などの商用実装に採用されるなど，データベースエンジンにおけるアーキテクチャの 1 つとして広く用いられている．

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[*1]，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. 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

3.1 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，本論文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づいて，実行中クエリへの演算資源の割当てを調整し，より効率の良い演算資源の利用や，ユーザの要求に基づくクエリごとの優先度に

[*1] 共有ストレージ型とは異なるデータベースエンジンのアーキテクチャとして無共有型データベースエンジンがあげられる．無共有型データベースエンジンに動的演算資源調整を適用する際には，データマイグレーションによるオーバヘッドが課題となることが推察される．無共有型データベースエンジンにおける動的演算資源調整については今後の課題とする．

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り当てることにより，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すべて割り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り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，当該演算資源を割り当てることで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．

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

図 1 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 1 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 1 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づいて実行計画を作成し，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．本論文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り当てる．

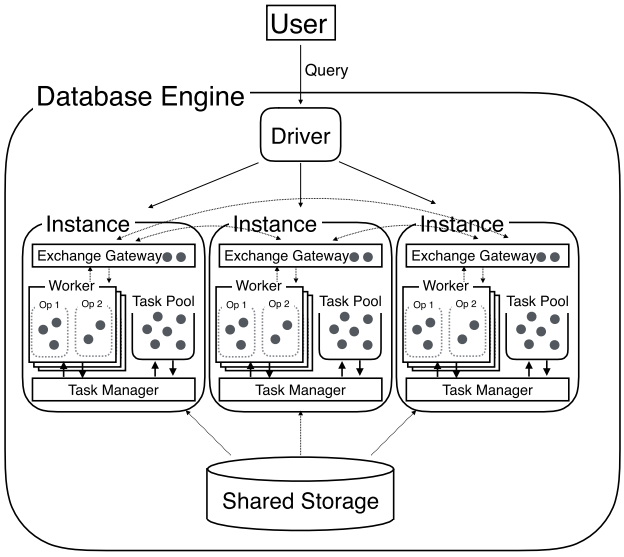

図 1　動的演算資源調整を可能とする共有ストレージ型データベースエンジンの概要

Fig. 1　Overview of database engine with dynamic resource adjustment mechanism.

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．そのため，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，すなわ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．たとえば，スキャン演算および選択演算を含む段を割り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．本論文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，ワーカごと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 2** 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．すなわ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

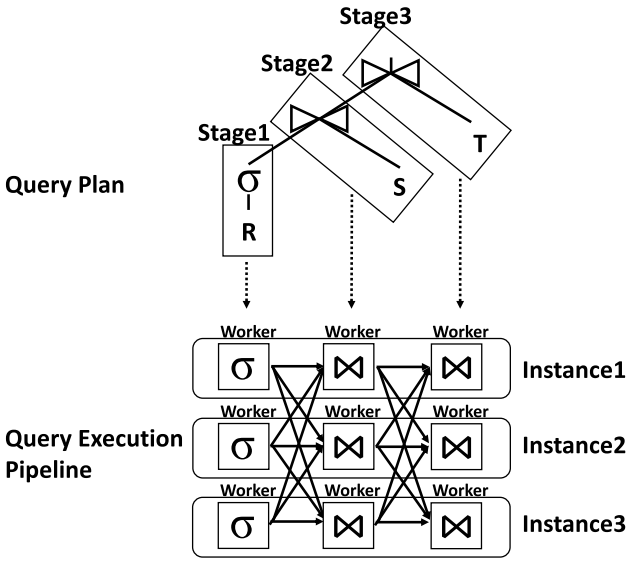

図 2　クエリ（$\sigma(R) \bowtie S \bowtie T$）の実行計画とクエリ実行におけるパイプラインの例

Fig. 2　An example of a query plan and a query execution pipeline.

して得られたタプルは即座に送信対象先へと送られる，いわゆるプッシュ型の方式によって後段ワーカへと送信される．たと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つねに駆動され続ける．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 2 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つねに駆動され続け，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 のタプルごとに処理が駆動され，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，受信したタプルごとに処理が駆動され，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 段目のワーカがすべ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．

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

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よって，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，クエリ実行に割り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．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づいて正しく処理された後に，後段のワーカへとすべて送らなければならない．すなわち，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すべ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．

Algorithm 1 に演算資源の追加，すなわ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が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．Algorithm 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

Algorithm 1 i 段目のワーカを追加する方法

Input: i: 起動するワーカの段
Input: num: 起動するワーカ数
1: **function** AddWorkers(i, num)
2:　$targets \leftarrow$ choose num from available instances
3:　$prevWorkers \leftarrow$ workers of stage $i-1$
4:　$nextWorkers \leftarrow$ workers of stage $i+1$
5:　**for** $t \leftarrow targets$ **do**
6:　　start worker of stage i on t
7:　　open connections from worker on t to $nextWorkers$
8:　**end for**
9:　$newWorkers \leftarrow$ workers on $targets$
10:　**for** $p \leftarrow prevWorkers$ **do**
11:　　open connections from p to $newWorkers$
12:　　start to send data from p to $newWorkers$
13:　**end for**
14: **end function**

Algorithm 2 i 段目のワーカを停止する方法

Input: i: 停止するワーカの段
Input: num: 停止するワーカ数
1: **function** RemoveWorkers(i, num)
2: $targets \leftarrow$ choose num from workers of stage i
3: $prevWorkers \leftarrow$ workers of stage $i-1$
4: $nextWorkers \leftarrow$ workers of stage $i+1$
5: **for** $p \leftarrow prevWorkers$ **do**
6: stop sending data from p to $targets$
7: close connections from p to $targets$
8: **end for**
9: **for** $t \leftarrow targets$ **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**

正しく受信し処理可能であることを可能としている．

Algorithm 2 に演算資源の削除，すなわ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．Algorithm 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すべて受信するまで待ち，受信したすべてのデータを実行計画に基づ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すべ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のっと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．すなわち，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，すなわ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．

4. パブリッククラウド環境を用いた評価実験

4.1 動的演算資源調整機構を有するデータベースエンジンの実装

著者らは，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，当該手法を実装し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，ユーザから演算資源の追加・削除を開始できるようにした．ドライバに対し演算資源の追加・削除コマンドを実行すると，あるインスタンスでのすべてのワーカを起動・停止し，インスタンスの単位で動的演算資源調整が行われる．

当該試作においては，ドライバがユーザからクエリを受け取り実行計画を作成すると，各インスタンスにおいて当該実行計画中の各段に対応するワーカがちょうど 1 つずつ実行されるようにワーカを配置した．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，すべ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．たとえば，図 5 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，3.3 節で述べたように隣接する段のワーカを同時に起動・停止しなければよいが，本試作においては，実装の簡潔さから後段のワーカから 1 つずつ順に起動・停止する方式を採用した．なお，3.3 節に述べた制約の範囲内で複数のワーカの起動・停止を実施することによりさらなる実行効率の改善が見込まれるが，本論文の対象範囲を超えることからその考察は別稿に譲る．

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づ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．そのため，以下の実験においてはワーカの実行に割り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）を用いて実験環境を構築した，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 1** に示す．

評価実験用のデータセットとして，TPC-H [17] の dbgen を用いて，$ScaleFactor = 100$ 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 2 次索引を作成した．

1 つのワーカで利用可能なスレッド数として，スキャン演算を実行するワーカでは 64 スレッド，結合演算を実行するワーカでは 1,024 スレッドとした．

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エリ Q1（**図 3**）と，part，lineitem の 2 表の結合演算を行うクエリ Q2（**図 4**）の，2 つのクエリを用いた．Q1，Q2 のプランツリーを**図 5** 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．

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

Instance: EC2 c4.8xlarge	
CPU	36 vCPU
Memory	60 GiB
OS	Amazon Linux 64-bit (hvm)
Hardware	Shared (Default Tenancy)
Network Bandwidth	10 Gbps
Network Location	Colocated (Placement Group)
Worker	
Number of Threads (Scan)	64
Number of Thread (Join)	1,024

```
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 3　評価実験に用いたクエリ：Q1
Fig. 3　Query for evaluation experiments: 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　評価実験に用いたクエリ：Q2
Fig. 4　Query for evaluation experiments: Q2.

4.3　演算資源量に対するスケーラビリティ

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り当てられたインスタンス数に対する実行時間を計測した．各々のクエリの実行時間を**図 6**，**図 8** に，またインスタンス数が 1 のときの実行時間に対する性能向上率を**図 7**，**図 9** にそれぞれ示す．インスタンス数 16 のときにインスタンス数 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を確認した．

4.4　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り当てる演算資源を調整可能であることを示

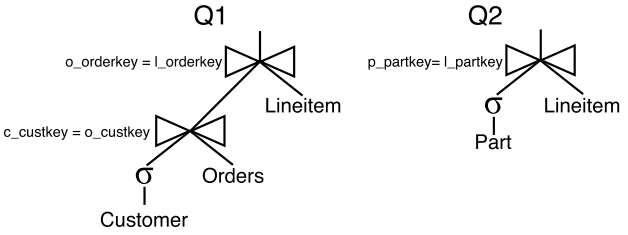

図 5　評価実験に用いたクエリのプランツリー
Fig. 5　Plan trees of queries for evaluation experiments.

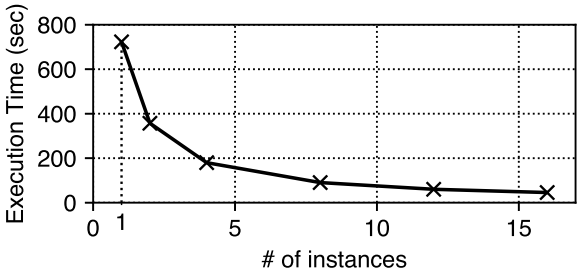

図 6　クエリ Q1 の実行時間
Fig. 6　Execution time of Q1.

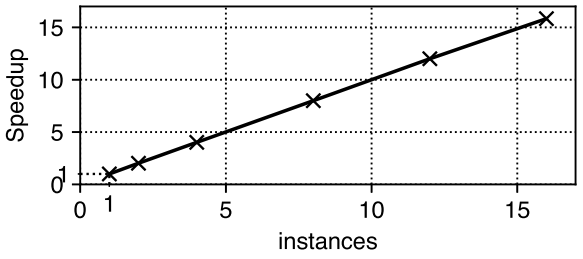

図 7　クエリ Q1 の性能向上率
Fig. 7　Speedup of Q1.

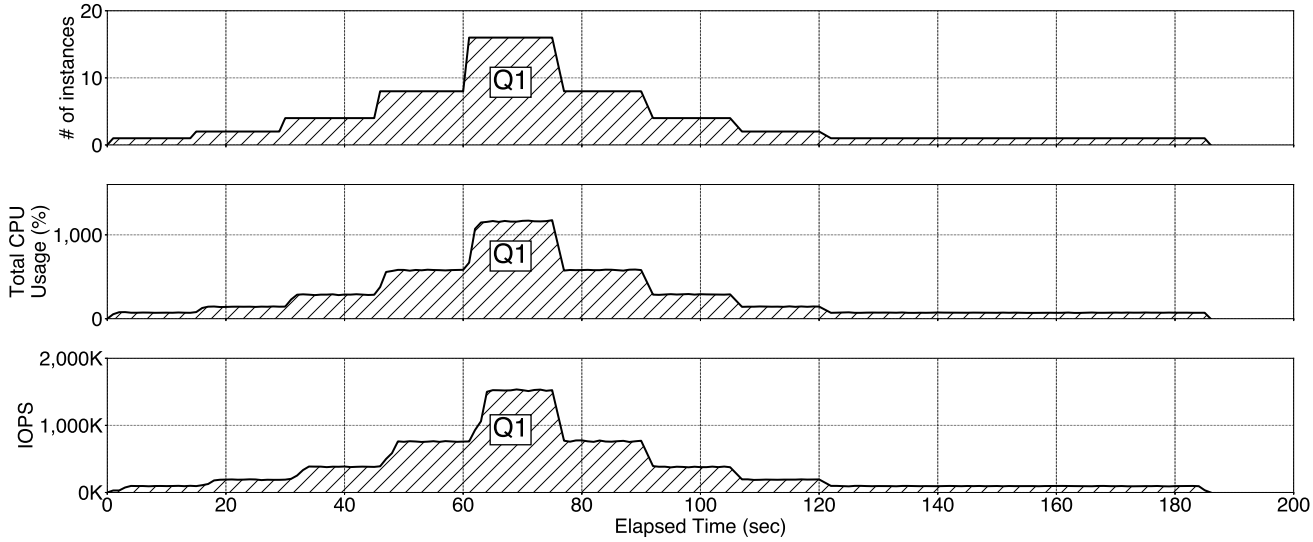

図 10　クエリ Q1 に対する動的演算資源調整
Fig. 10　Dynamic resource adjustment to Q1.

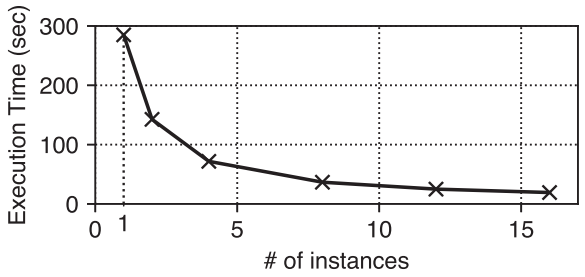

図 8　クエリ Q2 の実行時間
Fig. 8　Execution time of Q2.

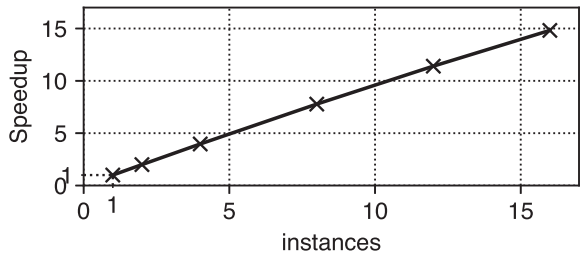

図 9　クエリ Q2 の性能向上率
Fig. 9　Speedup of Q2.

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，すべ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 秒ごと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 まで減らす操作を行った．結果を**図 10** 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り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．

表 2　クエリ Q1 で動的演算資源調整を行ったときの IOPS 性能向上率
Table 2　Speedup of Q1 while adjusting computing resources.

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

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，インスタンス数が 1 のときの平均 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 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わずかに下回った．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ヘッドによるものと考えられる．図 10 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り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．

4.5　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

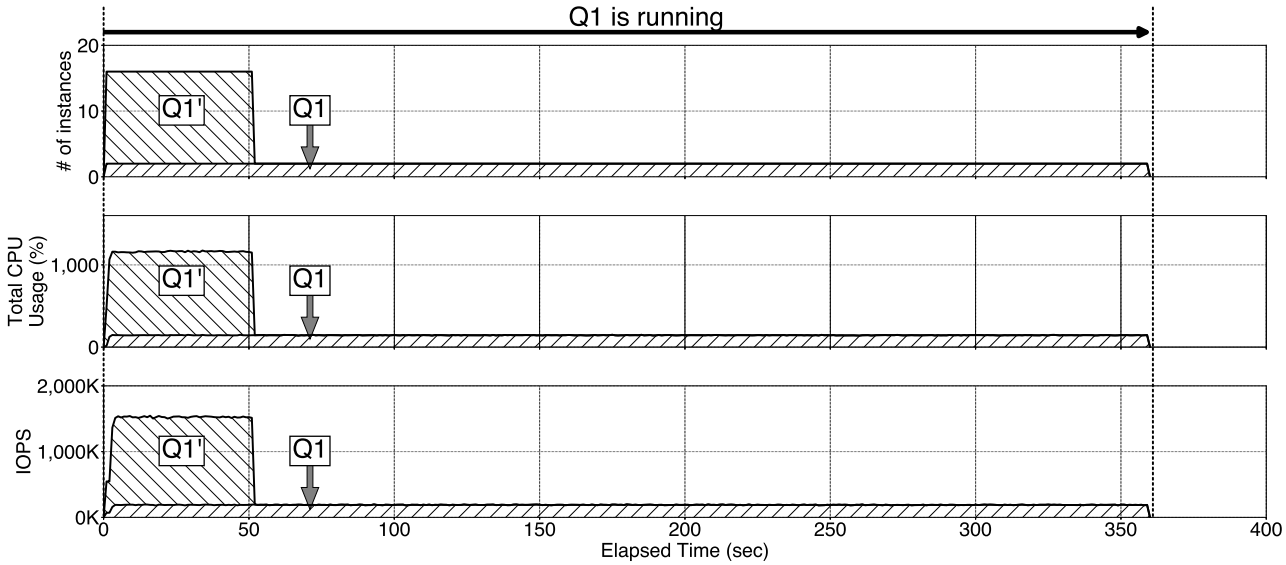

図 11　余剰演算資源を用いた実行中クエリの加速：動的演算資源調整なし
Fig. 11　Concurrent query execution with different priorities. w/o Dynamic resource adjustment.

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に対しても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．本論文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，各クエリ実行に割り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資源調整を行うものとする．また，データベースエンジン全体が利用可能なインスタンス数は最大で 16 とする．

1 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り当てられ，演算資源が多く割り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り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，すでに実行を開始している低優先度クエリは実行に割り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．

本論文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り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，その時点での利用しているインスタンス数，すべ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り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 11** 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 に割り当てた場合の結果を**図 12** 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’ のみとなり動的演算資源調整によって Q1’ にすべてのインスタンスが割り当てられるため，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 インスタンスが割り当てられた．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り当てることにより，当該実行中クエリを加速することが可能であることを確認した．

4.6　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

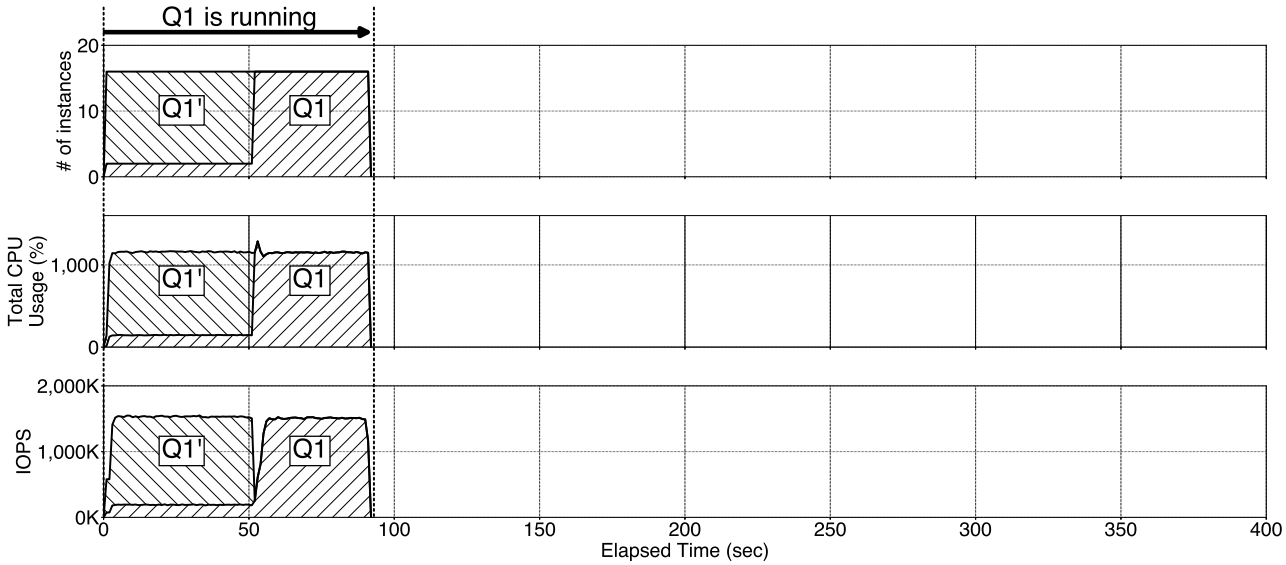

図 12　余剰演算資源を用いた実行中クエリの加速：動的演算資源調整あり

Fig. 12　Concurrent query execution with different priorities. w/ Dynamic resource adjustment.

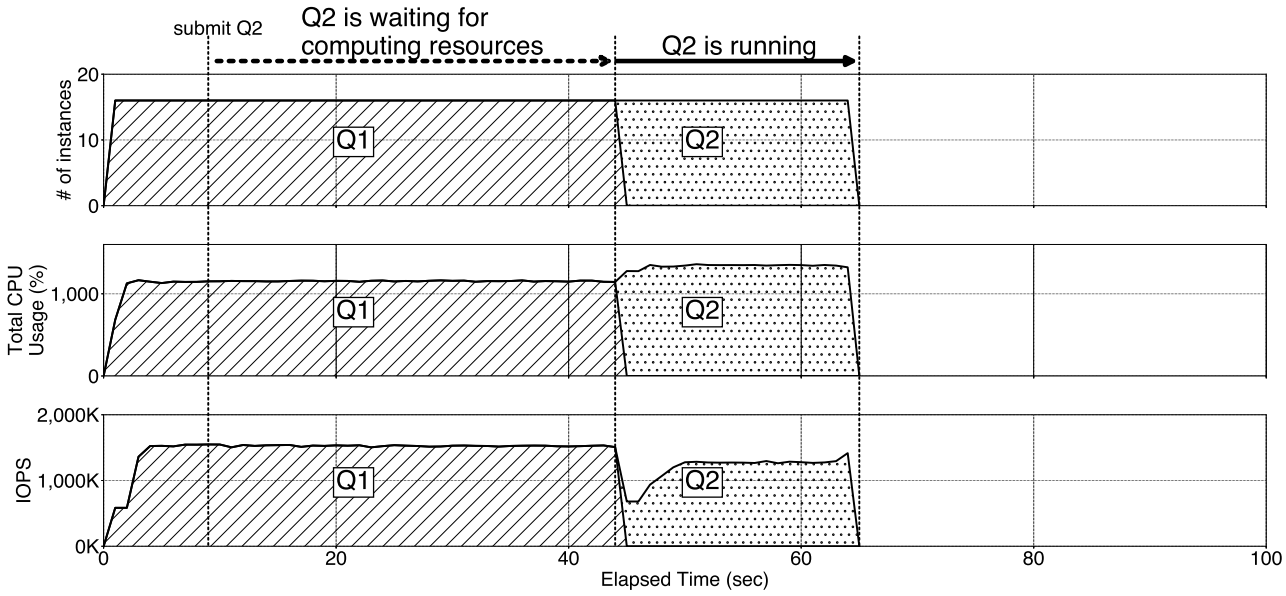

図 13　高優先度クエリの割り込み：動的演算資源調整なし

Fig. 13　Interruption by high priority query. w/o Dynamic resource adjustment.

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．本論文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，すべ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 **13** に，提案手法を用いて，Q2 のクエリ実行開始を要求したときに 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 **14** 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 インスタンスを割り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かかわらず，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

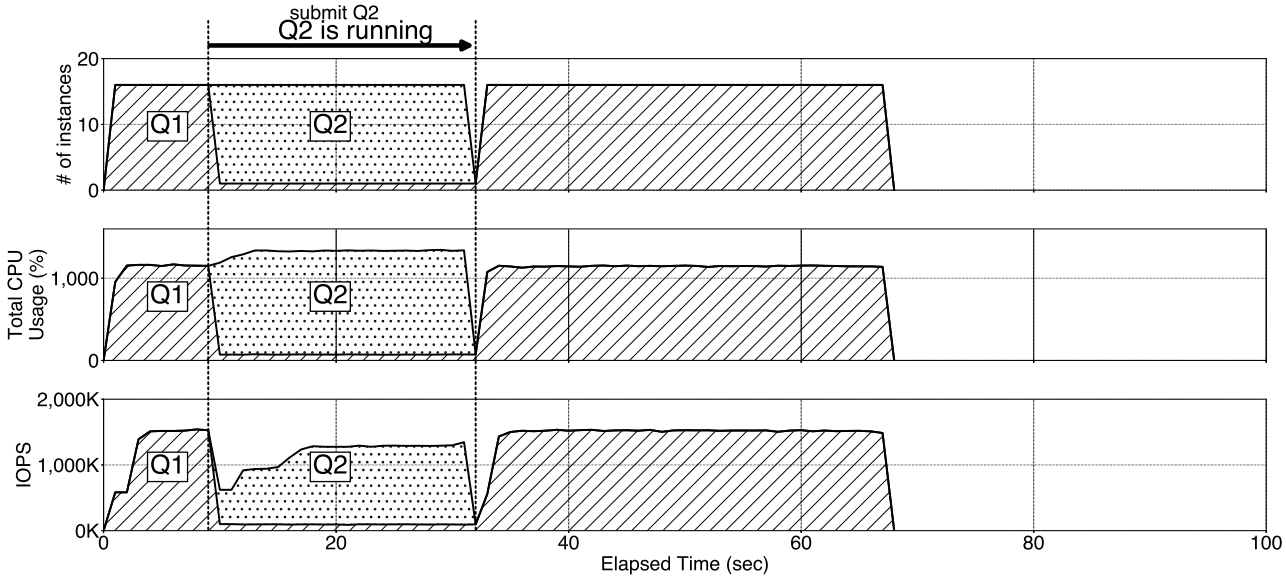

図 14　高優先度クエリの割り込み：動的演算資源調整あり
Fig. 14　Interruption by high priority query. w/ Dynamic resource adjustment.

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．すなわち，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，割り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．

4.7　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り当てる演算資源を調整し，たとえば優先度の高いクエリは多くの資源を割り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り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，よりクエリごと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，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，すべ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 15** に示す．Q1，Q2，Q3 に設定された優先度はすべて 1 であるため，各クエリは 5 インスタンスずつ割り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 に割り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 インスタンスを割り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 インスタンスを割り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り当てて開始することができた．また，

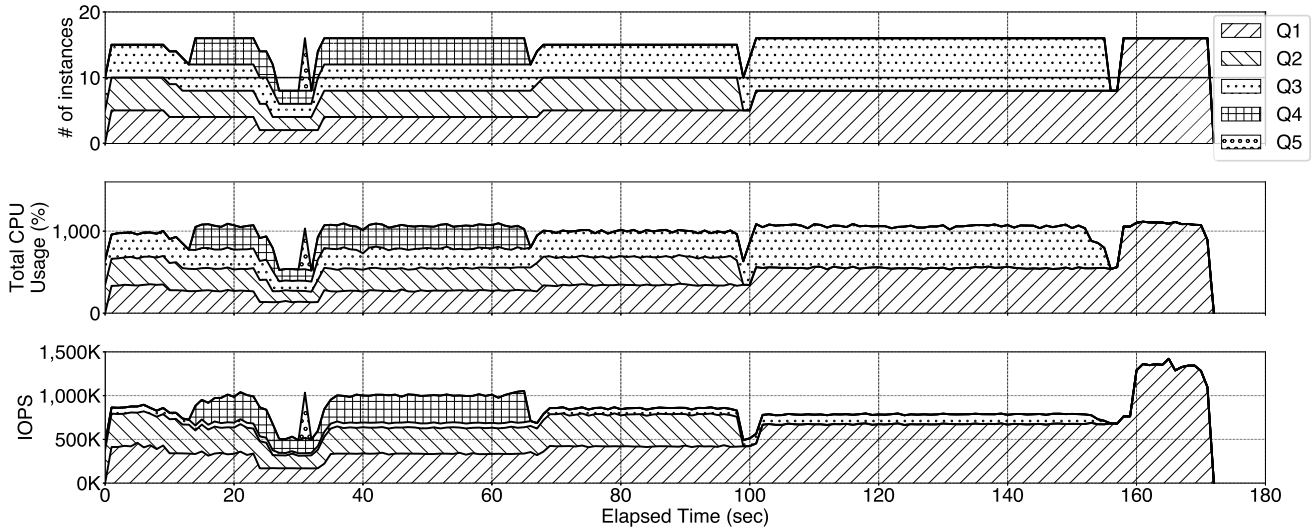

図 15　多数のクエリでの動的演算資源調整

Fig. 15　Dynamic computing resource adjustment for multiple queries.

IOPS についても，クエリごと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の，クエリごとの IOPS については割り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り当てられていないインスタンスが存在した．たとえば，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り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，どのクエリにも割り当てられていないものが最大 8 インスタンス存在した．演算資源の調整にともな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良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づいて調整可能であることを示した．

5. 関連研究

Volcano [18] 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，グループごと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．

また，3.2 節で示した共有ストレージ型データベースエンジンの実行方式は，1 タプルに対する演算からタスクを生成し，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ものであり，当該実行方式はアウトオブオーダ型実行方式 [19], [20] に基づいている．本論文はアウトオブオーダ型実行方式で実行されているクエリに対して動的に演算資源の調整を行う手法を提案する．

近年では，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さかんに行われており [7], [8], [9], [10], [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 [21] 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 [22] などが提案されている．本研究は，共有ストレージ型データベースエンジンにおいてクエリの実行時に動的に演算資源を調整することを目的としており，これらの研究とは目的を異とする．

データベースエンジンにおける共有ストレージ型以外のアーキテクチャとして無共有型データベースエンジンがあげられる．無共有型データベースエンジンはストレージを共有せずに演算資源ごと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 [15], [16]．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 [2] 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 [23], [24], [25] 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 [26], [27] が提案されている．無共有型データベースエンジンにおける動的演算資源調整の適用は今後の課題とする．

演算資源を処理の実行中に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動的資源調整を行う研究 [28], [29], [30] や，MapReduce [31] 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 [32], [33], [34] 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ものである．

データベースエンジンにおける複数クエリの実行については，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 [35], [36], [37] が古くから提案されているほか，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 [38] や，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手法 [39] 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ごと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．

6. おわりに

本論文では，共有ストレージ型データベースエンジンにおいて，実行中のクエリに割り当てる演算資源を動的に調整する手法を提案した．パブリッククラウド環境において，提案手法を実装したデータベースエンジンを用いた評価実験を行った結果，16 ノードで最大 15.8 倍の性能向上が得られた．また，クエリ実行中に新たに演算資源が利用可能となる場合や，演算資源の空きがない状況で高優先度のクエリ実行が要求される場合などにおいて，従来よりクエリ完了までの時間を短縮することが可能となることを明らかにし，さらに異なる優先度の複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効率良く演算資源を利用可能となることを示した．これらの結果により，提案手法を用いることにより，従前の共有ストレージ型データベースエンジンでは不可能であったクエリ実行中の資源調整を可能とし，よりユーザのサービスレベル要求に即したクエリの実行が実現できることを示した．今後は，より多数のクエリが随時実行される環境において各クエリの優先度に基づいた自律的な演算資源の割当て手法に取り組むとともに，無共有型データベースエンジンにおける動的演算資源調整の研究を進めていきたい．

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

参考文献

[1] Kozyrakis, C., Kansal, A., Sankar, S. and Vaid, K.: Server Engineering Insights for Large-Scale Online Services, *IEEE Micro*, Vol.30, No.4, pp.8–19 (2010).
[2] Apache Hadoop, available from ⟨http://hadoop.apache.org/⟩.
[3] Presto: Distributed SQL Query Engine for Big Data, available from ⟨https://prestodb.io/⟩.
[4] Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M. and Vassilakis, T.: Dremel: Interactive Analysis of Web-Scale Datasets, *PVLDB*, Vol.3, No.1, pp.330–339 (2010).
[5] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation*, *NSDI 2012*, pp.15–28 (2012).
[6] Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S. and Srinivasan, V.: Amazon Redshift and the Case for Simpler Data Warehouses, *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pp.1917–1923 (2015).
[7] Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., Littlefield, K. and Tong, P.: F1: The fault-tolerant distributed RDBMS supporting google's ad business, *Proc. ACM SIGMOD International Conference on Management of Data*, *SIGMOD 2012*, pp.777–778 (2012).
[8] Das, S., Agrawal, D. and El Abbadi, A.: ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud, *ACM Trans. Database Syst.*, Vol.38, No.1, pp.5:1–5:45 (2013).
[9] Brantner, M., Florescu, D., Graf, D.A., Kossmann, D. and Kraska, T.: Building a database on S3, *Proc. ACM SIGMOD International Conference on Management of Data*, *SIGMOD 2008*, pp.251–264 (2008).
[10] Levandoski, J.J., Lomet, D.B., Mokbel, M.F. and Zhao, K.: Deuteronomy: Transaction Support for Cloud Data, *Proc. 5th Biennial Conference on Innovative Data Systems Research*, *CIDR 2011*, pp.123–133 (2011).
[11] Pruscino, A.: Oracle RAC: Architecture and Performance, *Proc. 2003 ACM SIGMOD International Conference on Management of Data*, p.635 (2003).
[12] Josten, J.W., Mohan, C., Narang, I. and Teng, J.Z.: DB2's Use of the Coupling Facility for Data Sharing, *IBM Systems Journal*, Vol.36, No.2, pp.327–351 (1997).
[13] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T. and Bao, X.: Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases, *SIGMOD Conference*, pp.1041–1052 (2017).

[14] 奥野晃裕，早水悠登，合田和生，喜連川優：クラウド環境に於けるクエリ実行時の資源調整機構を備えた高速データベースエンジンの試作に関する一考察，信学技報，DE2017-25, Vol.117, No.374, pp.7–12 (2017).
[15] Stonebraker, M.: The Case for Shared Nothing, *IEEE Database Eng. Bull.*, Vol.9, No.1, pp.4–9 (1986).
[16] DeWitt, D.J. and Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, *Comm. ACM*, Vol.35, No.6, pp.85–98 (1992).
[17] The Transaction Processing Performance Council, available from ⟨http://www.tpc.org/⟩.
[18] Graefe, G. and McKenna, W.J.: The Volcano Optimizer Generator: Extensibility and Efficient Search, *Proc. 9th International Conference on Data Engineering*, pp.209–218 (online), DOI: 10.1109/ICDE.1993.344061 (1993).
[19] 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，日本データベース学会論文誌，Vol.8, No.1, pp.131–136 (2009).
[20] 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，電子情報通信学会論文誌 D，Vol.J97-D, No.4, pp.729–737 (2014).
[21] Pilman, M., Bocksrocker, K., Braun, L., Marroquin, R. and Kossmann, D.: Fast Scans on Key-Value Stores, *PVLDB*, Vol.10, No.11, pp.1526–1537 (2017).
[22] Loesing, S., Pilman, M., Etter, T. and Kossmann, D.: On the Design and Scalability of Distributed Shared-Data Databases, *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pp.663–676 (2015).
[23] Zamanian, E., Binnig, C. and Salama, A.: Locality-aware Partitioning in Parallel Database Systems, *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pp.17–30 (2015).
[24] Curino, C., Zhang, Y., Jones, E.P.C. and Madden, S.: Schism: A Workload-Driven Approach to Database Replication and Partitioning, *PVLDB*, Vol.3, No.1, pp.48–57 (2010).
[25] Quamar, A., Kumar, K.A. and Deshpande, A.: SWORD: Scalable workload-aware data placement for transactional workloads, *Proc. Joint 2013 EDBT/ICDT Conferences*, *EDBT '13*, pp.430–441 (2013).
[26] Chu, S., Balazinska, M. and Suciu, D.: From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System, *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pp.63–78 (online), DOI: 10.1145/2723372.2750545 (2015).
[27] Trummer, I. and Koch, C.: Parallelizing Query Optimization on Shared-Nothing Architectures, *PVLDB*, Vol.9, No.9, pp.660–671 (2016).
[28] 合田和生，田村孝之，小口正人，喜連川優：SAN 結合 PC クラスタにおけるストレージ仮想化機構を用いた動的負荷分散並びに動的資源調整の提案とその評価，電子情報通信学会論文誌 D-I，Vol.87, No.6, pp.661–674 (2004).
[29] Goda, K., Tamura, T., Oguchi, M. and Kitsuregawa, M.: 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, *Proc. Database and Expert Systems Applications, 13th International Conference*, *DEXA 2002*, pp.182–192 (2002).
[30] Oguchi, M. and Kitsuregawa, M.: Runtime Data Declustering over SAN-Connected PC Cluster System, *Proc. 18th International Conference on Data Engineering*, p.275 (2002).
[31] Groot, S., Goda, K. and Kitsuregawa, M.: Towards improved load balancing for data intensive distributed computing, *Proc. 2011 ACM Symposium on Applied Computing* (*SAC*), pp.139–146 (2011).
[32] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store, *SOSP*, pp.205–220 (2007).
[33] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.: Bigtable: A Distributed Storage System for Structured Data, *7th Symposium on Operating Systems Design and Implementation* (*OSDI '06*), pp.205–218 (2006).
[34] Apache Cassandra, available from ⟨http://cassandra.apache.org/⟩.
[35] Finkelstein, S.: Common Expression Analysis in Database Applications, *Proc. 1982 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '82*, pp.235–245 (1982).
[36] Sellis, T.K.: Multiple-Query Optimization, *ACM Trans. Database Syst.*, Vol.13, No.1, pp.23–52 (1988).
[37] Roy, P., Seshadri, S., Sudarshan, S. and Bhobe, S.: Efficient and Extensible Algorithms for Multi Query Optimization, *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pp.249–260 (2000).
[38] Harizopoulos, S., Shkapenyuk, V. and Ailamaki, A.: QPipe: A Simultaneously Pipelined Relational Query Engine, *Proc. 2005 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '05*, pp.383–394 (online), DOI: 10.1145/1066157.1066201 (2005).
[39] Phan, D. and Michiardi, P.: A novel, low-latency algorithm for multiple Group-By query optimization, *32nd IEEE International Conference on Data Engineering*, *ICDE 2016*, pp.301–312 (2016).

奥野 晃裕

2008 年東京工業大学生命理工学部生命科学科卒業．2010 年同大学大学院修士課程修了．2012 年株式会社スケールアウトに入社．データ分析基盤の開発に従事．2017 年東京大学大学院情報理工学系研究科電子情報学専攻博士課程単位取得満期退学．現在，東京大学生産技術研究所特任研究員．データベースシステムに関する研究に従事．

早水 悠登　（正会員）

2009 年東京大学工学部電子情報工学科卒業．2014 年同大学大学院情報理工学系研究科電子情報学専攻博士課程単位取得満期退学．同年博士（情報理工学）．日本学術振興会特別研究員 DC2 を経て，現在，東京大学生産技術研究所特任助教．データベースシステムに関する研究に従事．日本データベース学会会員．

合田 和生　（正会員）

2000 年東京大学工学部電気工学科卒業．2005 年同大学大学院情報理工学系研究科電子情報学専攻博士課程単位取得満期退学．同年博士（情報理工学）．現在，東京大学生産技術研究所特任准教授．データベースシステム，ストレージシステムの研究に従事．電子情報通信学会，日本データベース学会，ACM，IEEE，USENIX 各会員．

喜連川 優　（正会員）

1983 年東京大学大学院工学系研究科情報工学専攻博士課程修了，工学博士．東京大学生産技術研究所教授．2002 年日本データベース学会理事，2013 年 4 月国立情報学研究所所長，2013 年 6 月～2015 年 5 月まで本会会長．データベース工学の研究に従事．本会功績賞，電子情報通信学会業績賞，ACM SIGMOD E.F. Codd Innovations Award，紫綬褒章受章，C&C 賞受賞．電子情報通信学会，ACM，IEEE 各フェロー．本会フェロー．

（担当編集委員　吉田 稔）