平成 27 年度　博士学位論文

リアルタイムデータアクセス処理機構の最適化

お茶の水女子大学大学院
人間文化創成科学研究科　理学専攻

榎　美紀

平成 28 年 3 月

要旨

近年，様々なデバイスから位置情報・気象・ソーシャル・センサーデータなど，膨大な量のデータが日々発信され続けており，地球上で生成されるデータは 2013 年から 2020 年の間で 4 兆 4,000 億ギガバイトから 44 兆ギガバイトへと 10 倍の規模に拡大すると予想されている．このような膨大なデータを利用して関連性を見つけたり，変化を検知したりと，様々な観点から分析する要求は年々高まっており，それらを実現するための処理基盤は大規模なデータをリアルタイムに，また高速に扱えることが重要となる．このようなストリームデータをリアルタイムに分析するため，2000 年以降ストリームデータ処理を主にした DSMS (Data Stream Management System)が登場した．従来の DBMS (DataBase Management System) と異なり，連続的に発信されるデータを対象にリアルタイムにデータ加工や分析結果を返し続けるシステムである．

従来の DBMS は，データがあらかじめデータベースに格納されている状態から，そのデータを対象に SQL によるクエリが発行され，所望のデータが返されるアーキテクチャである．発行されてくるクエリのパターンは様々である．一方，DSMS は流れてくるストリームデータに対して，連続的にクエリが実行される．データへの問合せ部分に着目した時，あらかじめ指定したウィンドウと呼ばれる単位にデータを区切り，その範囲内に到着したデータに対してクエリの演算を施すという特徴がある．また，ストリームデータはリアルタイムにデータが発信されてきており，その流量が時間や何かのタイミングで変化する場合は，常に一定の性能を保って処理し続けるシステムを保証することは難しい．そこで，インプットデータは，ランダムにサンプリングする等のフィルタリングを実施する．

DSMS で演算処理された後のデータは，一般的に破棄されていた．それは現在の状態における何らかの演算結果が重要であり，ストリーム処理システムはその

ような要件を満たすためのシステムであるからである．また，ストリームデータはリアルタイムに到着し続けるため，全データをストレージに格納することは現実的ではなかった．ところが，近年のストレージの格納量の大規模化，SSD 等によるデータ入出力処理の高速化にも伴い，ストリームデータもそのままストレージに格納され，それをオフラインで分析対象にすることも増えてきた．しかしながら，ストリームデータのリアルタイムの分析とストレージに格納したオフラインでの分析を同時に考えたシステムは検討されていない．

そこで本論文では，リアルタイムなストリームデータを**逐次的にストリーム処理する機構**と，**蓄積されたデータを処理する機構**を兼ね備えたリアルタイム分析システムを提案する．ストリームデータのリアルタイム性の収束のタイミングを議論し，リアルタイム分析とオフライン分析との使い分けの手法を検討する．

また，近年，あらゆるものがインターネットに繋がり，ネットワークを介して様々な情報を交換するようになる IoT (Internet of Things) の拡大や，スマートフォンの普及も影響して，世界中の各個人がアカウントを持ちテキストや写真，動画を送受信するソーシャルネットワーキングサービス，車等のデバイスからの GPS や機器情報のセンサーの発信など，世界中のデータは今も増加し続けて大規模になっており，一般にリアルタイムデータといっても，多岐にわたる．これらの情報を利用することにより，ストリーム処理を分析する際に，各データの特性を考慮した最適化が行えるのではないかと考えられる．そこで本論文では，大規模リアルタイムデータの一つであるソーシャルメディアデータを対象として，ソーシャルメディアデータがもつ特徴を利用したデータアクセスの最適化手法を，逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構それぞれに提案する．

リアルタイム分析を実施するための, **逐次的にストリーム処理をする部分**については，出来るだけフレッシュなデータをストリーム処理内に維持するために，各メッセージの流行が収束するタイミングを拡散モデルを用いて早期に推定し，時間

ウィンドウ幅をカスタマイズしてメンテナンスする手法を提案し，Twitter の実データを用いてその効果を示す．次に，代表的なリアルタイムの問合せパターンを実際に測定し，性能を評価する．インメモリなデータアクセスは高速にクエリが実行できるが，複雑な問合せパターンは演算のオーバーヘッドがあり，単純なクエリより応答時間が長くなる．そのようなクエリに対しては，計算の中間結果を保持したビューを導入して問合せの高速化が実現できることを示す．

また，ソーシャルメディアには，ある出来事が突然大きく話題になると瞬間的に多くのユーザーが一斉にメッセージをを発信し出し，バースト状態を起こす特徴がある．このようなデータバースト時のクエリの応答時間の劣化を回避するため，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセージをフィルタリングして処理するデータ量をコントロールする手法を提案する．これにより，従来のランダムなフィルタリングと比較して，同程度のサンプリング量において，問合せ結果の精度高く維持しながら，問合せ時間をより短く処理できていることを示す．

提案システム内の**蓄積されたデータを処理する部分**については，データアクセスの処理性能の評価し，提案したデータアクセス最適化手法の効果を検証する．データアクセスには，POJO ベースの O/R マッピングアーキテクチャである JPA を利用するため，まずその基本性能を評価した．データアクセス処理の高速化には，キャッシュの利用が効果的であるが，更新クエリの割合が増えてくると，従来のキャッシュメンテナンス手法では性能が低下することを確認し，我々の提案手法である列単位，値単位で更新の影響が及ぶキャッシュのみを無効化する，より細粒度のキャッシュメンテナンス手法により，性能が改善することを示す．また，無効化すべきデータを特定するための無効化用インデックスが効果を維持できるよう，メモリ領域の容量にあわせてサイズ変更可能な無効化用ハッシュインデックス導入し，キャッシュヒット率を大幅に向上できることを示す．クエリアクセスに偏りがある場

合は，無効化用重み付ハッシュインデックスを導入することにより，さらに性能を高く維持できることを示す.

また，ソーシャルメディアの情報拡散ネットワークのようなグラフデータアクセスの高速化のため，ビットマップのインデックス導入する．このインデックスは大規模な粗行列になることが想定されるため，実際に Twitter のデータを用いて，行列を圧縮してサイズ削減の効果を評価する．ソーシャルメディアデータの特性である，情報拡散と拡散されやすい経路等を考慮して，あらかじめ行列を並び替えることにより，さらに圧縮率を高めることが出来ることを示す.

Title: Optimization Method for Data Access Processing in Real-time

Abstract:

Recently many amount of data such as location, weather, social, and censor data is being produced continuously from various devices. It is essential to analyze such data in real-time. For example, monitoring censor data of a particular machine can detect or predict any fault as soon as possible. To deliver such analysis environment, real-time processing infrastructure is needed.

The infrastructure system handling big data consists of two parts. One is focusing on "Velocity" for real-time data processing. It enables real-time analysis and data calculation against continuous data. Another one if focusing on "Volume" for massive data analysis such as OLAP (online analytical processing). Large volume of data is stored in HDD based storage, so it is important to provide various query results as soon as possible.

In this paper, I propose a data processing system enables both real-time and off-line data analysis with big data. Especially, I investigate performance of data access processing and resolve some bottleneck points for faster data access. I use social data to evaluate the system. A social media service such as Twitter characterized by frequent message posting and transient topics. Over four hundred million messages are posted around the world in a day. There is a need to respond quickly on the basis of an analysis of user behaviors and to quickly identify trending messages because much of the information shared through social media quickly loses its impact. In the current Twitter system, tweets can be searched for by using keywords related to a company and monitored manually. I develop a system for analyzing the diffusion of information through retweeted tweets. The retweeting of a tweet message by many users indicates that they find the content interesting and/or entertaining.

For real-time data processing, I propose custom time window model. Streaming data is usually divided into segments called windows. However, static time window fragments diffusion data. Therefore, we retain diffusion data in the data store for only as long as it is being retweeted frequently by using retweet propagation model. When the data becomes stale, they are removed from the data store. To control capacity of incoming tweet data against bursting, I propose a filtering method using extra attribute information of tweets. I evaluate the effectiveness of maintenance and filtering methods for data control with Twitter data. There was a trade-off between query performance and accuracy of analysis results.

For massive data analysis, I create data access layer between application and database with OpenJPA which is an implementation of the Java persistence API (JPA). It has a caching layer for databases queries to share cached objects among multiple client sessions. However the performance is limited when an application includes write transactions, because the default OpenJPA cache invalidation mechanism is course-grained and this results in a low cache hit rate. I implement two kinds of finer-grained invalidation mechanisms by using query dependency analysis and invalidation index. In experiments with TPC-W benchmark, I show the OpenJPA with the finer-grained invalidation mechanisms outperform the current OpenJPA. To access diffusion data more quickly, I also develop a matrix index containing some of the data needed for diffusion analysis in the data access layer. This would be a large and sparse matrix, so data compression techniques are applied. I evaluate the performance and overhead when compressing and reconstructing the matrix.

Contents

第1章　序論

1.1 背景と目的

1.1.1 ストリーム処理システム

近年，様々なデバイスから位置情報・気象・ソーシャル・センサーデータなど，膨大な量のデータが日々発信され続けており，地球上で生成されるデータは 2013 年から 2020 年の間で 4 兆 4,000 億ギガバイトから 44 兆ギガバイトへと 10 倍の規模に拡大すると予想されている[EMC]. これらのデータは綺麗に構造化されたデータではなく，欠損を含んでいたり，自然言語で書かれているものも多く，このような不確かなデータが全データの 80%を占めるともいわれている．このような膨大なデータを利用して関連性を見つけたり，変化を検知したりと，様々な観点から分析する要求

は年々高まっており，それらを実現するための処理基盤は大規模なデータをリアルタイムに，また高速に扱えることが重要となる．

このようなストリームデータをリアルタイムに分析するため，2000 年以降ストリームデータ処理を主にした DSMS (Data Stream Management System)が登場した．従来の DBMS (DataBase Management System) と異なり，連続的に発信されるデータを対象にリアルタイムにデータ加工や分析結果を返し続けるシステムである．Stanford 大学の STREAM[STREAM04] や M.I.T らの Aurora[Au03][Au05]，NiagaraCQ[Ni00] など，アカデミックな分野からプロトタイプが作成されはじめ，IBM InfoSphere streams[IB], Microsoft StreamInsight[Mi], SAS Event Stream Processing Engine[SAS] などの商用製品も多く出現している．DBMS とのアーキテクチャの違いを図 1.1 に示す．

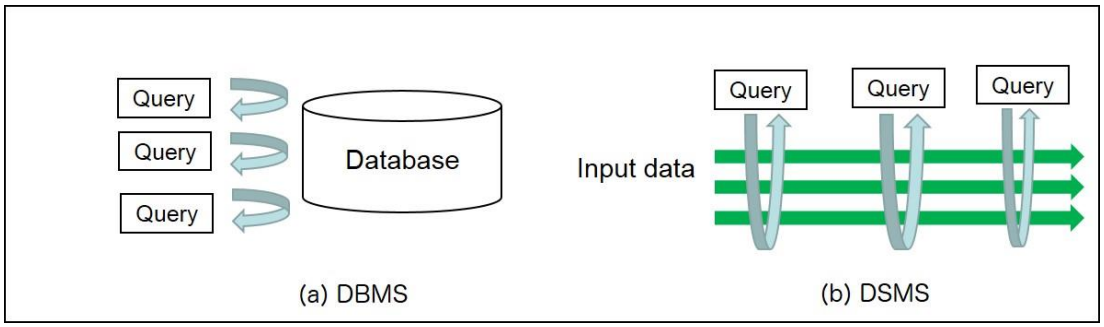


図 1.1　DBMS と DSMS

従来の DBMS は，データがあらかじめデータベースに格納されている状態から，そのデータを対象に SQL によるクエリが発行され，所望のデータが返されるアーキテクチャである．発行されてくるクエリのパターンは様々である．一方，DSMS は流れてくるストリームデータに対して，連続的にあらかじめ登録されたクエリが実行される．データへの問合せ部分に着目した時，DSMS にはいくつかの特徴があり[Go03][St05]，様々な研究が行われてきている．

1. 問合せ言語

   DSMS には，SQL のような標準的な言語は現在存在しておらず，SQL に似た宣言的な言語，CQL[CQL], SPADE[SP08], が存在してる．近年では，SQL のようにトランザクション処理を付与した研究や[Cet16]，SQL をサポートする DSMS も存在する[Esper][Jo11]．DSMS 内で， DBMS に問合せを行い, ストリームデータとリアルタイムに結合するための研究も存在する[Ro13].

   また, DSMS が開発されはじめた当初はストリームデータをフィルタリングしたり, 平均や最大最小値を出力するような比較的簡単な演算が主流であったが, 最近ではオンラインで機械学習を実施するような複雑なストリーム演算処理システムも研究されている [SPARK] [Ju].

2. 問合せの範囲

   DSMS のインプットとなるストリームデータは連続的に到着し続けるため, データを受信し始めた点から現在までの全データを保持して問合せ対象とすることは現実的ではない. そこで, ウィンドウと呼ばれる単位にデータを区切り, その範囲内に到着したデータに対してクエリの演算を施す. ウィンドウの手法にはいくつかの方法がある [SPL]. あらかじめ指定した条件が満たされた時点でストリームのデータをウィンドウとして区切り，演算を実施し，また新たにデータをウィンドウ内に蓄積していく Tumbling window や，直近 5 分間のデータを演算対象にするような，ウィンドウが時間やデータ数に応じて逐次スライドしていく Sliding window などがある.

   ここで，”あらかじめ指定した条件”としては，特定の時間(30 秒，10 分など)間隔や，到着するデータの個数が主に使用される.

3. 問合せ結果の正確性

前述にように，ストリームデータはウィンドウ単位で演算されるため，必ずしも所望のデータ範囲での正確な値を得られるとは限らない．例えば，1 日の株価の推移を知るために，秒単位の推移データをすべてデータベースに保存して計算することはデータサイズ的に現実てきではないだめ，ストリーム処理で数分単位での平均を計算して結果をデータベースに蓄積し続け，最終的に後で 1 日の平均としてそのデータをもとに近似した結果を計算することになる．

また，ストリームデータはリアルタイムにデータが発信されてくるため，その流量が時間や何かのタイミングで変化する場合は，どれ程のデータ量を 1 度に処理できるかをあらかじめ知ることは難しい．フィルタリングの手法としてはランダムにサンプリングしたり，複数のインプットストリームがある場合は，各ストリームでの削減率をもとに，全体のデータ量を調整している[Go03]．また，Wei ら[Wei06]は，ストリーム処理上の問合せ演算のコストを事前に計算し，ストリームデータがバーストする等で QoS を満たせなくなった場合はデータをサンプリングしてデータ量を削減する．削減率は，クエリコストの情報をもとに決定する．

DSMS で演算処理された後のデータは，一般的に破棄されていた．センサーデータはリアルタイムに到着し続けるため，全データをストレージに格納することは現実的ではなく，そもそもリアルタイムに異常検知したり，定期的に平均値を返すことなどを目的にしていたため，オリジナルデータを保持する必要もなかったのである．ところが，近年のストレージの格納量の大規模化，SSD 等によるデータ入

出力処理の高速化にも伴い，DSMS のオリジナルデータもストレージに格納され，それをオフラインで分析対象にすることも増えてきた[Azure].

1.1.2 ストリームデータの多様化

前述したストリーム処理に関する研究は，連続的に発信され続けるストリームデータ一般の話である．ところがここ数年では，これらストリームデータの種類がいっそう多様化している状態にある．図 1.2 は，データ数の推移と，それらデータの種類を表している．

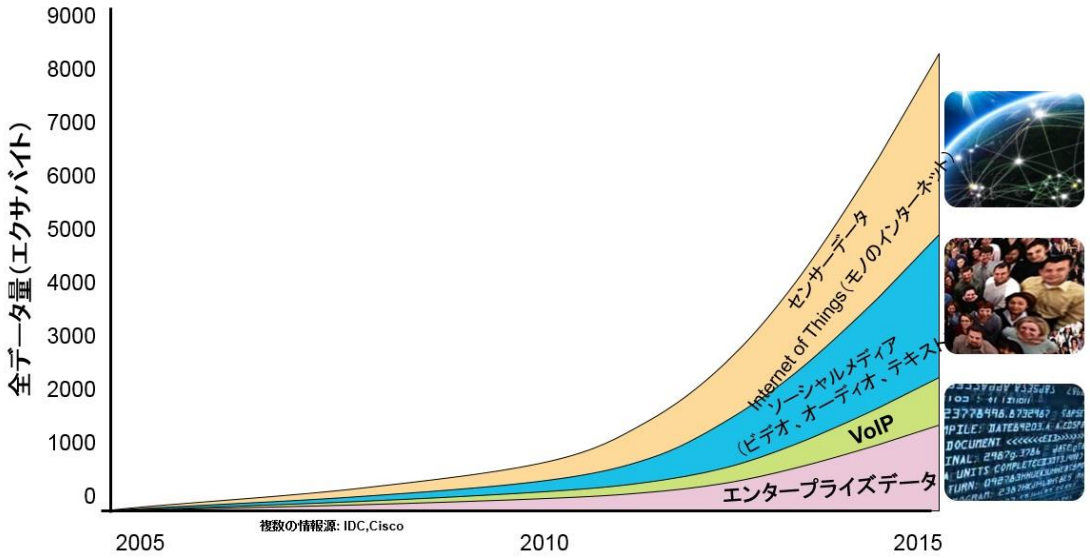


図 1.2　世の中のデータ生成量の変化

あらゆるものがインターネットに繋がり，ネットワークを介して様々な情報を交換するようになる IoT (Internet of Things) の拡大や，スマートフォンの普及も影

響して，世界中の各個人がアカウントを持ちテキストや写真，動画を送受信するソーシャルネットワーキングサービス，車等のデバイスからの GPS や機器情報のセンサーの発信など，世界中のデータは今も増加し続けて大規模になっており，一般にリアルタイムデータといっても，多岐にわたることが分かる．これらの情報を利用することにより，ストリーム処理を分析する際に，各データの特性を考慮した最適化が行えるのではないかと考えられる．

各センサーデータが持つ情報を考慮すると，それぞれに特徴がある．例えばソーシャルネットワーキングサービスに投稿されるデータは，それを発信したユーザーの名前，居住地，投稿したメッセージのトピックなどを併せ持つ．GPS データであれば，緯度経度の情報から地図へマッピングした住所の情報，道路情報，データを発信しているデバイスの持ち主の情報等を併せ持つ可能性がある．

そこで本研究では，大規模リアルタイムデータの一つであるソーシャルメディアデータを対象に，大規模リアルタイムデータアクセス処理基盤を構築する．ソーシャルメディアデータの代表である Twitter[Tw]は，1 分間に約 35 万メッセージが発信されている．Facebook[Fb]は，1 分間に約 410 万回以上のいいねボタンが送信されている[15]. 世界中で実際に起きた出来事をリアルタイムにユーザーが発信することにより，ユーザーは情報を受信するだけでなく，発信する側としても活動するようになった．それらは震災時などに，即時性のある情報として参考にされることも多く, ソーシャルセンサー的な役割を担うようになっている[Sakai10][Maru15]. 逆に，インパクトのある情報がソーシャルメディア上で発生して現実社会に影響を及ぼす現象も多く発生する[Ke]. それゆえ，ソーシャルメディア上で今どんな情報が広く拡散しているのか, をリアルタイムに知ることは, 企業や団体にとって炎上防止や流行把握のために重要である．

また，ソーシャルメディアのデータは，そのメッセージデータだけでなく，発信位置や，ソーシャルメディアを利用しているユーザーのプロファイル情報など

も含んで蓄積されていく．これらのデータを基に，メッセージの履歴や位置情報から各ユーザーの居住地や現在地を推定したり[Cheng10]，興味が似ているユーザーのコミュニティを発見して，ユーザー推薦に利用したりする研究も多く行われている[Guan11]．また，企業等もユーザーからの生の声を収集できる場として大きく注目しており，ソーシャルメディアから得られる情報を顧客対応や商品マーケティングにいかす動きも活発になってきている．

ソーシャルメディアデータに特化したリアルタイムストリーム処理のシステムに関してはこれまでも研究されている．Truthy[Rat11] [Mc13]は，Twitter 上でのアメリカの政党に関連する情報をリアルタイムにトラッキングして，各党が実施するイベントや政策等についての話題を分類したり，情報の拡散を可視化したりするウェブサービスである．Gupta ら[Gupta12]は，Twitter で盛り上がったイベントに関するツイートに対して，それらに言及したツイートの信頼度を自動で計算するアルゴリズムを提案し，信頼度順のランキングを表示するシステムを構築した．TweeQL[Mar12]は，Twitter のストリーミングデータに対して問合せを行うための，SQL-like な問合せ言語である．これにより，キーワードや位置情報，ユーザーID などを指定してストリーミングデータにフィルタリングをかけることができる．ウィンドウ単位での集約処理によるイベント検知等もサポートされている．TwitInfo[Adam11]は，ユーザーが指定したイベントに関する様々な情報をダッシュボードで表示するシステムである．必要な Tweet の情報は前述の TweeQL で収集する．イベントの盛り上がる瞬間を検知したり，ツイートの感情表現を分析して，ユーザーの感想を表示したりする．

これらの研究はソーシャルメディアデータを対象にリアルタイム分析システムを提案しているが，分析アルゴリズムをメインにしており，問合せ等のデータ処理性能については言及されていない．また，リアルタイム分析のみを提供し，蓄積されたデータ分析処理は対象としていない．前述のように，近年ではストリームデータはリアルタイム処理に使用されるだけでなく，大容量のストレージに格納され，

それがオフラインの分析にも利用されてきている[Azure]．しかしながら，ストリームデータのリアルタイムの分析とストレージに格納したオフラインでの分析を同時に考えたシステムは検討されていない．

そこで本研究では，リアルタイムなソーシャルメディアデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えたシステムを提案する．本研究ではストリームデータの収束のタイミングを議論し，リアルタイム分析とオフライン分析との使い分けの手法を検討する．また，それぞれの機構のデータアクセスの処理性能を評価し，ソーシャルメディアの特徴を考慮したデータアクセスの最適化を提案，評価する．

1.2 本論文の貢献

本論文の貢献は以下の通りである．

- 大規模なリアルタイムデータを逐次的に処理する機構と，蓄積されたデータを処理する機構を兼ね備えたシステムのフレームワークの生成とデータアクセス処理の性能評価

[逐次的なデータ処理]

- ソーシャルデータの拡散スピードを考慮してカスタマイズしたウィンドウ幅の導入と評価

1.1 の DSMS の特徴 2 で紹介したように，DSMS のデータへの問合せは，ウィンドウ単位にストリームデータを区切り，その範囲内に到着したデータに対してクエリの演算を施す．ソーシャルメディアには，あるメッセージが送信されると，それを読んだ他のユーザー達が再共有して，メッセージが拡散していく

特長がある．拡散の速度は，瞬間的に多数のユーザーに拡散したり，ゆっくりと少しずつ再共有が拡散していったりと様々である．この拡散の経路を可視化したり，拡散数によるメッセージのランキングの取得は代表的な分析シナリオである．

従来のウィンドウ機能でストリーム処理すると，”あらかじめ指定した条件”である一定の時間やデータ数でウィンドウ幅が指定され，演算処理が行われる．しかし，ソーシャルメディアデータの拡散などは，あるメッセージが送信されてから，それを再共有，もしくは返信したデータにつながりがある．そうすると，一定のウィンドウ幅ではこの繋がりが切られてしまい，例えば一つのメッセージに対する拡散の全体的な情報は分断されてしまう可能性がある．また，どのウィンドウ幅が最適であるかは自明ではなく、対象データによって適切な幅があると考えられ．これについては既存研究では議論が行われていないため、本研究でこれを明らかにする．

本研究では，ソーシャルメディアの各メッセージの流行が収束するタイミングを拡散モデルを用いて早期に推定し，メッセージそれぞれに独自の時間ウィンドウ幅を設けることで従来の固定されたウィンドウ幅よりも，拡散の全体がストリーム処理の演算対象となることを示す[Enoki01]．これは，従来の Tumbling window の手法をベースにして，ウィンドウ幅をデータごとに最適なサイズを決定して変更している．

これにより，リアルタイムに拡散が続いているデータはリアルタイム処理のウィンドウに残り続け，拡散が収束した頃にストレージに移動され，オフライン分析で利用するといった，データの使い分けが実現できる．

- ソーシャルデータをリアルタイム分析する時の代表的な問合せパターンの性能評価と，ランキング計算の高速化

  ソーシャルメディア分析における代表的な問合せの一つに，ランキング計算がある．例えば，現在広く拡散しているメッセージや，影響力の高いユーザーの発見に利用されるものである．しかしながらランキング計算はデータの集約や並び替えの計算を必要とするため問合せ応答時間が長い．そこで本研究では計算の前処理の結果を独自のビューとして保持し，ランキング計算の高速化を実現する[Enoki01].

- ソーシャルデータのメッセージの重要度を考慮したキャパシティコントロールの導入と評価

  1.1 の DSMS の特徴 3 で紹介したように, DSMS はデータの到着する量により，サンプリングされたり，近似計算結果を返すことによりリアルタイム性を確保している．ソーシャルメディアには，ある出来事が突然大きく話題になると瞬間的に多くのユーザーが一斉にメッセージをを発信し出し，バースト状態を起こすことがある. 例えば, 大規模震災時やオリンピック等のスポーツイベント，各国の選挙投票などがあげられる．そのような場合，システムは何千何十万のメッセージを同時に処理することになり，普段稼動しているサーバーのキャパシティの限界に達して処理が遅延したりデータを欠損してしまうような危険性が生じる.

  これまでは, データの流量をかえるためにランダムにサンプリングされることが主であったが，本研究では，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセージをフィルタリングして処理す

るデータ量をコントロールする手法を提案する[Enoki02]．これにより，従来のランダムなフィルタリングと比較して，問合せ結果の精度が向上することを示す．

[蓄積されたデータ処理]

- 蓄積された大規模データを高速に問合せ処理するためのキャッシュの導入と，データ更新の影響を出来るだけ抑えるためのメンテナンス手法の提案と評価

ストリームデータが逐次的に処理された後，オフラインでのデータ分析に利用するため，ハードディスクのストレージを用いたデータベースへ蓄積される．これらのデータを基に，ユーザーのプロファイル分析等を実施する．このような OLAP (online analytical processing) 処理は，データベースに発行されるクエリのパターンも多様化，複雑化する．データもハードディスクから取り出すことになるため，逐次的なデータ処理での問合せ時よりもデータアクセス処理時間が長くなる．

代表的な高速化手法は，メモリ上にデータの一部をキャッシュしたり，データベースのテーブルの結合結果や集約をビューとして保持することである．例えば DBCache[Luo02]は，頻繁にアクセスされるデータのみをキャッシュしている．DBProxy [Ami03]は materialized view の形でデータをキャッシュしている．MTCache [Lar04]は分散環境でレプリケーションされた設定でのクエリ実行時のデータをキャッシュしておくために，中間層にデータキャッシュを置くソリューションを提案した．これらの高速化手法は，データベースがほとんど更新されないことを前提としている．なぜならば，データベースとの整合性を保つため，データベースのデータが更新された場合は，キャッシュしたデータも更新しなくてはならない．Ferdinand[Cha08]はデータベースのクエリをキャッシ

ュして，オフラインでそれらを分析し，更新の影響が及ぶ範囲をいくつかのグループ分けにして更新時のメンテナンスを行う．Gupta [Gupta93]らとKenneth[Ken96],らは materialized view をインクリメンタルに更新する手法を提案している．しかしこのキャッシュメンテナンスが頻発すると，データベースとキャッシュのデータ双方の更新処理は，逆にオーバーヘッドになってしまう．

本研究で提案しているシステムは，リアルタイムに受信したソーシャルデータを，逐次的に処理した後にデータベースに蓄積される．この蓄積する処理を数分単位でのバッチ処理で実施したとしても，それなりの更新頻度となることが想定される．しかしデータアクセスの高速化に寄与しやすいデータを出来るだけキャッシュに残して保持することが理想的である．

そこで本論文では，データベースとの整合性を保ちながらも，可能な限り多くのキャッシュデータをインメモリ上に残すため，クエリパターンの依存性を分析して，データ更新時のキャッシュの無効化範囲の影響を小さくする手法を提案する[Enoki03]．さらに，データを無効化する時に使用するインデックスを導入することで，データベースの更新に対応して無効化すべきキャッシュデータを特定するための手法を提案する[Enoki04, Enoki05]．限られたメモリ領域内で無効化用インデックスが効果を維持できるよう，メモリ領域の容量にあわせてサイズ変更可能であり，さらにアプリケーションのデータアクセスの特性にあわせてインデックスの構成を最適化を行う．

- グラフデータ用のキャッシュの導入と，サイズ圧縮手法の提案と評価

  ソーシャルメディアのメッセージの再共有による情報拡散は，そのユーザー間の情報の流れをネットワークとして表現することで，可視化やグラフ分析に用いられる．このようなグラフ構造のデータを高速に取得するには，ビットマップ形式のインデックスを使用することが効果的である．しかし，このインデックスは大規模な粗行列になることが想定されるため，行列圧縮の手法を適用してより小さいサイズでインデックスを生成する[Enoki06].

  より圧縮率を高めるため，同時に参照されそうなソーシャルメディア上のユーザーをできるだけグループ化して圧縮が効きやすくするための手法を提案し，効果を評価する．また，データ更新時の行列の復元のオーバーヘッドも評価する．

1.3 本論文の構成

次章以降の構成は,以下の通りである.

第 2 章　リアルタイムデータアクセス処理システム概要

本論文で提案するシステムについて述べる

第 3 章　リアルタイム処理の最適化

逐次的なデータをリアルタイム処理するためのデータアクセスの最適化手法を提案する

第 4 章　リアルタイム処理最適化の検証

3 章で提案した手法の効果を実データを用いて検証する

第 5 章　データベースアクセスの最適化

蓄積されたデータの問合せ処理の最適化手法を提案する

第 6 章　データベースアクセスの最適化の検証

5 章で提案した手法の効果をベンチマークを用いて検証する

第 7 章　関連研究

第 8 章　結論

第2章　リアルタイムデータアクセス処理システム概要

本研究で提案する，リアルタイムなソーシャルメディアデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えた，リアルタイムデータアクセス処理システムの概要を紹介する．前述のように，本研究では大規模リアルタイムデータの一つであるソーシャルメディアデータを対象とする．現実社会で起きた災害やイベントについてユーザーが即座に情報を発信したり，逆に，インパクトのある情報がソーシャルメディア上で発生して現実社会に影響を及ぼす現象も多く発生する[Ke]．それゆえ，ソーシャルメディア上で今どんな情報が広く拡散しているのか，をリアルタイムに知ることは，企業や団体にとって炎上防止や流行把握のために重要である．これらは現状では，人手で人海戦術によるモニタリングを

実施するか，あらかじめ登録したキーワードのバーストを発見して異常検知する商用サービスを利用することが近年の企業のソーシャルメディア活用の傾向である[St]．ソーシャルメディアの情報をリアルタイムに分析してバーストやイベントを検知する研究は多く存在する．Twitter のメッセージ内容を解析して特徴的なキーワードを抽出し，その頻度のバースト性により，今何がトレンドとなっているかをモニタリングする[Mat10] [Asur11]．イベント情報を検知するためには位置情報やメッセージ内容を分析して，そのキーワードやツイート発信場所のバースト性により，今どんなイベントが発生しているかを発見する[Lee12]．これらのサービスや研究は，各ツイッターのメッセージに出現する「キーワード」の増減の情報を基にした分析である．必ずしもツイート間には繋がりはなく，同じキーワードを話題にしているという状態を分析対象にしている．

対して，本研究が主な対象とするのは，メッセージの再共有(リツイート)で広がっていく情報拡散である．あるツイートが多数のユーザーにリツイートされて広く拡散したメッセージは，多くのユーザーが興味をもち，インパクトを与えた情報であるといえる．また，あるトピックに関する複数のツイートの拡散データを対象にして，それらのツイートを頻繁にリツイートしているユーザー達や，逆によくリツイートされているユーザー等，「キーパーソン」を見つけることにより，「どのようなユーザーが興味をもっているか」「誰が話題の中心になっているか」「どのようなユーザー間の流れを介して情報が拡散しているのか」という事を発見することが期待される．発見したユーザーはユーザープロファイリングなどの分析を行うことにより[Nasu13]，人となりを深く分析可能になる．本論文はこのような情報拡散データの分析を代表例としてシステム構築を行う．

本研究では, 大規模なソーシャルメディアデータの代表である Twitter[Tw]のデータを例に，システムを構築しデータアクセスの性能評価を実施する．

2.1 情報拡散データ

2.1.1 Twitter

Twitter [Tw] は，マイクロブログサービスを代表するサービスであり，2006 年にサービス開始されてから, 2015 年 2 月時点で約 3 億 2000 万人のアクティブユーザーがいる[Tn]．ユーザーが発するメッセージはツイートと呼ばれ，1 メッセージは 140 文字以内の文章で構成される．Twitter はフォロー(Follow)と呼ばれる機能を有し，あるユーザーが他のユーザーをフォロー登録することで，そのユーザーが発信するメッセージを自分の Twitter のホームページに表示させることができる(=タイムライン)．自分をフォローしているユーザーを，フォロワーと呼ぶ．また，ユーザー間でのメッセージのやり取りとして Twitter には主に二つの特徴がある．これらの特徴は，Twitter 特有のものではなく，ソーシャルメディアシステム一般にみられる特徴である．

返信 (Reply)

あるユーザーが発信したツイートに対して，返信のツイートを発信すること．ツイートの先頭に@ユーザー名を挿入することで，そのメッセージはユーザー名宛の返信扱いとなる．

リツイート (ReTweet, RT)

あるユーザーが発信した Tweet を再発信すること．公式リツイートと非公式リツイートが存在し，公式リツイートは再発信元のユーザー名でツイートが発信され，非公式リツイートは再発信元ツイート(再発信元のユーザー名を AA とする)の先頭に

"(再発信するユーザーのコメント) リツイート @AA " を挿入して再発信者のツイートとして発信する.

2.1.2 情報拡散ネットワーク

ある一つのツイートに対して，それをリツイートしているデータを関連付けて蓄積していくと，1 つの情報拡散データとなる．1 リツイートをエッジとし、リツイートしたユーザーとされたユーザーをノードとするグラフ構造を拡散ネットワークと呼ぶ．拡散ネットワークを可視化すると，拡散の規模や拡散経路が視覚的に捉えられて直感的に理解しやすくなる.

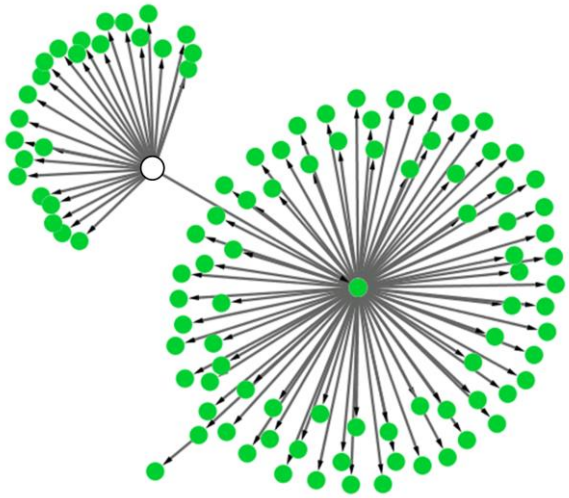


図 2.1 情報拡散ネットワーク

図 2.1 はオリジナルのツイートを発信したユーザー(中心が白色のノード)と，そのツイートをリツイートしたユーザー(色の付いたノード)をネットワークのノードとし，エッジは情報の流れを表している．例えば，ユーザー@b がユーザー@a の

ツイートをリツイートした時，@a のノードから@b のノードへ向かうエッジがはられる．

図 2.1 をみると，オリジナルツイート発信ユーザー以外にも，多くリツイートされているユーザーがいることが分かる．そのようなユーザーは，ソーシャルメディア上の情報を収集･整理し提供するキュレーターユーザー[Cu]である可能性が高い．キュレーターは面白い記事や情報価値の高い記事を共有することが多いため，フォロワーも多く，情報拡散力が高いといえる．

しかしながら Twitter では，リツイートで共有されたツイートには，リツイートしたユーザーと，オリジナルのツイートを発信したユーザーの情報しか含んでいないため [Enoki07]，仮に誰かがリツイートしたツイートを読んで，同じツイートをさらにリツイートしていたとしても，そのような情報伝播の経路をツイートの情報だけを用いて明らかにすることは不可能であり，図 2.1 のような拡散ネットワークを生成することができない．

Twitter を例にすると，図 2.2 に示すように，Twitter で@IBMResearch をフォローしているユーザーのタイムラインには，ユーザーIBM Research (@IBMResearch)が@IBMcloud のツイートをリツイートしたという情報が付与されているリツイートが表示される．これをタイムラインで読んだユーザー@B がリツイートした場合，次はユーザー@B をフォローしているユーザーのタイムラインに，ユーザー@B が@IBMcloud のツイートをリツイートしたという情報が付与された状態でリツイートが表示される(@IBMResearch と@B を両方フォローしている場合は表示されない時もある)．すなわち，実際にはユーザー@B はユーザー@IBMResearch がリツイートしたことによりこのツイートを読み，リツイートすることになったのだが， @B がリツイートしたツイートにはリツイートしたユーザー@B とそのオリジナルの発信ユーザーである@IBMcloud の情報しか保持されていないため，ユーザー@B は，”

@IBMResearch がリツイートした情報のついた@IBMcloud のツイートをリツイートした” という伝播経路の情報は失われてしまう．

図 2.2　タイムラインに表示された IBM Research によるリツイート

2.1.3 情報拡散経路の推定

リツイートしたユーザーがオリジナルのツイートの発信ユーザーを直接フォローしていた場合は，そのツイートが自分のタイムラインに表示され，オリジナルユーザーを直接リツイートした可能性が高いといえる．しかしながら，オリジナルのツイート発信者をフォローしていない場合，リツイートしたユーザーは他のユーザーがリツイートしたツイートを読んだか，タイムライン以外の手段でツイートを読んだ可能性が高いといえる．

そこで我々は，どのユーザーのリツイートを読んで（もしくはオリジナルツイートを直接読んで），リツイートしたのかを，Twitter のフォローネットワークの情報と，リツイートした時刻を用いて推定する[Enoki07]. フォローネットワークとは，Twitter 上で，各ユーザーが誰をフォローしているのか，という関係を保持したネットワークのことである．

Twitter の場合，自分のタイムラインに，あるリツイートが表示され，その後，他のフォローユーザーが同じツイートをリツイートした場合は，タイムラインには

重複して表示されない(しかし初出のリツイートツイートから約 24 時間経過した場合は，そうなるとは限らない)．したがって，推定対象となるリツイートしたユーザーが，オリジナルツイート発信ユーザーをフォローしている場合は，オリジナルのユーザーから情報が伝播したとし，フォローしていない場合は，自分がフォローしているユーザーかつ，そのツイートをリツイートしたユーザーの中で，リツイートツイートの発信時刻が最も早いユーザーほど，そのユーザーからリツイート情報が伝播してリツイートした可能性が高いとする．

2.2 リアルタイムデータアクセス処理システム概要

本研究のシステム構成を図 2.3 に示す．Twitter から発信されるツイートをリアルタイムに取得し，各データはその拡散がリアルタイムに続いている限りはウィンドウの中に保持し続け，リアルタイム処理の対象とする．この一時的なデータのストア場所に，インメモリのデータストアを使用する．Twitter から日々発信されるツイートは膨大な量であるため，収集対象とするツイートを例えば，特定のユーザーが発信するツイートの拡散や特定のキーワードが含まれるツイートのみを格納するように指定しても良い．

システムのインメモリデータストアには，リレーショナルデータベース，グラフデータベースやキー/バリューストアが候補としてあげられる[HB][Neo]．分析ユーザーが人気のリツイートやユーザーを発見するためには集約やソートの処理が必要であり，キー/バリューストアよりも，SQL を用いて複雑なクエリが実行できるリレーショナルデータベースのほうが分析の幅が広がると考えるため，本研究ではインメモリデータベースを採用する．データストアから退避されることになった拡散データは，オフラインでのデータ分析に利用するため，ハードディスクのストレージを用いたデータベースへ蓄積されるか，不要なデータの場合はそのまま破棄する．

アプリケーションサーバーには，拡散ネットワークを分析するための複数のモジュールが入り，分析ユーザーはインタラクティブに分析を実施する．例えば，分

析ユーザーは特定のツイートの集合に対して，多くリツイート されている人気のユーザーやツイートを発見する．例えば"Ranking"は，ユーザーが指定したツイートの集合に対して，インフルエンサーを見つけたり[Gu11][Enoki07]，人気のあるツイートを発見したりする．"Visualization"モジュールは，ユーザーが指定したツイートの拡散経路を可視化する．他にも，あるツイートをリツイートしたユーザーの集合を取得し，それらのユーザーのプロファイル情報を分析することにより，趣味や居住地などの特徴を分析することなども可能である．

各分析モジュールは，所望のデータを取得するために，アプリケーションサーバー内にある”Data access layer”を介してデータベースへアクセスする．”Data access layer”は，分析モジュールからのリクエストを基に，拡散ネットワークデータをデータベースから取得する．どのような属性情報が必要かは分析モジュールに依存する．データベースへのアクセスには，O/R マッピングの Java 用フレームワークである JPA(Java Persistence API)を用いる．これにより，各分析モジュールは DB の種類に依存することなく，Java のオブジェクト操作でデータをやり取りすることができる．

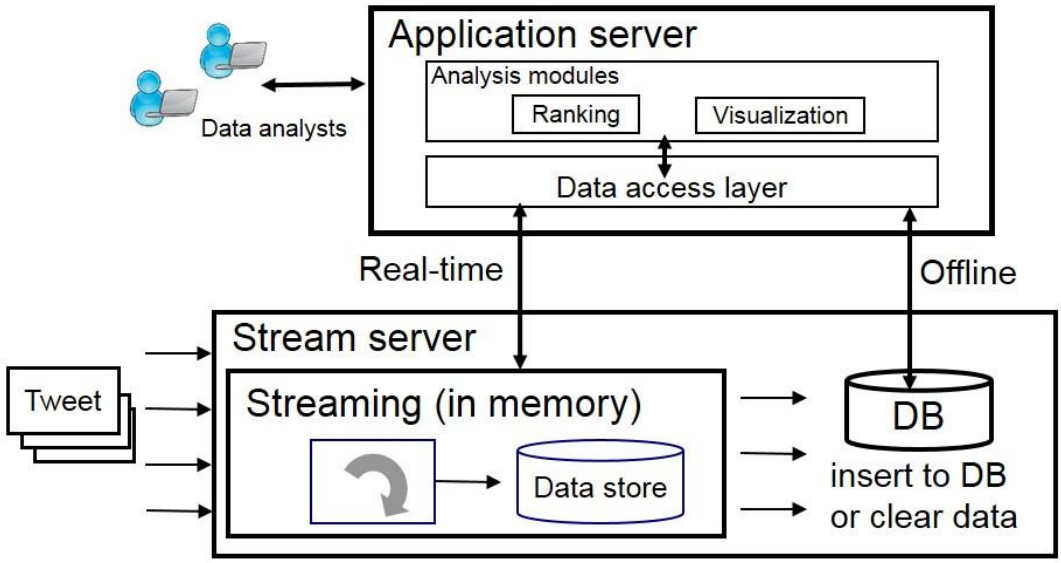


図 2.3 情報拡散ネットワーク

2.3 まとめ

本章では，本研究で提案する，リアルタイムなソーシャルメディアデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えた，リアルタイムデータアクセス処理システムの概要を紹介した．本研究が主な対象とするデータは，ソーシャルメディアデータの中でも，メッセージの再共有(リツイート)で広がっていく情報拡散のデータである．

次章以降では，逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構それぞれに対し，ソーシャルメディアデータの特性を考慮した，データアクセスの最適化と，提案手法の評価を実施していく．

第3章　リアルタイム処理の最適化

本章では，本システムにおける，リアルタイム処理部分においてのデータアクセスの最適化について述べる．ソーシャルメディアのデータをリアルタイムに取得し，各データに対して拡散がリアルタイムに続いているデータをウィンドウの中に保持し続け，リアルタイム処理の対象とする．そのために，ストリームデータの収束のタイミングを議論し，リアルタイム分析とオフライン分析との使い分けの手法を検討する．具体的には，アクティブに拡散し続けているデータをリアルタイム分析の対象として維持するためのウィンドウ幅の計算を，ソーシャルメディアの各メッセ

ージの流行が収束するタイミングを推定し，メッセージそれぞれに独自の時間ウィンドウ幅を設ける．一定値で固定された従来のストリーム処理よりも，メッセージの拡散全体がストリーム処理の演算対象となることが期待される．これは，従来の Tumbling window の手法をベースにして，ウィンドウ幅をデータごとにカスタマイズしてサイズを変更している．これにより，リアルタイムに拡散が続いているデータはリアルタイム処理のウィンドウに残り続け，拡散が収束した頃にストレージに移動され，オフライン分析で利用するといった，データの使い分けを実現する．

次に，このデータに対する，代表的なリアルタイムの問合せパターンを紹介する．リアルタイム分析においては，クエリに対して早い応答性能が重要であるため，複雑な問合せパターンには計算の中間結果を保持したビューを導入して問合せの高速化を実現する．複雑な問合せの一つである，影響力の高いユーザーのランキング演算においては，文書集合の頻出単語を高速に計算するアルゴリズムを応用して，ビューを作成する．

ソーシャルメディアの特徴の一つとして，ある出来事が突然大きく話題になると瞬間的に多くのユーザーが一斉にメッセージをを発信し出し，バースト状態を起こすことがある．そのような場合，システムは大量のメッセージを同時に処理することになり，普段稼動しているサーバーのキャパシティの限界に達してクエリの処理が遅延したりデータを欠損してしまうような危険性が生じる．これまでは，データの流量をかえるためにランダムにサンプリングされることが主であったが，その場合，データがランダムに無くなるため，問合せ結果の精度への影響がトレードオフとなる．本研究では，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセージをフィルタリングして処理するデータ量をコントロールする手法を提案する．これにより，従来のランダムなフィルタリングと比較して，問合せ結果の精度が高く維持できることを目標とする．

3.1 リアルタイム処理のデータストアメンテナンスのための，拡散の収束を考慮したカスタマイズした時間ウィンドウの導入

3.1.1 リアルタイムストリーム処理

ソーシャルメディアのメッセージはリアルタイムに逐次ユーザーから発信されるため，ストリームデータとして捉えることができる．ストリームデータをリアルタイムに分析する研究がこれまでにも行われてきている[Jain08] [Pol07] [Babu01] [Ara06]. ストリーム処理の特徴は，ストリームデータの数や時間によってウィンドウ幅を設けてデータを区間に区切り，その範囲に含まれるデータを分析対象にする．ウィンドウの手法にはいくつかの方法がある [SPL]．あらかじめ指定した条件が満たされた時点でストリームのデータをウィンドウとして区切り，演算を実施し，また新たにデータをウィンドウ内に蓄積していく Tumbling window や，直近 5 分間のデータを演算対象にするような，ウィンドウが時間やデータ数に応じて逐次スライドしていく Sliding window などがある．ここで，”あらかじめ指定した条件”としては，特定の時間(30 秒，10 分など)間隔や，到着するデータの個数が主に使用される．

Tumbling window を採用して，ソーシャルメディアの拡散データをストリーム処理で扱う場合，逐次到着するリツイートを時間ウィンドウで切り，再共有元であるオリジナルメッセージの ID 単位でデータを区分化して処理すると，図 3.1 のようになる．あるツイート(Tweet1)が投稿された後，時間経過と共に他のユーザーがリツイートすると，Tweet1 のリツイートのスレッドが発生して，リツイートの情報が蓄積される. Tweet2 も同様に, 他ユーザーからリツイートされて情報が追加されていく．

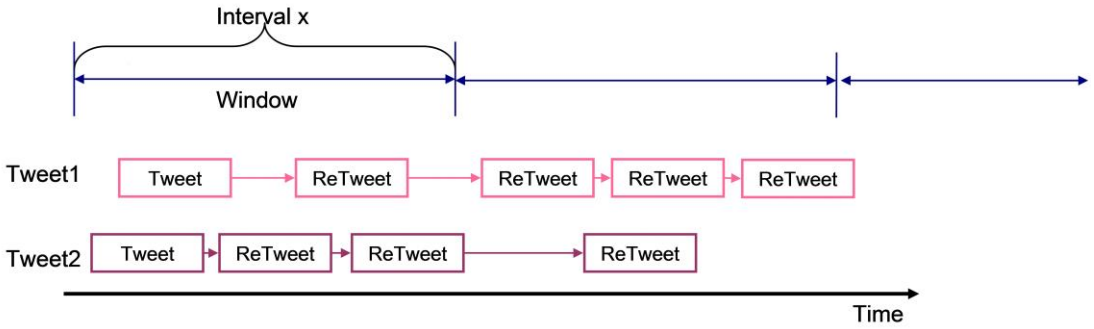


図 3.1　リツイートのストリーム処理

ここで，ウィンドウ幅を 1 時間として，各ツイートのリツイートをカウントした場合，過去 1 時間の範囲でのリツイート総数が分かる．しかしながら，それ以前のデータはストリームシステムには残されないため，各ツイートの拡散ネットワークは時間で分断された状態になる．したがって，あるツイートが発信されたから，リツイートが広がり，それが収束するまでの，拡散データ全体を捉えた分析をすることが困難である．ツイートの拡散は，短期間で爆発的に拡散するものや，時間経過と共に少しずつ拡散するもの等様々である[Matsu13]．短期間で爆発的に拡散した場合，ウィンドウで切った場合でも頻繁にリツイートされた瞬間を捉えて，多数リツイートされたツイートとして発見できるかもしれないが，少しずつ時間をかけてリツイートが広がった場合は，時間単位のリツイート数はそこまで多くならず，発見されない可能性がある．

そこで本研究では，拡散データをインメモリのデータストアに一時的に格納して，リアルタイムに拡散分析を行うシステムを提案する．ストリーム処理は一般的に事前にクエリを登録して，逐次到着するデータに対してクエリを発行するが，インメモリに蓄積されているデータストアを対象にすることで，分析ユーザーがインタラクティブにクエリを発行することも可能になる．

一方で，データストアに格納した場合，サーバーのメモリサイズには限りがあるため，延々と蓄積し続けることは現実的ではない．また，すっかりリツイートされなくなった鮮度の低い拡散データなどが残り続けて分析対象になってしまうことが懸念される．そこで，各ツイートに最適な時間ウィンドウ幅を設定するアルゴリズムを提案する．これにより，今もリツイートされ続けているアクティブな拡散データはデータストアに出来るだけ格納し続け，拡散が収束したデータはデータストアから退避するようなメンテナンス処理を実現する．

3.1.2 拡散の収束を考慮したカスタマイズした時間ウィンドウ設定手法の提案

本節にて，ツイートごとにカスタマイズした時間ウィンドウを設定する手法を提案する．これにより，ツイートが頻繁にリツイートされている間はデータストアに拡散データが格納され続け，リツイートが収束してきたらデータストアから退避される．

図 3.2 はカスタマイズした時間ウィンドウの例を表している．時間ウィンドウ幅を，各ツイートが発信されて，リツイートが続いて，それが収束するまでに設定することで，データストアには全体の拡散データが格納されると期待される．

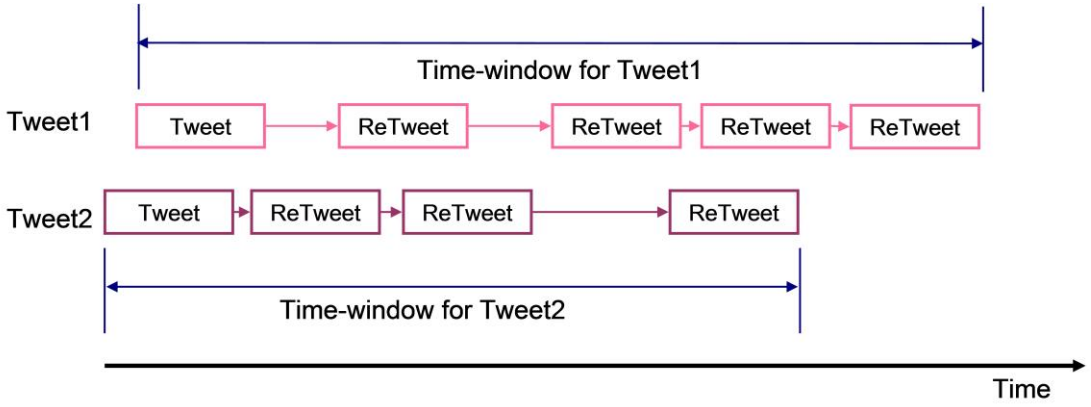


図 3.2　カスタマイズされた時間ウィンドウ

ウィンドウ幅を決める単純な手法としては，あらかじめある時間幅を一つ設定して，どのツイートも発信時刻からその時間が経過した時点て，拡散が収束したとみなすことが考えられる．リツイート元であるオリジナルツイートの ID 単位でデータを区分化した後，それぞれに対して，現在のストリーム処理の Tumbling window のモデルを拡張して，各区分にウィンドウ時間を設定すると，各ツイートの拡散データは，オリジナルツイートが発信されてから，指定されたウィンドウ幅の時間だけインメモリデータストアに格納される．指定した時間が経過したら，そのオリジナルツイートとそのリツイートデータはすべてインメモリデータストアから退避する．しかしながら，拡散が収束するまでの時間はツイートによって様々であるため，固定値ではそれぞれの拡散の収束のタイミングをうまく捉えることは難しい [Kw10].

3.1.2.1 拡散速度を考慮した拡散収束予測

拡散が収束するということは，そのツイートが徐々にリツイートされなくなっていくことを意味している．そこで，それぞれのツイートの拡散の速度を計算し，ある速度を下回ったら，拡散が収束したと判断する．各ツイートに対して，一定時間(= t)あたりのリツイートの数を用いて以下のように表される．

$$\alpha = \frac{(\#\quad of\quad RT)}{t}$$

時間が経過するに従い，時間 t の対象する時間幅をスライドさせていき，現在時刻から，過去時間 t 前までさかのぼった領域が，計算対象となる．したがって，この計算は定期的に実施される．そして，この α が閾値を下回った時点で，拡散が収束したとみなし，ウィンドウの終点とする．

3.1.2.2 情報拡散モデルを用いた拡散収束予測

前述の手法では，リツイートの頻度が下がり，拡散が収束した時点を判断できることが期待されるが，閾値αの設定はユーザーが決定しなくてはならず，その閾値によっては，時間をかけてコンスタントにリツイートされ続けて拡散がゆっくり続いているツイートがあったとしても，途中で閾値以下となった時点で収束したと判断されて，ウィンドウ幅が終わりインメモリデータストアから退避されてしまう可能性がある．そこで，本節では，拡散の広がりをモデル化し，それぞれのツイートの拡散の収束判定をより精度よく実施できるようにする．

ツイートの拡散のモデル化はこれまでも研究されてきており，時間経過の拡散の分布は主に対数正規分布に従うとされている[Matsu13] [Gal10] [Asur11]．図 3.3 はある一つのツイートが発信されてからの，リツイートの発信時間の分布を示している．まずオリジナルのツイートが発信された後，それを読んだユーザーがリツイートを始め，またそれを読んだユーザーがリツイート，といったように拡散が多数広まり，ある時間が経過した後に収束していく．このような流れはリアルタイムのソーシャルデータでは頻繁に見られる現象である．

そこで我々は，ツイートが発信された時刻を基準にして，そこから短時間内のリツイートの拡散度合いの情報を対象にして対数正規分布にフィッティングする．この分布をもとに，各ツイートの拡散がいつ頃収束するかを推定する．

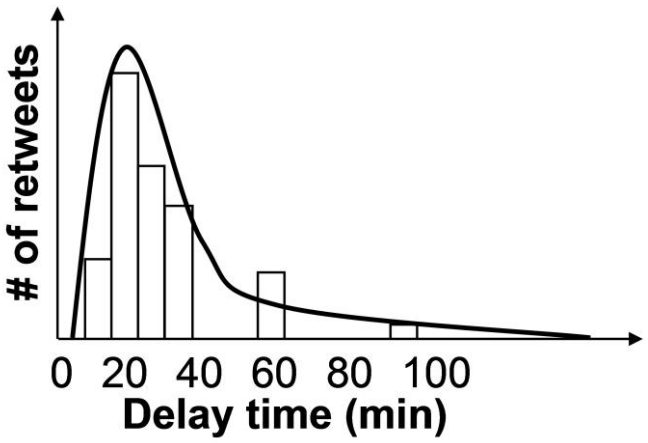


図 3.3　あるツイートのリツイートの分布

具体的な手順は以下のようになる．

対数正規分布の確率密度関数は, パラメータ μ と σ を用いて以下のように表される．

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{\frac{-(\ln x - \mu)^2}{2\sigma^2}} \quad , \quad x > 0$$

オリジナルのツイートが発信されてから 1 時間経過した時点でのそのツイートに対するリツイートの集合を用いて，オリジナルツイートが発信された時刻を 0 とし，各リツイートは，自身が発信された時刻と，オリジナルツイートの発信された時刻との差分を計算する．この差分の時間のリストを入力データとする．この入力データをもとに，データに尤もらしい確率密度関数の μ と σ を推定する．

得られた確率密度関数に対して，上側確率の 90%に対応する点を求め，そのツイートがほぼ収束する時間であるとみなす．これにより，ツイートがリツイートされなくなるまでを全体とした 90%程度をカバーする時間を推定できると期待する．すなわち，この点がそのツイートのウィンドウ幅の終点となる．終点に到達した拡散データは，インメモリデータストアから退避され，リアルタイム分析の対象ではなくなる．

この推定は，ツイートとそれがもつリツイートの集合を 1 セットとして，それぞれに実行される．したがって，各ツイートの拡散ごとに確率密度関数の μ と σ の推定が計算される．本手法では，オリジナルツイート発信後 1 時間のリツイートの情報を基に推定しているが，この時点で総リツイート数が少ない場合はうまくフィッティングできない可能性がある．このような場合に対処するため，リツイート数がある閾値を下回る場合は，ツイートは，1 時間後の時点でほとんど拡散していないと捉え，既に拡散収束していたと判断して，インメモリデータストアから退避する．

以上，カスタマイズしたウィンドウ幅を導入し，その終点を計算するためにストリームデータの収束のタイミングを推定して，リアルタイム分析とオフライン分析とを使い分けるためのストリームデータのメンテナンス手法を提案した．

3.2 代表的な問合せパターンと高速化

本節では，リアルタイム処理部分においてのインメモリデータストアのデータの構造と，代表的な問合せパターンを紹介する．

3.2.1 インメモリデータベース

インメモリデータベースには逐次到着するツイートのデータを格納していく．インメモリのデータベースの実装はいくつか存在し, オープンソースの H2, Apache Derby や，商用の TimesTen, solidDB などがある[In]．データベースには，RETWEET テーブルと，ORIGIN_TWEET テーブルを生成する (図 3.4)．RETWEET テーブルには，リツイートのツイートの ID(TweetID) ，オリジナルツイートのツイート ID(RT ID)，リツイートを発信したユーザー名(Dst)，リツイート元であるオリジナルツイートを発信したユーザー名(Src)，リツイート発信時刻(Time)，使用言語(Lang)，位置情報(Location)等を属性として持たせる．1 レコードが図 2.1 の 1 エッジに該当する．

ORIGIN_TWEET には，ツイート ID (TweetID)，ツイート発信時刻(Time)，発信ユーザー名(User)，ツイートのメッセージ(Msg)，リツイートされた回数(RTcount)がある．RTcount は，対応するリツイートを受信するたびにカウントされる．あるリツイートの，リツイート元となるオリジナルツイートの情報を取得したい場合は，RETWEET テーブルの RTID と，ORIGIN_TWEET テーブルの TweetID を Join 結合して問合せを行う．

RETWEET table

TweetID	RTID	Time	Src	Dst	Lang	Location	‥
100	1	Time data	u1	u2	Ja	GPS	‥
101	1	Time data	u2	u4	Ja	GPS	‥
‥	‥	‥	‥	‥	‥	‥	‥

ORIGIN_TWEET table

TweetID	Time	User	Msg	RTcount	‥
1	Time data	u1	message	29	‥
2	Time data	u5	message	14	‥
‥	‥	‥	‥	‥	‥

図 3.4　拡散ネットワークデータベースのテーブル

3.2.2 拡散分析のための問合せパターン

本システムにて拡散データを対象にした代表的な問合せパターンを以下に紹介する．

(1) 指定したツイートの拡散ネットワークを取得

拡散ネットワークを生成するため，所望の拡散データを問合せる．これにより，「ユーザー間をどのような経路でリツイートが拡散していったのか？」といったような情報を得ることができる．また，生成した拡散ネットワークは，クラスタリングや頻出経路発見などのネットワーク分析に応用することも可能となる．この問合せパターンに対応する SQL は以下のようになる：

[Query 1]

```
SELECT    Src, Dst
FROM      RETWEET
WHERE     RTID in (tweet ids)
```

インプットはツイートIDであり, このIDは分析モジュールにて指定されるか, もしくはデータベースから取得される．例えば，データベース内の ORIGIN_TWEET テーブルに対して，あるトピックを表す特定のキーワードが含まれるツイートのツイート ID リストを取得し，そのリストを Query 1 のインプットにすることにより，そのトピックに関連する拡散ネットワークを取得する．

(2) 人気のあるツイートを取得

リツイート数の多い順のツイートランキングを問合せる．この問合せパターンに対応する SQL は以下のようになる：

[Query2]

```
SELECT    *
FROM      ORIGIN_TWEET
WHERE     TweetID in (tweet ids)
ORDER BY          RTcount    DESC
FETCH FIRST 100 ROWS ONLY
```

WHERE 句にてツイート ID を指定しない場合は，単純に現在格納している拡散データの中で，最もリツイート数の多い上位 100 件が出力される．

(3) キーパーソンとなるユーザーを取得

リツイートされた，もしくはリツイートした数の多い順のユーザーランキングを問合せる．この問合せパターンに対応する SQL は以下のようになる：

```
[Query3]
SELECT    Src, count(Src)
FROM      RETWEET
WHERE     RTID in (tweet ids)
GROUP BY          Src
ORDER BY          count(Src)   DESC
FETCH FIRST 100 ROWS ONLY
```

Query 3 にて Src を指定した場合は，指定したツイートの集合の中で，最もリツイートされた総数の多い順のユーザー(＝拡散影響力の高いユーザー)が出力される．一方，Dst を指定した場合は，指定したツイートの集合の中で，最もリツイートしていた総数の多いユーザー(=情報をリツイートしやすいユーザー)が出力される．

以上のような問合せは，インメモリデータベースで処理されるためディスクベースのデータベースよりもデータ処理が高速になることが期待される．しかしながら，予めインデックスを張ったカラムに対するシンプルな問合せは高速に処理可能であるが，ソートや Join，副問合せ等メモリ上でのデータ演算が問合せコストの多くを占めるような複雑な SQL になるほど，処理能力が HDD のデータベースと同程度まで下がってしまう可能性がある [Id].

本システムの場合，Query 3 の問合せ処理は，ユーザー単位での集約，ソートの処理等が必要になるため，やや複雑な SQL となる．

3.2.3 ランキング計算のためのデータアクセス高速化

Query 3 のようなランキング計算を伴う問合せは，Top-k 計算の最適化アルゴリズムを応用できる．そこで本研究では，ランキング計算のための事前処理結果を独自のビューとして保持し，問合せの高速化を実現する．図 3.5 は，リツイートされたユーザーのランキングを計算するためのビューである．オリジナルツイートの ID ごとに，リツイートされたユーザーとリツイートされた回数のペアが．リツイート回数の多い順にソートされてリストで保持されている．

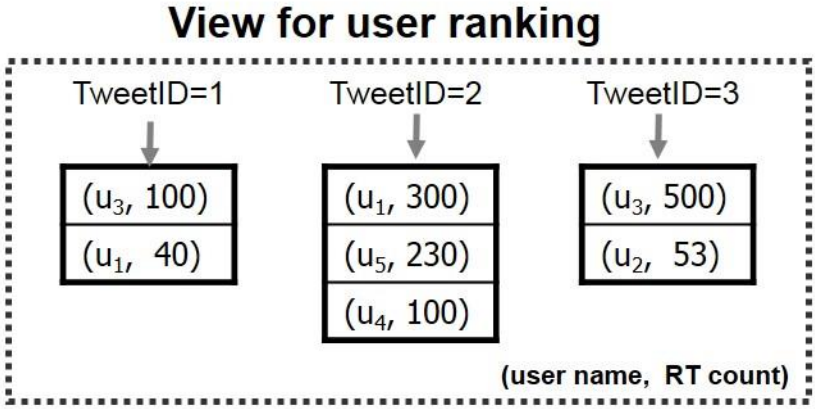


図 3.5　ユーザーランキング計算のためのビュー

このようなユーザー単位に集約したビューを用いると，top-k 計算の代表的な効率的な計算アルゴリズムである，Fagin ら[Fagin01]の Threshold Algorithm を適用することができる．

以下に，「ツイートの集合を対象にして，リツイートされた回数によるユーザーのランキング計算」を実現する場合を例に問合せ処理の手順を説明する．ランキングは，上位 k 件を出力するものとする．

[top-k 検索のためのビュー作成]

各オリジナルツイート ID（t とする）に対し，以下の検索ビュー S(t), 検索インデックス R(t) を作成する

S(t):ユーザーと被リツイート数の値の組(u, w)を w 値の降順に並べ，値の大きい順から各組に順次アクセス可能なリスト

R(t):任意のユーザーx に対して、S(t) 内の組 (x, w) を定数時間で取得するためのランダムアクセス検索インデックス．任意のマップ構造で生成可能

図 3.5 の例では，S(1) = [(u3, 100), (u1, 40)], S(2) = [(u1, 300), (u5, 230), (u4,100)], S(3)=[(u3, 500), (u2, 53)] となる．

[top-k アルゴリズム]

ツイートの集合(TweetID のリスト)を {t1, t2, …, tm}として，被リツイート数の多いユーザー上位 k 件を出力する手順を以下に示す.

```
Input   S(t), R(t), Tweetid_list = [t1, t2, …, tm],k
Output  result_candidate  //top k user list
1.    result_candidate = [];
2.    For each i=1,...,m, Do
   1.    If !S(ti).hasNext() Then continue;
   2.    Retrieve (ui, wi) from S(ti);
   3.    If result_candidate.contains(ui) Then continue;
   4.    V := sum of w in each R(tj) for ui // j=1,...,m
   5.    Vk := sum of w in the k-th candidate in
         result_candidate;
   6.    If (V > Vk) Then update result_candidate      by
         (ui, V).
   7.    V' := sum of the minimal value of w in S(ti)
         that have been retrieved so far // i=1,...,m
   8.    If (V' >= Vk) Then break;
3.    End For
```

ビューを導入することにより，問合せ処理の高速化を実現できるが，その分サーバーのメモリ領域を消費する．クエリの頻度などを参考に，どのランキング用のビューを作成するべきか考慮する必要がある．

また, top-k のランキング計算に関しては, リレーショナルデータベースの top-k 計算を高速化するために，あらかじめ materialized view を生成しておき，それを利用する研究がある[Gupta99] [Hri01]．また，データベースが更新されたときの効率的なメンテナンス手法についても提案されている[16]．本研究においても，今回対象にしたクエリは，ユーザーやツイートのランキングであったが，今後他にも複雑な観点でのランキング処理が分析に必要となる場面がでてくる可能性がある．その場合，中間処理結果を materialized view を入れる等の高速化が役に立つと考えられる．しかしながら，view 用に追加のデータ領域が必要となるため，view 導入におけるトレー

ドオフを考慮することが大切である．また，時間経過に伴いリツイートの拡散が広がっていくと，導入しているインデックスの情報も更新する必要があるため，メンテナンス処理についてこれらの関連研究を参考にしていきたい．

3.3 データバースト時のキャパシティコントロール

3.3.1 データバースト時に起こる性能劣化

ソーシャルメディア上では，あるトピックが瞬間的に大きく話題になると多くのユーザーが一斉にツイートを発信し，バースト的な状態を起こすことがある．例えば，大規模な震災が発生した瞬間やオリンピックなどのスポーツの試合で盛り上がった瞬間，各国の選挙投票日などがあげられる．そのような場合，システムは何千何十万のメッセージを同時に処理することになり，普段稼動しているサーバーのキャパシティの限界に達して処理が遅延したりデータを欠損してしまうような危険性が生じる．そこで本節では，各ツイートの重要度を計算し，重要度が低いと判断されたツイートをフィルタリングして処理するデータ量をコントロールする手法を提案する．

図 3.6 は日本の衆議院議員選挙の開票日(2014 年 12 月 14 日)から翌日にかけての各時間帯の，政党名を含んだツイート（含リツイート）の発信数を示している．これにより，開票開始時間である 14 日の 20 時に大きくツイート数が跳ね上がっていることが分かる．このように瞬間的なバーストが発生した時，システムを稼動しているサーバーのリソース不足等を引き起こしてしまう可能性がある．

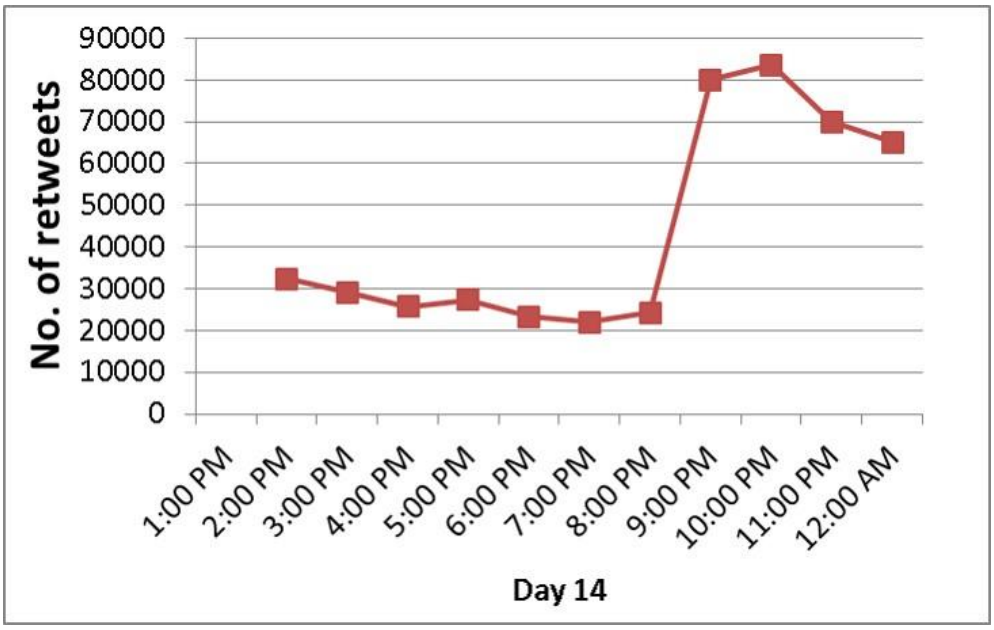


図 3.6　ツイートのバースト

図 3.7 は各時間帯においての，Query 2, 3 の平均応答時間とストリームサーバーの CPU 使用率をを示している．開票時間後に徐々に応答時間は長くなり，22 時台には約 3 秒もかかっている　（詳細な実験設定については次章に述べる）．CPU 使用率も 90%にまで到達し，リソースが限界近くまで達していることがわかる．本システムは，分析ユーザーからインタラクティブに問合せ処理が発行されることを想定している．したがって，問合せ応答時間が数秒を要するということは性能上で問題である．

解決策として，システムを分散することが考えられるが，データが分散することになり，分散問合せ処理を実行しなくてはならなず，問合せ実行時間に影響を及ぼす可能性があり，トレードオフとなる．本研究では，他の手段として，分析においてあまり重要ではないデータをフィルタリングすることにより，性能劣化を回避する手法を提案する．

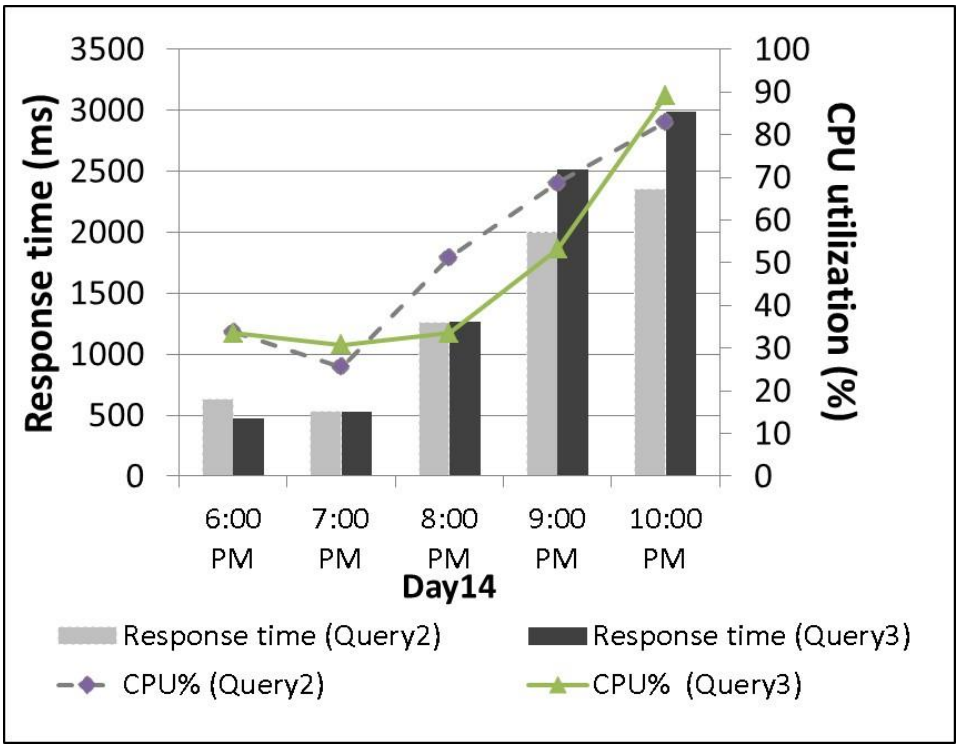


図 3.7　バースト時の問合せ応答時間

3.3.2 データバースト時のキャパシティコントロール手法の提案

解決策としては, 何らかの手段で送信されてくるツイートをフィルタリングして，処理可能な量のみを扱うようにするという方法が考えられる.

(手法 a) Random Filtering

従来のストリーム処理で用いられる，入力ツイートをある特定のレートでランダムにフィルタリングを行うナイーブな手法である． インプットのデータの流量と，サーバーの負荷などを考慮して，フィルタリングのレートがセットされる．

しかしながらこの場合, 各ツイートのリツイートがランダムフィルタリングされ，拡散ネットワークの経路が分断されてしまうことが懸念される．本システムが分析対象とするような，データ同士に返信や再共有等の関連のあるデータでは，このようにデータが欠損してしまうことは望ましくない．

(手法 b) Weight filtering (提案手法)

ソーシャルデータは，そのメッセージや発信時刻，その他にも，発信したユーザーの ID，フォロー/フォロワーのユーザー数など, 様々な情報が含まれている. そこで，ツイートに付随しているそれらの情報を用いて，ツイートに重要度を表す重み値を付与することを考える．これまでは，クエリのコストの大きさによって，コストが大きいクエリが発行される場合はフィルタリングのレートをあげたりする等実施し，レートを調整する[Wei06]等のキャパシティコントロールの手法が存在したが，ストリームデータそのものに付随している情報を用いてフィルタリングする手法は存在していなかった．

ソーシャルメディアの拡散データ分析というの観点においては, 拡散数が大きいツイートほどインパクトの大きいツイートであり，分析対象とする価値が高いと捉えられる．そこで，そのようなツイートの重みが高くなるように設定して，できるだけフィルタリングで削除されないようにしたい．すなわち，拡散数が大きくなりそうなツイートとそのリツイートのデータはできるだけフィルタリングせずに残すようにしたい．

以下にある観点においてのメッセージの重みを計算する式を表す.

$$Weight(t) = \begin{cases} getUserInfo(t_{RTID}), & if \quad t \quad is \quad retweet \\ getUserInfo(t_{ID}), & otherwise \end{cases}$$

*Weight(t)*はツイート，もしくはリツイート *t* の重み値を示す．*getUserInfo* メソッドは，*t* に付随する情報をインプットとして，重み値を返す．例えば本論文の実験では，ユーザーのフォロワーの数を返す．これは，フォロワー数が多いユーザーのツイートはリツイートされる頻度が高いという仮定のもとに設定している．送信されてきたツイートがツイートの場合はそれを発信したユーザーのフォロワー数を，リツイートの場合は，そのオリジナルツイートを発信したユーザーのフォロワー数を参照している．この重み値がある閾値を下回った場合，フィルタリングされる．

また，他の観点においても重み付けは可能である．例えば，ある地域の情報を取得するという観点であれば，メッセージに含まれる地名や発信位置・居住地の位置情報を基に，分析したい地域に関連するメッセージの重み値を高くすることで，それらのメッセージがフィルタリングされにくくなることが期待される．世界中で朝方に発信されたメッセージを中心に分析するという観点では，各メッセージが発信されたタイムゾーンとその該地域の現在時刻を基に，分析したい時間帯に該当するメッセージの重み値を高くする．

この閾値をあげて，多くのツイートをフィルタリングする程，分析結果の精度をさげてしまう可能性が大きくなる．これは，キャパシティのコントロールと分析結果の精度のトレードオフとなる．

これらのフィルタリング手法は，本システムで発行されるクエリの応答時間を監視し続け，あらかじめ決められたクエリ応答時間上限値を上回った瞬間に適用される．フィルタリングの割合は，その上限値を下回るようになるまで徐々にあげて調整していく．

3.4 まとめ

本章では，本研究で提案する，リアルタイムなソーシャルメディアデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えた，リアルタイムデータアクセス処理システムにおいての，逐次的なストリーム処理部分にのデータのメンテナンスと代表的な問合せパターンの最適化処理について述べた．

逐次的なデータを一時的に格納するインメモリデータストアを, 出来るだけフレッシュなデータを維持するために，各メッセージの流行が収束するタイミングを拡散モデルを用いて早期に推定し，時間ウィンドウ幅をカスタマイズしてインメモリデータストアをメンテナンスする手法を提案した．また，入力データ数がバーストした時に，各メッセージの重み値を計算して，その値に応じてデータをフィルタリングして処理量をコントロールする手法を提案した．

次章にてこれらの手法の効果を検証する．

第4章　リアルタイム処理最適化の検証

前章にて述べた，リアルタイムデータアクセス処理システムにおいてのデータアクセスの最適化手法の効果を，Twitter の実データを用いて評価する．リアルタイムデータアクセス処理では，アクティブに拡散が続いている新鮮なデータを対象にしたい．そこで前章ではストリームデータの収束のタイミングを議論し，カスタマイズしたタイムウィンドウによる，リアルタイム分析とオフライン分析との使い分けの手法を提案した．本章ではこのウィンドウ幅の決定が実際に収束のタイミングを捉えられるかを評価する．

次に，代表的な問合せシナリオの処理性能評価を行う．リアルタイム分析においては，クエリに対して早い応答性能が重要であるため，複雑な問合せパターンには計算の中間結果を保持したビューを導入して問合せの高速化を実現する．複雑な問合せの一つである，影響力の高いユーザーのランキング演算を対象に，文書集合の頻出単語を高速に計算するアルゴリズムを応用して，ビューを作成する．

最後に，データバースト時におけるキャパシティコントロールのためのデータフィルタリング手法の検証を行う．提案手法では，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセージをフィルタリングして処理するデータ量をコントロールする．これにより，問合せ結果の精度が高く維持しながら，コントロールが実現できることを目標とする．

4.1 実験データと環境

実験データは， 選挙関連のデータである，2014 年 1 月の東京都知事選挙の立候補者が発信したツイートとそのリツイート, 2014 年 12 月 の衆議院議員選挙の投票日に，政党名をメッセージに含んでいた日本語ツイート，リツイートを用いる．用いるツイート，リツイートの数は合計で 692,741 である．日本では 2013 年から，「インターネット選挙運動」が全国区で認められるようになり，候補者等がインターネットを通じて選挙活動を実施できるようになった．これにより，ソーシャルメディア上での選挙の話題がよりいっそう活発に発信されるようになった．データサイズは，選挙関連のトピックに絞った場合，1 時間あたり約 250MB のデータが発信されてきていた．

実験に用いるマシンは 2 x CPU Xeon X5670 (2.93GHz, 6 cores) with RAM 32 GB, OS は Red Hat Linux 5.5 を使用した.システムは Java (IBM J9 VM JRE 1.7.0)で実装した． インメモリデータベースは，H2 v1.4.184 をインメモリモードで使用し，RETWEET テーブルと ORIGIN_TWEET テーブルの，オリジナルツイートの ID

(TweetID, RTID)と，RETWEET テーブルの，オリジナルツイートを発信したユーザー(Src)にインデックスを張った．

4.2 カスタマイズしたウィンドウ幅決定の評価

4.2.1 実験シナリオ

3.1 で紹介した，拡散速度を考慮した拡散収束予測と，情報拡散モデルを用いた拡散収束予測の手法を評価する．

(a) 固定ウィンドウ幅

現在のストリーム処理の Tumbling window のモデルを拡張して，各ツイートのウィンドウ幅を固定値で設定する．あらかじめある時間幅を一つ設定して，オリジナルツイートの発信時刻からその時間が経過した時点て，拡散が収束したとみなす．リツイートデータを，リツイート元であるオリジナルツイートの ID 単位でデータを分けた後，それぞれに対してオリジナルツイートが発信されてから，指定されたウィンドウ幅の時間だけインメモリデータストアに格納される．指定した時間が経過した時点がウィンドウ幅の終点となり，そのオリジナルツイートとそのリツイートデータはすべてインメモリデータストアから退避する．

(b) 拡散速度を考慮した拡散収束予測実施法

まず，各ツイートのリツイートデータに対して，オリジナルツイートが発信されてから，何秒後に発信されたかを計算する．オリジナルツイートが発信されてから一定時間(= t)経過した時点で，時間 t あたりのリツイートの数を拡散速度とする．

$$\alpha = \frac{(\#\quad of\quad RT)}{t}$$

時間が経過するに従い，時間 *t* の対象する時間幅をスライドさせていき，現在時刻から，過去 t 時間までさかのぼった領域が，計算対象となる．速度の計算は，一定時間おきに連続的に実行する．

この α が閾値を下回った時点で，拡散が収束したとみなし，ウィンドウの終点とする．

(c) 情報拡散モデルを用いた拡散収束予測実施法

ツイートが発信されてから 1 時間のデータを用いて対数正規分布にフィッティングし，拡散の収束時間を推定する．オリジナルのツイートが発信されてから 1 時間経過した時点でのそのツイートに対するリツイートの集合を用いて，オリジナルツイートが発信された時刻を 0 とし, 各リツイートが発信された時刻との差分(ms)を計算する．この入力データをもとに，データに尤もらしい確率密度関数の μ と σ を推定する．本実験では，R[R]の fitdistr 関数を用いて最尤推定を行う．

```
result <- fitdistr(input_data, "log-normal")
```

得られた確率密度関数に対して，上側確率の 90%に対応する点を求め，そのツイートがほぼ収束する時間であるとみなし，ウィンドウ幅の終点とする．

```
window_last <- qlnorm(.9,meanlog=result[1]$estimate[1], sdlog=result[1]$estimate[2])
```

これにより, ツイートがリツイートされなくなるまでを全体とした 90%程度をカバーする時間を推定できると期待する．すなわち，この点がそのツイートのウィンドウ幅の終点となる．終点に到達した拡散データは，インメモリデータストアから退避され，リアルタイム分析の対象ではなくなる．

それぞれの手法において，ウィンドウ幅をカスタマイズしたときに，そのウィンドウ内には，各ツイートの拡散データ(＝リツイートデータ) は，総リツイート数のう

ち何%のリツイートをカバーできていたかを評価する. 総リツイート数は各ツイートが発信されてから約 1 週間経過した時点でのリツイート数を使用する．ツイートがリツイートされなくなるまでを全体とした 90%程度をカバーすることを目標とする.

4.2.2 実験結果

実験データは，期間 2014/1/19 – 2014/2/11 における，2014 年東京都知事選挙の立候補者が発信したツイートとそのリツイートのデータを用いる．発信されたメッセージのうち, リツイート件数が多かったトップ 50 ツイートとリツイートのデータを実験対象とする．合計総リツイート数は 13,310 である．(a) 固定ウィンドウ幅については，あらかじめ指定する時間 (= *T* 時間)を様々に変えて測定する． (b) 拡散速度を考慮した拡散収束予測 については，閾値を 0 とし， *t* =10, 20, 40, 60 (min) と変えた時の平均カバー率を測定し，目標である 90％のカバー率に最も近い t=60 の結果を採用している．速度の計算は，1 分単位で連続的に実行した.

表 4.1 に結果を示す．各ツイートが発信されて 1 時間経過した時点では，平均して総 RT の約 55% までしかまだ到達していないことが分かる. 11 時間経過した時点で，手法(b)と同等の約 88% をカバーできている．手法(a)では，各ツイートが発信されて，一律 11 時間経過 i するまでのリツイートをインメモリデータストアに保持することを意味する．手法(c)は手法(b)よりもやや高いカバー率であり，目標の 90% に近い結果を得られている.

表 4.1.拡散の平均カバー率

(a) *T=1*	(a) *T*=11	(a) *T*=12	(b) 拡散速度	(c) 拡散モデル
55.0%	87.6%	90.4%	**88**%	**89.9%**

図 4.1 は，50 件のツイートそれぞれの，実際にリツイート数が総リツイート数の 90%に到達した時点の，オリジナルツイートからの経過時間と，手法(c)の拡散モデル推定によりリツイート数が全体の 90%に到達する時間を推定した時点の，オリジナルツイートからの経過時間を表している．我々の手法は，実際の時間と近い結果を取得できていることが分かる．但し，3 件のツイートに対しては実際の経過時間が推定よりも 2 倍以上長い．これらのツイートは，数日に渡って数件ずつリツイートされ続けたためであり，我々の手法はオリジナルツイートが発信されてから 1 時間までのデータを用いて推測するため，推定しきれていない．これらのツイートに対処するには，(b)の拡散速度の計算を用いて，ウィンドウ終了地点で本当に収束しているかチェックする等の改善が考えられる．

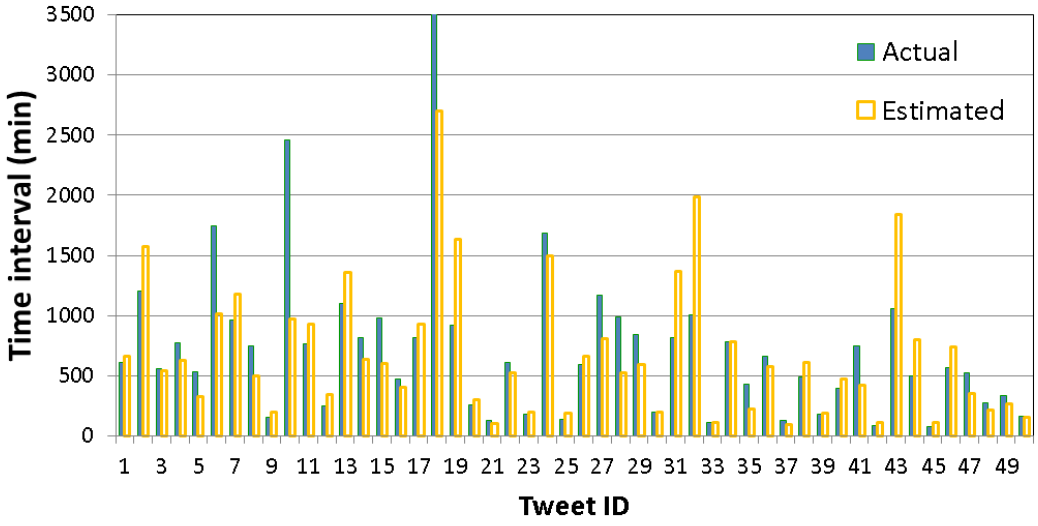


図 4.1 各ツイートの総リツイート数の 90%のリツイート数をカバーした実際の時点と手法(c)で推定した時点

次に，各ツイートが実際に 90%の RT 数をカバーした時点の，オリジナルツイートからの経過時間と，各手法で推定した 90%点のオリジナルツイートからの経過時間との差を比較する．差が小さいほど，実際の 90%の点に近い結果が得られていることを意味する．まず，手法(a)の T=11 のときの結果は平均して 535 分であった．手法(b)は 510 分，手法(c)は 368 分であった．これは，手法(a), (b)においては，平均して 90%近くのカバー率を出せているが，実際に拡散が収束した時点よりも長いウィンドウ幅が設定されているツイートが多く，既にアクティブに拡散していないデータがインメモリデータストアに残り続けることになる．また，手法(a)と手法(b)の時間(T または t)や閾値の値は自明ではなく，データセットによって適切な値を決定しなくてはならない．対して手法(c)は各ツイートの初期の拡散情報をもとに自動的に推定できる利点がある．

手法(c)の拡散推定が他のデータセットでも有効性を示せるかを検証する．もう一つのデータセットである，2014 年 12 月 の衆議院議員選挙の投票日に，政党名をメッセージに含んでいた日本語ツイート，リツイートを用いる．衆議院議員選挙投票日の 20 時に発信されたツイートの，総リツイート数上位 100 件のツイートに対して，ツイート発信時間から 1 時間以内のリツイートの分布を対象に拡散の収束を推定した．先のデータの実験と同様に，推定された拡散収束時間の時点で，各ツイートの総リツイート数のうち何%のリツイート数をカバーできていたかを計算する．

結果を表 4.2 に示す．こちらのデータセットでも，平均 90.2%のカバー率を出すことができている. 比較として, 手法(a)の T を 2,3,4 に変更したときの結果を見ると，手法(a)では T=4 のときに平均 90.4%のカバー率に到達している．先のデータセットと比較すると，T の値は小さくなっている．これは，先のデータセットが，選挙活動期間中のデータだったことに対して，本データセットは，選挙開票日のデータセットであるため，より情報の移り変わりが激しく，拡散スピードとその収束がは速かったのではないかと考えられる．各ツイートが実際に 90%のリツイート数をカバーした時点の経過時間と，手法(c)で推定した 90%点の時間との差は平均で 252 分であ

った．一方で，一律 4 時間で固定した時の差は平均で 412 分であった．よって，一定のウィンドウ幅で固定するよりも，本手法のほうが高い精度で 90%の点を推定できていることが分かる．

この結果から，手法(c)の拡散モデルの推定は，このデータセットにおいても有効だったと言える．さらに，手法(a)の T は前回よりも短くなっており，あらかじめ適切なウィンドウ幅を固定で与えることは難しいことが分かる．

表 4.2 .拡散の平均カバー率

(a) T =2	(a) T =3	(a) T =4	(c) 拡散モデル
82.5%	87.6%	90.4%	**90.2**%

図 4.3 は 2 つ目のデータセットで推定した 100 件のツイートそれぞれの，実際にリツイート数が総リツイート数の 90%に到達した時点の，オリジナルツイートからの経過時間と，本手法によりリツイート数が全体の 90%に到達する時間を推定した時点の，オリジナルツイートからの経過時間を表している．我々の手法は，実際の時間と近い結果を取得できていることが分かる．但し，前実験と同様に数件のツイートに対しては実際の経過時間が 2000 分をオーバーしているものがあり，前述のような改善を入れることにより，それらを救える可能性がある．

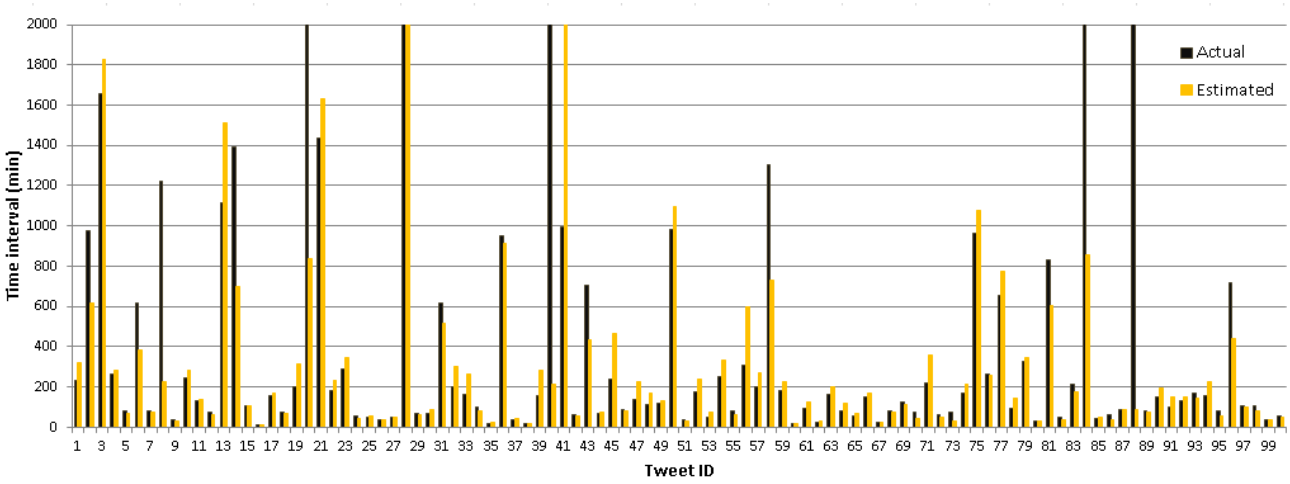


図 4.2 各ツイートの総リツイート数の 90%のリツイート数をカバーした実際の時点と本手法で推定した時点

以上，カスタマイズしたウィンドウ幅が，実際に拡散データが収束するまでをカバーできているかを実験し，拡散モデルを利用することの効果を示した．

4.3 問合せ性能評価

4.3.1 実験シナリオ

3.2 で述べた代表的な問合せクエリに対して性能評価を行う．各 Query 供，3.2 に掲載した SQL を用いる．

[Query1] 指定したツイートの拡散ネットワークを取得

[Query2] 人気のあるツイートを取得

[Query3] キーパーソンとなるユーザーを取得

各回とも，10,000 回問合せた時の 1 問合せあたりの平均処理時間を計算する(ただし，最初の 1,000 回は平均計算には含めない)． Query 1 は，WHERE 句で TweetID を 1 件指定した時の結果である．Query 2, 3 は TweetID を重複無しで 100 件指定した時の結果である．TweetID は実験データの中からランダムに選択されてる．

またユーザーランキング計算のためのビューを導入したときの効果を検証する．

4.3.2 実験結果

図 4.3 に Query 1, 2, 3 の問合せ処理時間の結果を示す．平均処理時間は Query 1 が 0.2ms，Query 2 が 0.4ms であり，高速に問合せ処理ができている．一方で，Query 3 は 5.6ms であり，前者のクエリと比較して問合せ処理に時間が大幅にかかっている．これは集約演算のオーバーヘッドによるものであると考えられる．

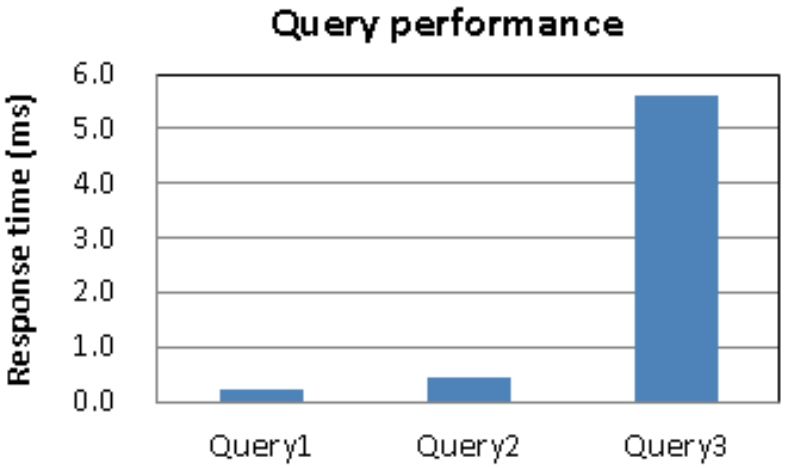


図 4.3　各 Query の平均応答時間

次に, Query 3 にランキング前処理のビューの導入前後の性能比較結果を図 4.4 に示す. WHERE 句で指定する TweetID の数を 100 から 500 に変化させたときの問合せ処理時間を測定した．ビューを導入することにより，問合せ処理時間を大幅に短縮できている．

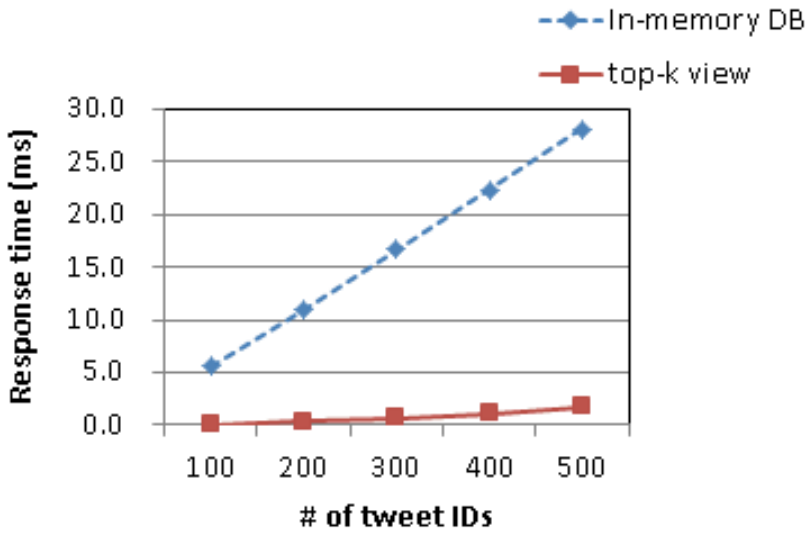


図 4.4　　ビュー導入前後の性能比較

4.4 バースト時のキャパシティコントロールの評価

4.4.1 実験シナリオ

3.3 で提案した，データバースト時のコントロール手法についての評価を行う．以下の 2 手法にて実験結果を比較する．満たすべきクエリの応答時間の最大を約 1 秒(900ms～1100ms)とし，各時間帯においてそれを満たすようになるまでフィルタリングの割合を調整する．

(手法 a) Random Filtering

入力ツイートをある特定のレートでフィルタリングを行うナイーブな手法である．0%削減の状態から，応答時間が約 1 秒以下になるまで 5%ずつフィルタリングレートをあげていく．

(手法 b) Weight filtering (提案手法)

重み値としてユーザーのフォロワー数を用いる．閾値は応答時間に応じて設定する．すなわち，ツイートの発信ユーザー（リツイートの場合はオリジナルツイートの発信ユーザー）のフォロワー数が閾値よりも小さい場合，そのツイートはフィルタリングされる．閾値は 0 から，応答時間が約 1 秒以下になるまで 100 ずつ増やしていく．

4.4.2 実験結果

4.4.2.1 バースト時の問合せ性能評価

問合せ性能評価として，3.2 で紹介した問合せパターンである，Query 2 と Query 3 の SQL を用いて応答時間を測定した．100 ユーザーが同時にこれらのクエリを発行してくると想定して測定する．WHERE 句で指定するツイートの ID は，我々の実験データの中で，“自民党”というキーワードが含まれるツイートの ID のリストをセットする．これは，選挙に関するリツイートのデータを対象にして，自民党に関するツイートの中で，どのツイートが最もリツイートされていたのか (Query 2)，どのユーザーが最もリツイートされたか (Query 3) のランキング結果を取得することになる．

100 ユーザーは，20, 21, 22 時台のデータを対象に Query 2 と Query 3 の SQL をそれぞれ 10 回発行し，平均応答時間を計算する．図 4.5 は各時間帯での Query 2 の応答時間の中央値を示している．

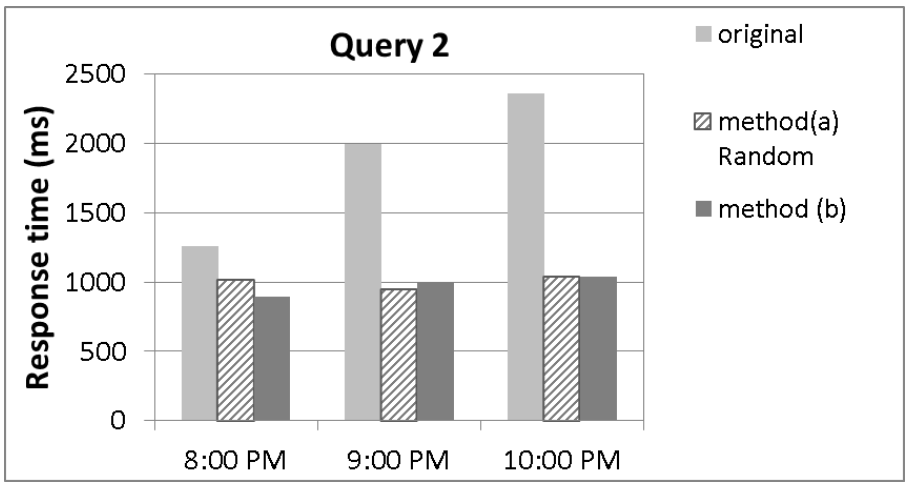


図 4.5　Query 2 の応答時間

全手法において，フィルタリングを行うことにより約 1 秒以下の応答時間を満たすまでに下げることができている．同様に，Query 3 の結果も応答時間を下げることができている (図 4.6)．図 4.7 はそれぞれの時間帯で実際にセットされたフィルタリングの値を示している．午後 10 時のフィルタリング率がもっとも高く，手法 a では 70%のランダムな削減，手法 b では閾値 1700 でのフィルタリングとなっている．

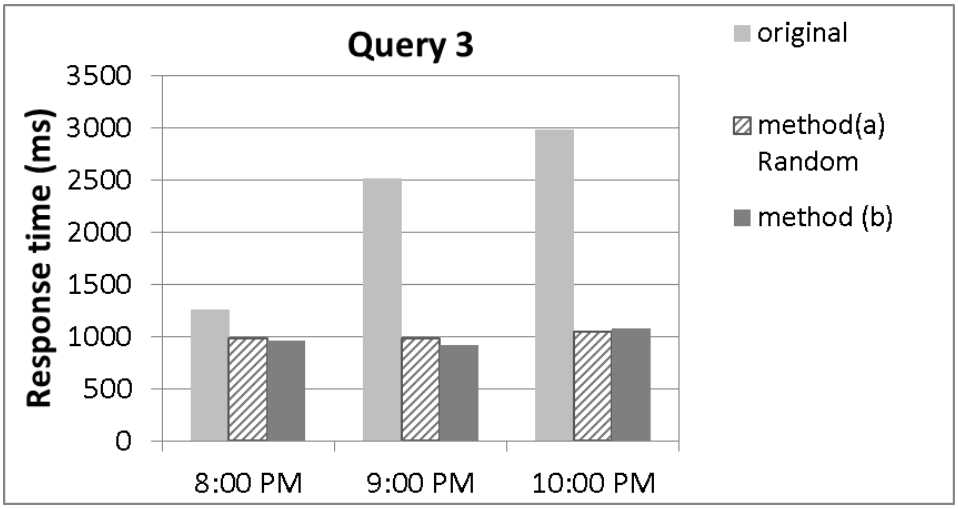


図 4.6　Query3 の応答時間

	8:00 PM	9:00 PM	10:00 PM
method (a) sampling rate	30% cut	35% cut	70% cut
method (b) threshold	400	1000	1400

図 4.7　各時間帯のフィルタリング率

4.4.2.2 問合せ結果の精度の評価

フィルタリングの手法を適用することにより，データ数が削減されて問合せ性能が改善されたが，一方で，フィルタリングしたことにより，問合せ結果がオリジナルのデータを用いた時と異なってしまう可能性があるというトレードオフがある．そこで，フィルタリング手法を適用したときの Query 1, 2, 3 の結果が，オリジナルのデータを用いたときの結果と比較して，どの程度結果を維持できていたかを検証する．

図 4.8 は，各クエリでの，オリジナルデータの結果のカバー率を表している．100%の場合は，オリジナルデータの結果をそのまま再現できたということを意味する．Query 1 は，指定したツイートの ID のリツイートデータをすべて取得する問合せである．すなわち，指定したツイートの拡散ネットワークを生成するための問合せとなる．ツイートの ID には，Query 2 の結果である総リツイート数トップ 100 件の ID を指定する．Query 2, 3 は，それぞれ総リツイート数，ユーザー数のトップ 100 件の結果を返すランキング問合せである．したがって，オリジナルのトップ 100 の結果のうち，何件をトップ 100 以内に維持できていたかの割合を計算している，

Query 1 のカバー率は，WHERE 句で指定した ID の拡散ネットワークのエッジ総数のうち，何%を維持できたかを意味する．各ツイートの拡散ネットワークは，2.1 節で示したように，{リツイートされたユーザー，リツイートしたユーザー}を 1 エッジとするデータ構造となっている．手法 a は約 70%のエッジを失った結果となっている．一方で手法 b は約 90%,のカバー率を維持できている．

Query 2 と Query 3 の結果においては，手法 a は 90%以上の結果を維持できている．これは，手法 a はランダムに一様にデータを削減するために，もともとリツイート数の多いツイートやユーザーは，相対的に多く残ったためと考えられる．一方，手法 b は，約 80%であり手法 a よりも低い結果となっている．この要因の一つとして，フォロワー数が少ないユーザーが発信したツイートでも，フォロワー数の多いユーザーにリツイートされたことにより多くのユーザーがそのツイートを閲覧することになり，ツイートが広く拡散した場合があると考えられる．

手法 b のカバー率は，リツイートしたユーザーの属性も考慮すると，より改善できる可能性がある．現時点では，オリジナルツイートを発信したユーザーの属性しか考慮していない．

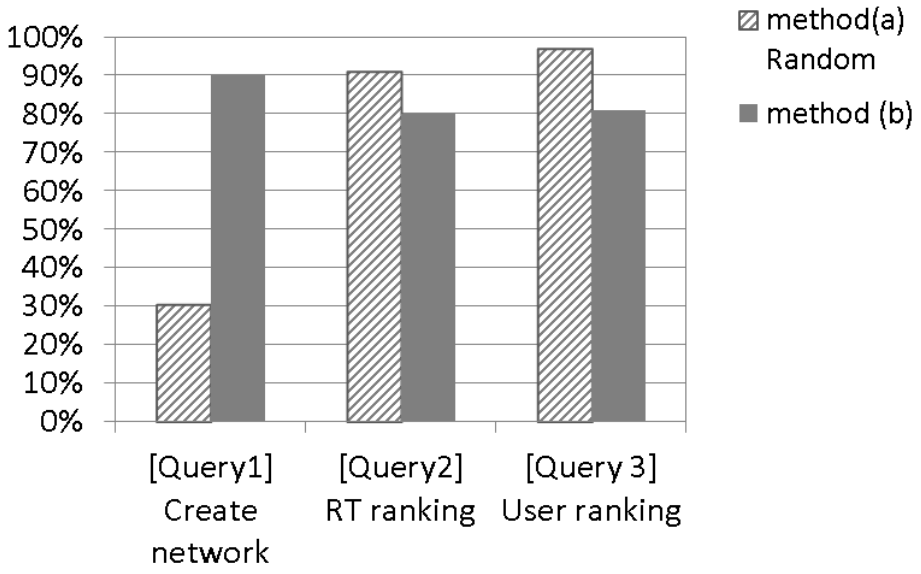


図 4.8　オリジナルの結果のカバー率

そこで, 提案手法を改良して, オリジナルツイートのユーザー情報のみならず, リツイートしたユーザーの情報も使用する．具体的には，リツイートのユーザーの重み値が閾値以上になった場合，それ以降，そのリツイートのオリジナルツイートをリツイートしたデータは全て閾値以上の重みをみなしてフィルタリングされないようにする．必然的に，フィルタリングされないデータが増えるため，閾値の値は改良前よりも大きくなる．実際に 22 時台にて本改良手法を適用した時，閾値は 2500 の状態で目標の応答時間を維持することができた.

このときの，問合せ結果の精度を図 4.9 に示す．全 Query において 90%以上の精度となり，前結果で手法(a)が勝っていた Query2, 3 に関しても同等化それ以上の精度となった.

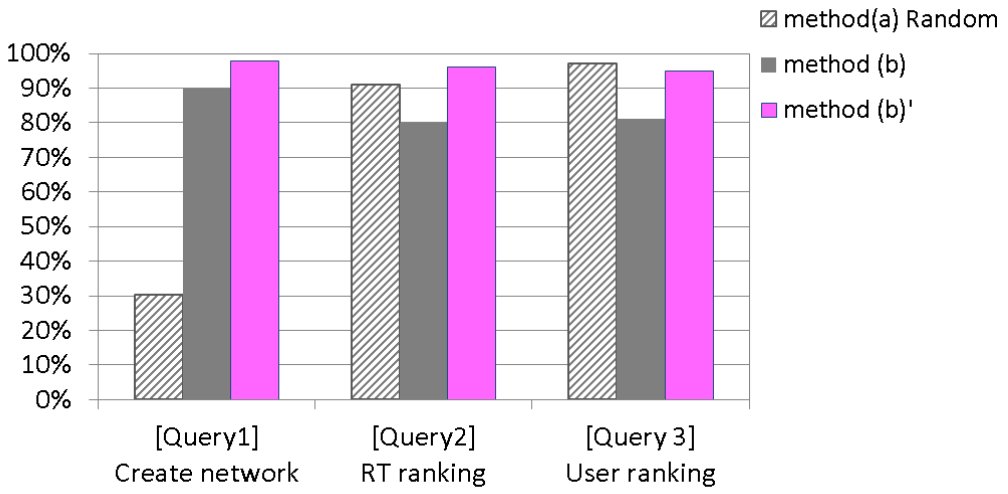


図 4.9　提案手法の改良版のカバー率評価

本実験において，バースト時のキャパシティコントロールのため，ユーザーの属性情報を用いてデータ数を制御することによる効果を示した．

4.5 まとめ

本章では，本システムにおける，リアルタイム処理部分においてのデータアクセスの最適化手法の効果を Twitter の実データを用いて評価した．リアルタイム処理対象である，今も拡散され続けているアクティブなデータを出来るだけその拡散のつながりを分断することなく維持するため，カスタマイズしたタイムウィンドウを導入し，従来の固定ウィンドウ幅と提案手法とで精度を比較した．提案手法である拡散モデルを用いた収束推定は，拡散が始まって収束するまでの時間幅の推定を90%程度カバーでき，かつ収束判定の正解値（実際の収束時点）との誤差を最も小さく抑えられていることを示した．また，他のデータセットにも適用し，同程度の効

果を得られることを示した．今後においては，他のトピックについてのデータセットを使う等して，本手法の有効性をさらに検証したい．

次に，代表的なリアルタイムの問合せパターンを実際に測定し，インメモリなデータアクセスは高速にクエリが実行できることを示した．しかし，複雑な問合せパターンは演算のオーバーヘッドがあり，単純なクエリより応答時間が長くなっていた．そのようなクエリに対しては，計算の中間結果を保持したビューを導入して問合せの高速化が実現できることを示した．また，データバースト時のクエリの応答時間の劣化を回避するため，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセージをフィルタリングして処理するデータ量をコントロールする手法を検証した．これにより，従来のランダムなフィルタリングと比較して，同程度のサンプリング量において，より問合せ時間が短く処理できていることを示した．また，問合せ結果の精度が高く維持できていた．問合せ性能については，データ更新時におけるビューの更新のオーバーヘッド等を今後検証したい．

本実験で用いたデータは，選挙のトピックに絞っていたが，例えば日本語の拡散データを全て扱おうとすると，public な Twitter API (データ取得量があらかじめサンプリングされている)を用いた場合，1 時間あたり約 6GB のデータ量となる．従って，データ処理を分散させるなどの拡張が必要となると考えられる．また，問合せは連続的に発行されることが想定される．その場合，前回の計算結果との差分のみを計算することにより，問合せは時間さらに最適化できることが期待される．例えば人気のあるツイートのランキングは，前回結果と同じデータがランキングにまだ残っている可能性がある．このように，ストリームの特徴と，ソーシャルデータの特徴を考慮した問合せの最適化についてもさらに考えていきたい．

本論文では，ソーシャルメディアのデータを対象にしてストリームデータアクセス処理の最適化について述べ，効果を検証した．実験データには Twitter のデータ

を用いたが，最適化に利用した，データのバースト性やメッセージの再共有による情報拡散の特長は Twitter のみならず，マイクロブログのようなソーシャルメディアデータに共通の特徴であるため，他のソーシャルメディアを分析対象にした場合も同様の最適化の効果があると考えられる．

また，本最適化は，他の種類のストリームデータにも適用可能である．例えば，ある有料道路区間を走る車のセンサーデータは，時間帯や道路状況によって，データの流量が大きく変化すると考えられる．これらも一種のバースト性を引き起こすと考えられ，キャパシティの制御が必要になったときには分析したい観点によって，優先度の高い車の重み値を高くしてフィルタリングすることで，問合せ精度の低下を防げることが期待できる．また，それぞれの車が，有料道路に入ってから出るまでをストリームのウィンドウとしてカスタマイズすれば，その車の時系列データを分断させることなくリアルタイム分析対象にすることができる．

第5章　データベースアクセスの最適化

ストリーム処理システムが世の中に普及し始めた 2000 年前半では，センサー等のリアルタイムデータはシステム内で演算処理をされた後に，一般的に破棄されていた．それは現在の状態における何らかの演算結果が重要であり，ストリーム処理システムはそのような要件を満たすためのシステムであるからである．また，ストリームデータはリアルタイムに到着し続けるため，全データをストレージに格納することは現実的ではなかった．ところが，近年のストレージの格納量の大規模化，SSD 等によるデータ入出力処理の高速化にも伴い，ストリームデータもそのままストレージに格納され，それをオフラインで分析対象にすることも増えてきた[Azure]．しかしながら，ストリームデータのリアルタイムの分析とストレージに格納したオフラインでの分析を同時に考えたシステムは検討されていない．

そこで本研究では，リアルタイムなソーシャルメディアデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備たリアルタイム処理システムを構築し，データアクセスの処理性能を中心にシステムの最適化を行う．本章では，蓄積されたデータを処理する機構において，蓄積された大規模データを高速に問合せ処理するためのキャッシュの導入と，データ更新の影響を出来るだけ抑えるためのキャッシュメンテナンス手法を提案する．

ストリームデータが逐次的に処理された後，オフラインでのデータ分析に利用するため，ハードディスクのストレージを用いたデータベースへ蓄積される．これらのデータを基に，ユーザーのプロファイル分析等が実施されることが想定される．このような OLAP (online analytical processing) 処理は，データベースに発行されるクエリのパターンも多様化，複雑化する．データもハードディスクから取り出すことになるため，逐次的なデータ処理での問合せ時よりもデータアクセス処理時間が長くなる．代表的な高速化手法は，メモリ上にデータの一部をキャッシュしたり，データベースのテーブルの結合結果や集約をビューとして保持することである．これらの多くの高速化手法は，データベースがほとんど更新されないことを前提としている．なぜならば，データベースとの整合性を保つため，データベースのデータが更新された場合は，キャッシュしたデータも更新しなくてはならない．データベースの更新頻度が多くなると，データベースとキャッシュのデータ双方の更新処理は，逆にオーバーヘッドになってしまう．

本研究で提案しているシステムは，リアルタイムに受信したソーシャルデータを，逐次的に処理した後にデータベースに蓄積される．この蓄積する処理を数分単位でのバッチ処理で実施したとしても，それなりの頻度の更新が想定される．データアクセスの高速化に寄与しやすいデータを出来るだけキャッシュに残し，かつ低コストでメンテナンスすることが理想的である．

そこで本章では，データベースとの整合性を保ちながらも，可能な限り多くの問合せ結果のデータキャッシュを保持するため，クエリパターンの依存性を分析して，データ更新時のキャッシュの無効化範囲の影響を小さくする手法を提案する．さらに，データを無効化する時に使用するインデックスを導入することで，データベースの更新に対応して無効化すべきキャッシュデータを特定するための手法を提案する．また，情報拡散データのようなグラフ構造のデータを高速に取得するには，ビットマップ形式のインデックスを使用することが効果的である．しかし，このインデックスは大規模な粗行列になることが想定されるため，行列圧縮の手法を適用してより小さいサイズでインデックスを生成する．より圧縮率を高めるため，同時に参照されそうなソーシャルメディア上のユーザーをできるだけグループ化して圧縮が効きやすくするための手法を提案する．

5.1 Apache OpenJPA による EJB3.0 コンテナの利用

図 2.3 に示したように，本システムでは，各分析モジュールは，所望のデータを取得するために，アプリケーションサーバー内にある”Data access layer”を介してデータベースへアクセスする．データベースへのアクセスには，O/R マッピングの Java 用フレームワークである JPA(Java Persistence API)を用いる．これにより，各分析モジュールは DB の種類に依存することなく，Java のオブジェクト操作でデータをやり取りすることができる．また，メモリ上にデータをキャッシュする機構を持つため，データベースへのデータアクセスを高速化できることが期待される．

5.1.1 OpenJPA

EJB(Enterprise JavaBeans) [E6]はビジネスロジックをモデル化するために，ネットワーク分散型ビジネスアプリケーションのサーバサイドで JavaBeans と同様のものを実現した仕様である．EJB は業務システムにおいて，組織内に存在する永続データを分散オブジェクトとして扱うことを可能にし，安全性，柔軟性を保ちながらさまざまな業務システムで利用できるため，アプリケーション開発に多く使用されてきた．

しかし，EJB2.1 以前ではその開発の複雑さとメンテナンスのコストが開発者への大きな負担となっていた．例えば，EJB コンポーネント定義の多数のインタフェースの実装義務，EJB 独自の規約，煩雑な設定ファイル，単体テストの困難性等があげられる．また，性能上の側面からも，EJB を用いた方が，EJB を用いずに JDBC を直接使用したアプリケーションよりもスループットが低かった[Em02]．そこで 2006 年にリリースされた EJB3.0 では開発者の負担を軽減するための大幅な仕様変更が施行され，JDK 1.5 で新たに追加されたアノテーション記述の採用等，開発者は EJB コンポーネントを POJO(Plain Old Java Object)ベースで簡単に利用できるようになった．

EJB3.0 の仕様の一部である JPA(Java Persistence API)は，POJO ベースの O/R マッピングアーキテクチャであり，JDBC を直接使用したアプリケーションよりも少ないプログラムコードでデータにアクセスすることができる．さらに，SQL-like な JPQL(Java Persistence Query Language)言語を使用することにより，JOIN,ORDER BY,LIKE 演算子等を用いるような複雑な問合せも可能になった．

図 5.1 に示すように，アプリケーションは JPA が提供するインタフェースを介してデータアクセスを行う．JPA の実装には，Apache OpenJPA[AP]，Hibernate[Hi]，TopLink Essentials[To]等がある．これらは JPA の仕様に加え，アプリケーション側でデータベースへの問合せ結果をキャッシュに入れる機能を有効にすることができ，例えば OpenJPA ではデータベースのテーブルのデータを DataCache，JPQL からの問合せとその結果として返したデータの ID リストを QueryCache に入れておくことが出来る[Pat07]．これらのキャッシュのふるまいは OpenJPA が管理してくれるため，開発者はキャッシュの機能を設定ファイルから ON に指定するだけで，更なる高いパフォーマンス結果を得ることが期待できる．しかし，更新トランザクションが発生した場合はデータの一貫性を保つために，OpenJPA はキャッシュ上のデータを更新，削除等のメンテナンスの必要が生じ，これがオーバーヘッドとなる可能性がある．

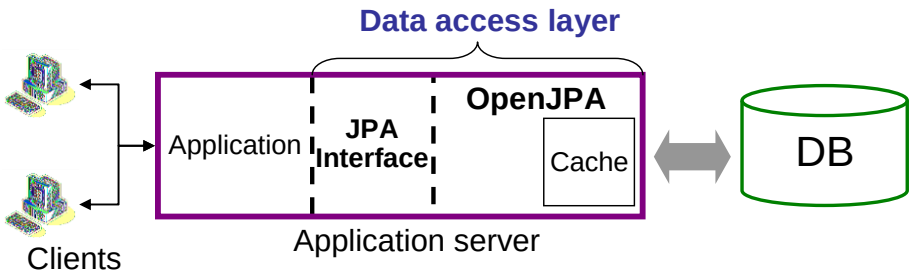


図 5.1　JPA のアーキテクチャ

本節では, OpenJPA のデータ更新方法を変更することによる更新トランザクション時のキャッシュメンテナンス方法の違いとそれに伴う性能への影響について紹介する.

5.1.2 Java Persistence API (JPA)データマッピングとデータ取得

JPA とは，EJB3.0 で規定された O/R マッピングであり，API の定義には JPQL と O/R マッピング用のメタデータ定義が含まれている．データベースのテーブルが Java エンティティオブジェクトに関連付けられ，クライアントプログラムはエンティティを操作することによって，データの挿入，取得，更新，削除を行う.

図 5.2 は，データベースの Item テーブルを，Item エンティティクラスにマッピングする例を表している．テーブル Item には，本屋の本データが入っており，属性として{ID, TITLE, A_ID, PUBDATE, COST, STOCK}を持ち、それぞれ{主キー，本のタイトル，作者の ID，出版日，価格，在庫数}に対応する.

Item エンティティクラスには，アノテーションを用いてマッピングを行う．エンティティクラス定義には，@Entity を記述し，マッピングするデータベースのテーブル名を@Table で指定する．クラスの中にはテーブルの属性に相当するフィールドが定義される．主キーに対応するフィールドには@Id アノテーションをつける．

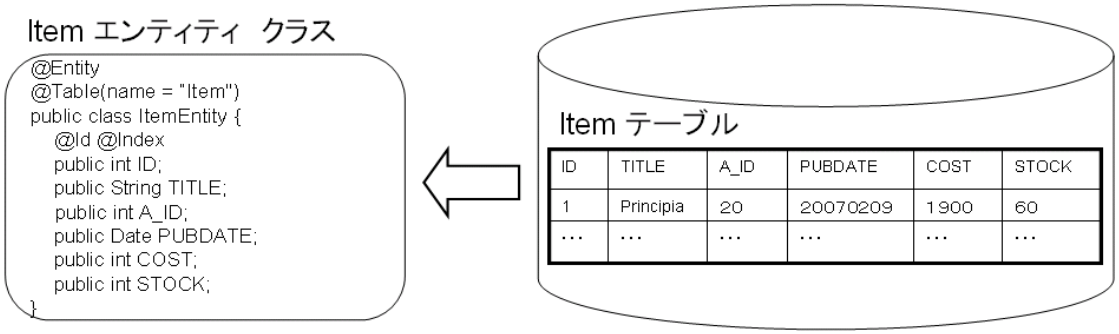


図 5.2　Item テーブルから Item エンティティクラスへのマッピング

EntityManager のメソッドからのデータ取得

図 5.3 は，エンティティを操作するためのインタフェースである，EntityManager のメソッドを使用して，データベースの Item テーブルにある主キー値=29 の Item データを取得する例を表している．

まず，EntityManager を取得した後，find メソッドで取得したいデータのエンティティクラス名と，主キー値を指定する．実際のデータベースアクセス部分については，JPA が SQL を発行して JDBC アクセスを実行してくれるために， データベースアクセス処理プログラムを自ら書く必要がなく，簡単にデータをデータベースから取得することが出来る．

```
1.    public void getItem() {
2.      EntityManager em = factory.createEntityManager();
3.      em.getTransaction().begin();
4.      ItemEntity item  = (ItemEntity)em.find(ItemEntity.class, 29);
5.      em.getTransaction().commit();
6.    }
```

図 5.3　EntityManager のメソッドを使用した Item データ取得例

JPQL からのデータ取得

JPQL(Java Persistence Query Language)を使用しても同様の問合せを実行できる. 図 5.4 は，JPQL を用いた同様の処理の例を表している.

JPQL はエンティティに対する SQL-like な問合せ言語であり，バックエンドで使用するデータベースに適した SQL に変換され，データベースに問合せが実行される．SQL はデータベース製品ごとに方言が存在するが，JPQL を使用する場合はその方言を意識することなく，JPA が使用しているデータベースにあわせた SQL に変換して問合せを発行する．したがって，もし使用データベースを変更しても，クライアントは設定ファイルに記載した JDBC ドライバの記述等を変更するだけで対応可能であり, プログラムの高い移植性を実現できる. また, JPQL を用いることにより, JOIN,ORDER BY,GROUP BY,HAVING 等複雑な問合せも記述できる.

EntityManager, JPQL のどちらのデータアクセス手法も，データの照会のみならず更新・削除も可能である.

```
1.    public void getItembyJPQL() {
2.      EntityManager em = factory.createEntityManager();
3.      Query query = em.createQuery("SELECT item FROM
                              ItemEntity item WHERE item.I_ID=?1");
4.      em.getTransaction().begin();
5.      query.setParameter(1, 29);
6.      ItemEntity item = (ItemEntity) query.getSingleResult();
7.      em.getTransaction().commit();
8.    }
```

図 5.4　JPQL を使用した Item データ取得例

5.1.3 OpenJPA のキャッシュと無効化における問題点

OpenJPA はアプリケーション側にデータベース問合せ結果のデータと問合せの情報をキャッシュしておく機能を独自に持つ．これにより，データベースへのアクセスを高速化することが期待される．以下にその機能の概要と，異なるデータ更新方法によるキャッシュメンテナンス手法の違いについて述べる．

5.1.3.1 DataCache and QueryCache

OpenJPA は，問合せ結果として返したエンティティを，DataCache の中に保持する．JPQL から発行された問合せは，その問合せと結果として返したエンティティの ID リストを QueryCache の中に保持する．図 5.5 は，OpenJPA における DataCache と QueryCache の例である．

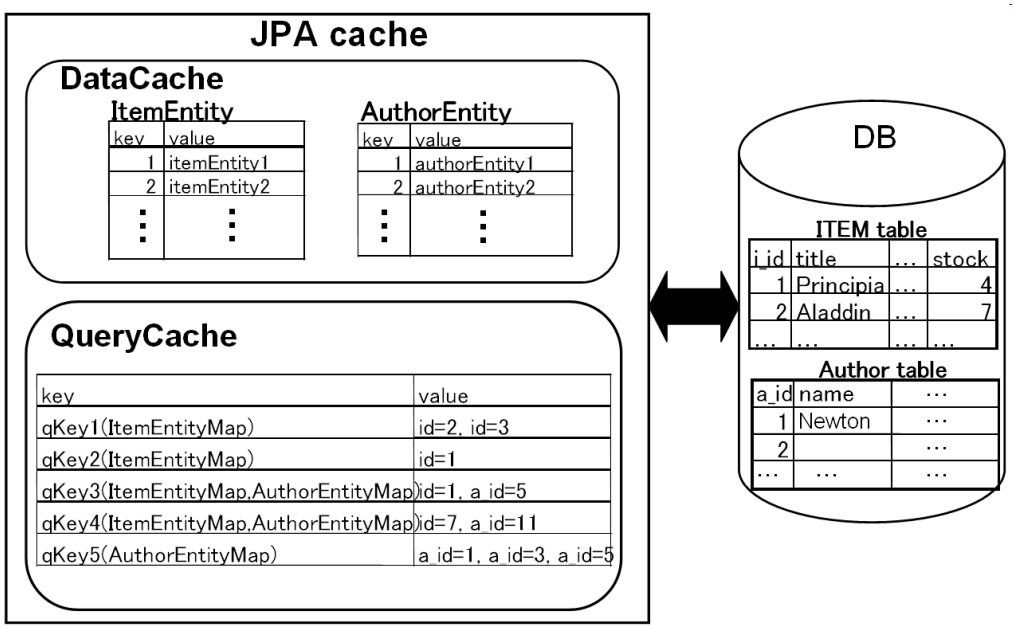


図 5.5　OpenJPA の DataCache と QueryCache

DataCache では，各エンティティごとにデータを保持している．図 5.5 においては，データベースに存在する Item テーブルと Author テーブルが，それぞれエンティティの ID(対応するテーブルの主キー)となる値を key に持ち，エンティティの実体オブジェクトを value として保持している．

QueryCache では，JPQL からの問合せ結果をまとめて格納している．JPQL の問合せに一意な値(key)を割振り，問合せの結果として返したエンティティの ID もしくは属性値をリストにして value に保持している．結果として返したエンティティの ID が DataCache の中に存在しなかった場合は，DataCache にそのエンティティを同時に格納する．また QueryCache には，value の中に入れたエンティティが属するエンティティクラス名をアクセスパスとして key の一部として保持しておく(図 5.5 の QueryCache の key の中の()の部分)．エンティティクラス名を保持することで，更新トランザクション時の，キャッシュとデータベースのデータ一貫性保持のためのキ

ャッシュの無効化(invalidation)に使用することができる．また，現時点の OpenJPA 実装上の制約により，JPQL の問合せすべてが QueryCache に入るわけではない．

5.1.3.2 異なるデータ更新手法による DataCache のメンテナンス手法の違い

5.1.2 にて JPA のデータアクセス手法には EntityManager のメソッドを使用する場合，JPQL を使用する場合の 2 通り存在することを前述したが，それぞれの手法において OpenJPA の DataCache のメンテナンス手法が異なる．例として， Item テーブルの，主キー値=29 のレコードに対し，そのレコードの STOCK の値を“5”に変更する場合のキャッシュのメンテナンス手法について示す．

<a> EntityManager のメソッドを使用して更新した場合

```
1.  public void updateItem() {
2.    EntityManager em = factory.createEntityManager();
3.    em.getTransaction().begin();
4.    ItemEntity item  = (ItemEntity)em.find(ItemEntity.class, 29);
5.    item.setStock(5);
6.    em.persist(item);
7.    em.getTransaction().commit();
8.  }
```

図 5.6　EntityManager のメソッドを使用した Item データ更新例

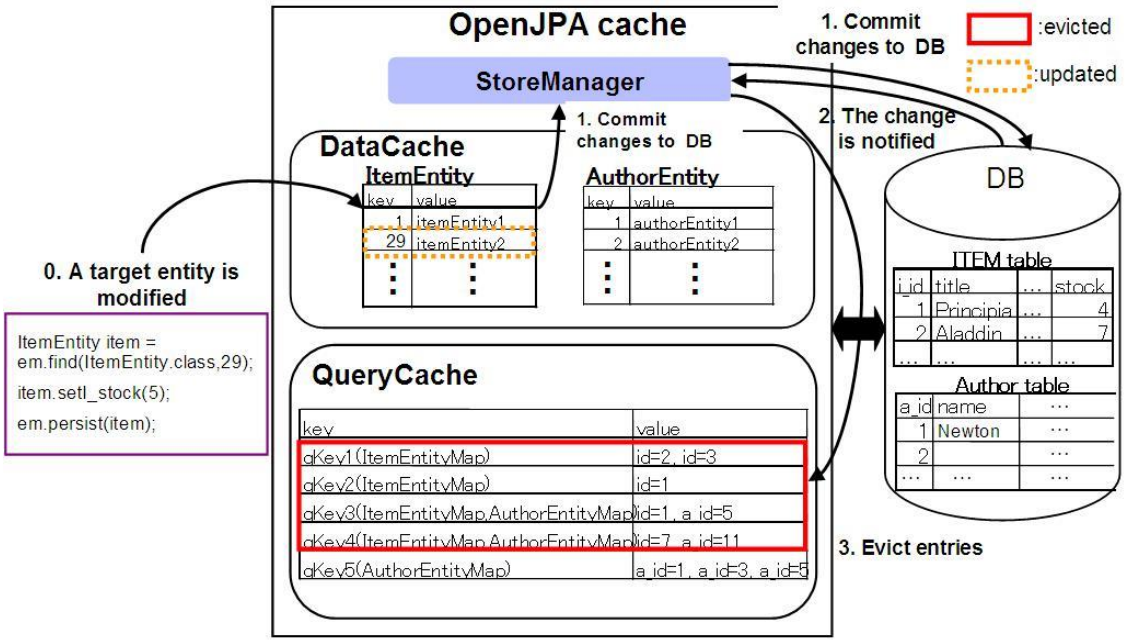


図 5.7　EntityManager のメソッドを使用したデータ更新時のキャッシュメンテナンスメカニズム

図 5.6 に示す通り，まず EntityManager から更新対象の Item エンティティを取得した後，そのフィールドのデータ（STOCK の値）を更新値に書換えることでキャッシュのデータも更新される(図 5.7 の 0)．そして，persist を実行することにより，データベースへ変更が反映される(図 5.7 の 1)．データベースのコミットの通知を受け取ると(図 5.7 の 2)，更新したエンティティをアクセスパスに含む QueryCache のデータをすべて削除(evict, invalidate)する(図 5.7 の 3).

<b> JPQL を使用して更新した場合

```
1.   public void updateItem() {
2.     EntityManager em = factory.createEntityManager();
3.     Query query = em.createQuery("UPDATE ItemEntity item SET
                                item.i_stock = ?1 WHERE item.i_id = ?2");
4.     em.getTransaction().begin();
5.     query.setParameter(1, 5);
6.     query.setParameter(2, 29);
7.     query.executeUpdate();
8.     em.getTransaction().commit();
9.   }
```

図 5.8　JPQL を使用した Item データ更新例

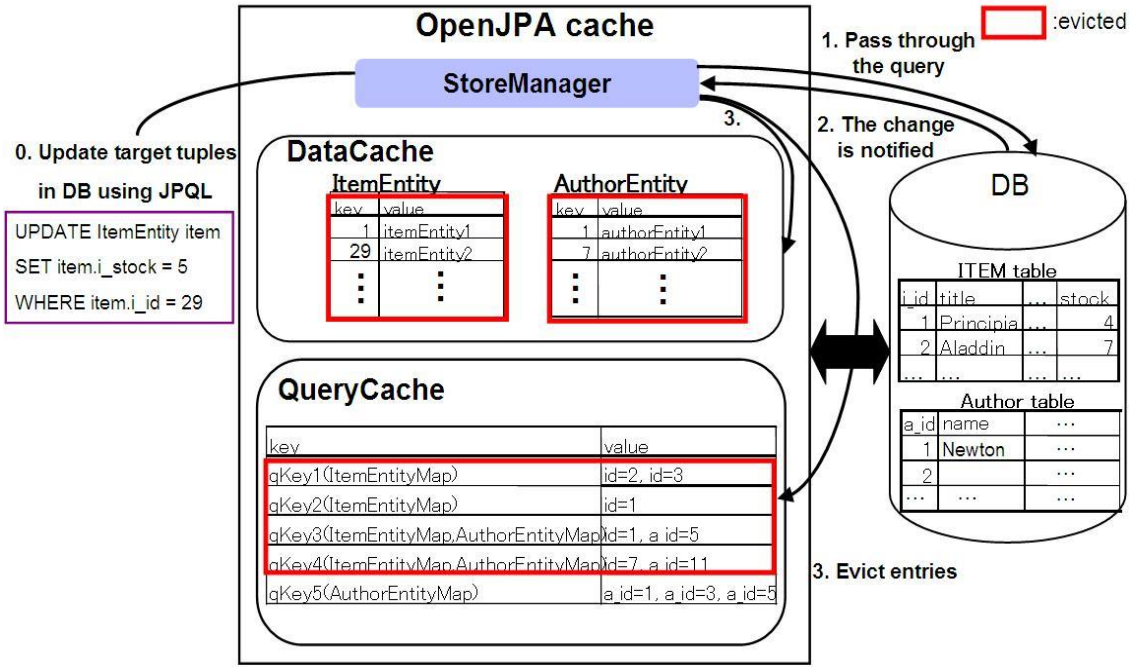


図 5.9　JPQL を使用したデータ更新時のキャッシュメンテナンスメカニズム

JPQL からのデータ更新時は, まずデータベースへその変更が反映される(図 5.9 の 0, 1). データベースでコミットされると(図 5.9 の 2), OpenJPA 側から, その更新したエンティティをアクセスパスに含む QueryCache のデータと, DataCache に格納されているエンティティのデータをすべて削除する(図 5.9 の 3).

したがって, 一度でも JPQL から更新がかかると, すべての DataCache のデータが削除されてしまうので, 大量のキャッシュを削除するコストと, DataCache のキャッシュヒット率の低下をまねくことになる. 一方, EntityManager のメソッドから更新した場合は, DataCache のデータは更新時に削除されることはないが, 更新データをキャッシュに反映させるコストがかかる. このようにテーブル単位の祖粒度な無効化が更新トランザクションがある場合にはキャッシュヒット率の低下を招いている. しかしながら, テーブル単位の無効化は, データ更新時に更新すべきキャッシュデータを細かく特定するコストを抑える. したがって, より細粒度にキャッシュをメンテナンスするかは, 更新クエリの性能とのトレードオフとなると考えれれる.

5.2 細粒度なキャッシュのメンテナンス手法の提案

本節にて, OpenJPA のキャッシュをより細粒度にメンテナンスする手法を提案する. データ更新時にオーバーヘッドが発生すると考えられるが, ユーザーは分析のためにデータを頻繁に参照する. 更新が数秒遅延したとしても, 主にはオフライン分析対象となるデータベースへのデータ挿入操作に遅れが出るのみである. したがって, 多少更新のオーバーヘッドが高くなったとしても, 細粒度なキャッシュメンテナンスにより, 参照クエリの性能を改善することを優先する.

EJB2.1 以前の性能評価がこれまでに報告されてきた. Paul ら[Pa01]は, EJB1.1 で提供されていた, 3 種類のコミットパターンに対して, オブジェクトプールとキャッシュのサイズを変更した場合の性能比較を行っている. Emmanuel ら[Em02]は, Servlet アプリケーションと, EJB2.0 の複数のタイプ(stateless/stateful session beans)を

使用するように変更したアプリケーションとの性能比較を行っている．Avraham ら[Avr03] は，JDBC アプリケーション，EJB アプリケーション，EJB のキャッシュを有効にした時のアプリケーションにおいて，アプリケーションサーバ台数を増加させた時のスケールアウトの性能を測定し，キャッシュがアプリケーションサーバのスケールアウトに効果的であることを示している．

EJB3.0 を対象とした研究としては，Ben[Ben08]らが，EJB からの参照トランザクション実行時のデータのロードを最小限にするために Java コンパイラを拡張して静的解析・変換を行い，オブジェクトロード数の削減を行った．また，JPQL のクエリ部分が Java コンパイラの解析対象となっていないために，クエリをタイプチェックを可能にした Zachary[Zac08]らの研究がある．EJB3.0 に用いられるキャッシュの効果や性能比較に関連する研究はこれまでになかった．

本研究では，EJB3.0 実装の OpenJPA のキャッシュのメンテナンス手法を細粒度に行う手法を提案し，キャッシュの効果がより大きくなることを目指す．メンテナンスの機能は OpenJPA の実装の中の，キャッシュアクセス部分に本手法の機能を追加して実現する．OpenJPA は Java で実装されており，キャッシュデータへは StoreManager クラスを介してアクセスしている．この StoreManager とキャッシュのデータとの間に，本研究で提案するメンテナンスの機構(MyCacheManager)をいれる(図 5.10).

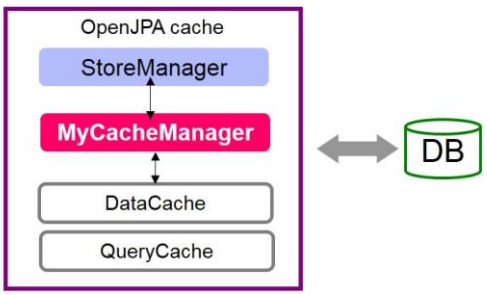


図 5.10　カスタマイズしたキャッシュメンテナンス機構の導入

データベースのクエリをキャッシュする技術に関しては，例えばDBCache[Luo02]は，頻繁にアクセスされるデータのみをキャッシュしている．DBProxy [Ami03]は materialized view の形でデータをキャッシュしている．MTCache [Lar04]は分散環境でレプリケーションされた設定でのクエリ実行時のデータをキャッシュしておくために，中間層にデータキャッシュを置くソリューションを提案した．Ferdinand[Cha08]はデータベースのクエリをキャッシュして，オフラインでそれらを分析し，更新の影響が及ぶ範囲をいくつかのグループ分けにする．Gupta [Gupta93]らと Kenneth[Ken96],らは materialized view をインクリメンタルに更新する手法を提案している. これらのキャッシュ形式はOpenJPAのQueryCacheとDataCacheという 2 種類からなるキャッシュ構成と形が異なる．OpenJPA は全てのクエリとデータそのものの結果をそれぞれにキャッシュするため，これらとは異なるキャッシュメンテナンスのメカニズムが必要となる.

また，これらの手法は，データベースがほとんど更新されないことを前提としている．データベースとの整合性を保つため，データベースのデータが更新された場合は，キャッシュしたデータも更新しなくてはならず，データベースとキャッシュのデータ双方の頻繁な更新処理は，逆にオーバーヘッドをまねく懸念がある．本研究で提案しているシステムは，リアルタイムに受信したソーシャルデータを，逐次的に処理した後にデータベースに挿入するため，定期的に更新が入ることが想定される．そのため，更新の影響を受けないキャッシュのデータを出来るだけ多く維持するための新しい手法を提案する.

5.2.1 データベースクエリの依存性解析を用いた QueryCache のメンテナンス手法

祖粒度なキャッシュの無効化をさけるために，本節では QueryCache の無効化をより細粒度で実施する手法を提案する．

キャッシュの無効化の対象をより細粒度に適切に決定するために, 更新クエリと照会クエリ間の依存関係を解析する手法がある[Levy93] [Gar08]．これは JDBC を直接用いたアプリケーションの SQL の更新クエリと照会クエリ間の依存関係をクエリテンプレート(Prepared statement)からあらかじめ解析しておくことにより，ある更新が発生したときに，実際に影響のある照会クエリのキャッシュエントリのみが削除されるようになる. この手法を, OpenJPA のアプリケーションに応用することで, キャッシュの無効化の粒度を細粒化する．

5.2.1.1 列情報を利用したクエリの依存性解析

ある更新クエリが実行されたときに，それがキャッシュ中のどの照会クエリの結果に影響を与えるかということは，更新クエリが更新する列と照会クエリが参照する列を比較することによって決定することができる．図 5.11 に示された例では stock 列を更新するクエリ U1 の実行は, stock 列を参照している Q1 にのみ影響することがわかる．さらにその解析結果を利用して，キャッシュの無効化対象を Q1 cache に絞ることができる．

このようにアプリケーションが使用するクエリの情報があらかじめ得られれば，この解析をオフラインで行うことにより，それぞれの更新クエリの実行時に無効化対象となる照会クエリを特定することができる．

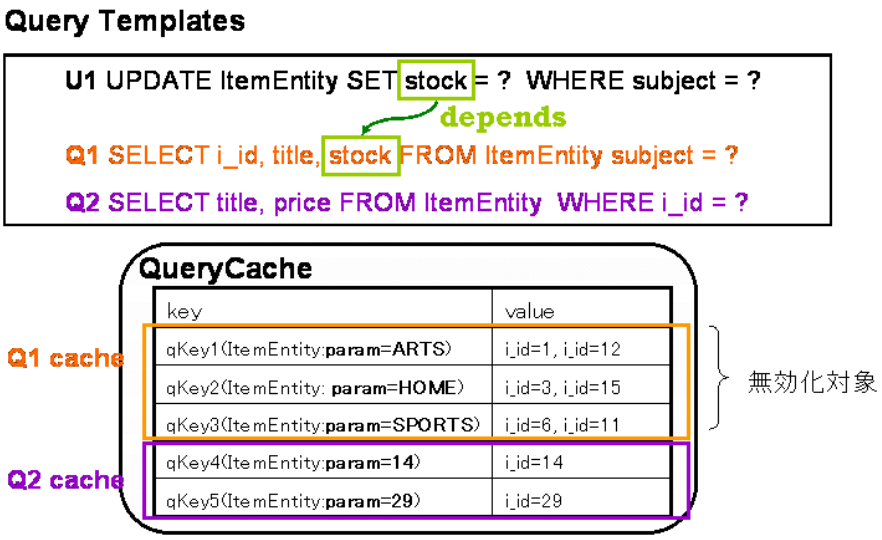


図 5.11　列情報を利用したクエリの依存性解析

5.2.1.2 値を利用したクエリの依存性解析

クエリテンプレートは多くの場合，実行時に定まるパラメータを持っており，その情報をもとに依存性解析を行うことで，さらに細粒度に無効化対象を決めることができる．図 5.12 に示された例では，クエリ U1 は subject の値が”SPO リツイート S”である行の stock を更新する．したがってその実行は，stock 列を参照している Q1 の検索結果のうち，subject の値が”SPO リツイート S”のもののみに影響する．

ただし，クエリのパラメータは実行時に定まるため，列情報を利用したクエリの依存性解析とは異なり，事前にオフラインで解析を行うことはできず，オンライン時に上記の解析を行う必要がある．

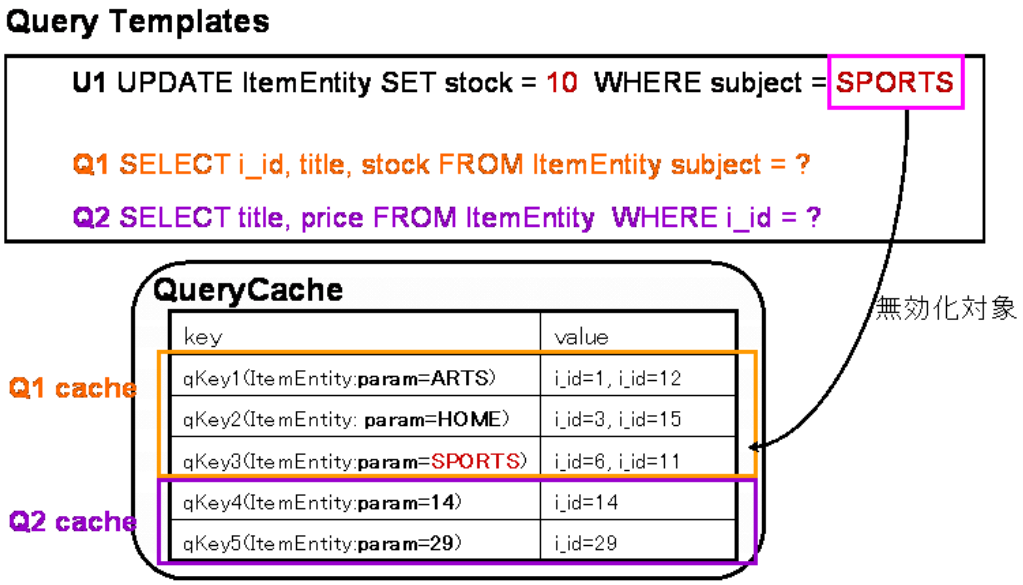


図 5.12　値情報を利用したクエリの依存性解析

5.2.1.3 列情報を利用した時と値情報を利用した時のクエリの依存性解析の実施手順

図 5.13 に列情報を利用した時と値情報を利用した時のクエリの依存性解析の実施手順を示す．更新クエリが発行された時に，QueryCache の依存性がチェックされる．値情報を利用した依存性解析を実施する場合は，追加でクエリのパラメータを解析する. クエリの依存性解析は, クエリ更新時の依存解析手法を基にしている[Levy93].

5.1.3 で紹介したように，データの更新手法はエンティティを介した更新と，JPQL による更新がある．従って，本手法をどちらの更新が来ても適用できるようにする (図 5.14)．まず更新のクエリを受信し，列レベルもしくは値レベルでの依存性解析を実施するために必要な情報を取得する．次にその情報を用いて依存性解析を実施し，無効化対象のデータを出力する．

```
1.      invalidate_QueryCache(updateQuery)
2.      //iterate until all query pattern in QueryCache are checked
3.       while(selectQueries.hasNext()){
4.         selectQuery = selectQueries.getNext();
5.         //check dependency by the column-level analysis
6.         isdepend = doColumnBasedDependencyAnalysis (updateQuery, selectQuery);
7.           if(isdepend){
8.           //add keys to list for invalidation
9.            invalidationList.add(getQueryKeys(selectQuery));
10.          }
11.      }
12.     //invalidate only cache entries related in update Query
13.      if(column_based_invalidation){
14.        invalidate(invalidationList);
15.      }else if(value_based_invalidation){
16.        //vlue-level analysis
17.        while(invalidationList.hasNext()){
18.          selectQuery = invalidationList.getNext();
19.           //check dependency by the value-level analysis
20.          isdepend = doValueBasedDependencyAnalysis(updateQuery,uValues,selectQuery,sValues);
21.          if(isdepend){
22.               valueBasedInvalidationList.add(getQueryKeys(selectQuery));
23.          }
24.         invalidate(valueBasedInvalidationList);
25.      }
26.     }
```

図 5.13　列情報と値情報を利用したキャッシュ無効化手順

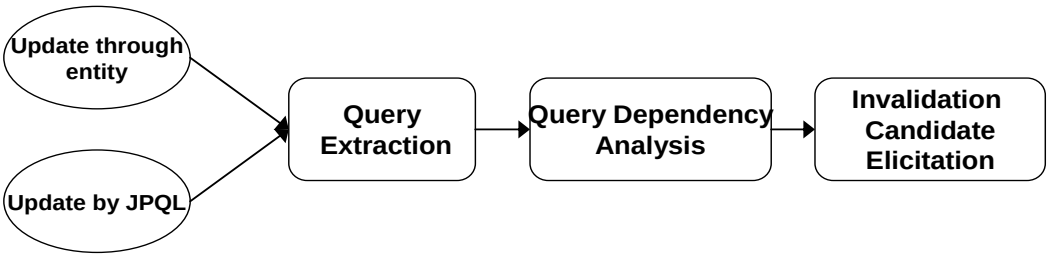


図 5.14　OpenJPA でのクエリ依存性解析

5.2.1.4 クエリの依存性を解析してキャッシュを無効化するコストモデル

理想的には，更新クエリが発行された時に，その影響が及ぶキャッシュデータを厳密に特定することで高いキャッシュヒット率を維持できると期待される．しかしながら，更新の影響範囲を細粒度せ特定するにはそれなりのコストがかかることとなるため，細粒度の手法が必ずしも性能向上に最適であるとは限らない．本節では本手法を用いたキャッシュ無効化のコストを考える．

本手法によるオンライン時に発生するコストは大きく 2 つからなる．1 つはキャッシュを無効化するコスト *Cr* であり，もう 1 つはクエリの依存性を解析して無効化すべきデータを特定するするコスト *Ca* である．I_p, I_q と I_r はそれぞれテーブルレベル, 列レベル, 値レベルでの無効化すべきキャッシュエントリの数を表している．*R* は 1 つのキャッシュエントリを削除するコストを表す. 値レベルのクエリの依存性解析のコストを Qv とする．

$$Table-based: C_r = I_p \times R,\ C_a = 0$$

$$Column-based: C_r = I_q \times R,\ C_a = 0$$

$$Value-based: C_r = I_r \times R, C_a = I_q \times Q_V$$

クエリテンプレートを事前にアプリケーションから取得できる場合, 列レベルのクエリの依存性解析はオフラインで実施できるため，テーブルレベルと列レベルの Ca は 0 になる．また，もし取得できなかった場合も，アプリケーション稼動の初期段階でクエリのテンプレートを取得して依存性解析を 1 度実施すれば良い．対して，値レベルの依存性解析は，クエリのパラメタを毎回チェックしなければならないため，クエリが発行されるたびにオンラインで実施する必要がある．従って，どのレベルでの依存性解析を実施してキャッシュをメンテナンスするのが最適であるかはアプリケーションによって異なる可能性があり，容易に決定できるものではない．

5.2.2 メモリ効率の良い無効化用インデックスの導入

5.2.2.1 無効化用インデックス

5.1.3 節で述べたように，JPQL からの更新クエリにより，データベースのデータが更新されると，データベースとキャッシュのデータ一貫性を維持するために，更新されたテーブルの DataCache のデータを全て無効化してしまう．これは，OpenJPA の DataCache では ITEM テーブルの主キー以外はオブジェクトとしてエンティティに格納されているため, 例えば author 名が “Brown”に該当するキャッシュエントリを一意に特定するには，DataCache 内の ITEM Entity を全て iterate して探す必要がある．しかし DataCache 内のエントリを更新クエリの度に iterate することは逆にオーバーヘッドになることも考えられる．現状，OpenJPA は iterate してキャッシュエントリを特定することはせず，更新されたエンティティ（テーブル）に属するキャッシュエントリをすべて無効化する．

このような DataCache の粗粒度なキャッシュ無効化を防ぐために，無効化用のインデックスを OpenJPA のキャッシュレイヤに対し導入する．従来インデックスは参照クエリの高速化のために用いられるが，無効化用にインデックスを使うことにより，無効化するべきキャッシュエントリを効率良く特定することが可能となる．図 5.15 はインデックスを導入したキャッシュの無効化について表している．

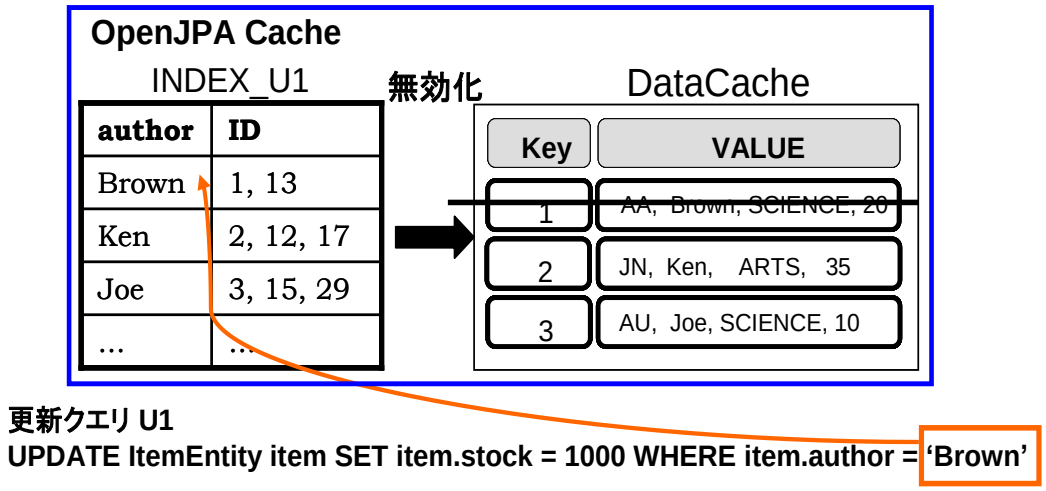


図 5.15　無効化用インデックスを用いたデータ更新時のキャッシュのメンテナンス

更新クエリ U1 が発行されると，クエリを解析して，WHERE 句で指定されているパラメータ値を取得する．その値を以て INDEX_U1 を参照し，無効化すべき DataCache のエントリの ID リストを取得して，該当のエントリを無効化する．これにより，author 名が”Brown”のエントリだけを無効化し，それ以外のキャッシュエントリは DataCache に保持しておくことが可能となる．もし，データベース側へ直接トリガを利用した更新等が入る場合，現状の OpenJPA ではそのトリガを検知することはできないため，データベースとキャッシュのデータに不整合が生じる可能性があるが，OpenJPA のインタフェース外からデータベースを直接操作されることは

OpenJPA の仕様外である．しかしながら，今後 OpenJPA でトリガがサポートされた場合は，トリガ発生を検知してキャッシュメンテナンス可能であると考える．

このように, アプリケーションの更新クエリパターン毎に無効化用インデックスを生成することにより, キャッシュヒット率の向上が見込まれる．しかし一方で, キャッシュ領域のメモリサイズは限られているため，無効化用インデックスに多くの領域を割当てると，DataCache の領域が圧迫される可能性があり問題となる．

データベースの，限られたサイズ領域の中で作るインデックスに関して，Stonebraker[Stone98]らは，限られたデータ空間を有効に使うために，partial index を作成している．このインデックスは，クエリのパターンを参照して，データベースのテーブルのデータの一部を格納するものである．Wu et al. [Wu11]らは，インデックスを水平分割して分散格納する手法を提案している．これらの partial index の手法は，照会クエリ用に生成されたものであり，トータルのインデックスの一部を切り出して作成したものである．無効化用インデックスにこの手法を適用した場合，DataCache に格納されているデータの一部について無効化用のインデックスがはられるということを意味する．その場合，無効化用インデックスに入らなかった DataCache のデータは, 更新クエリが発行された時に無効化対象になるかの判断が出来なくなるため，結局 DataCache から消さなくてはならない．その場合は従来の粗粒度なメンテナンスとなるため結果的にキャッシュヒット率が低下してしまう．

そこで次節では, 与えられたキャッシュ領域内で無効化用インデックスが効果を維持できるよう，キャッシュ領域の容量にあわせてサイズ変更可能な無効化用ハッシュインデックスを提案する．さらにキャッシュヒット率を向上するために，アプリケーションのデータアクセスの特性にあわせてインデックスの構成を最適化した無効化用重み付ハッシュインデックスを提案する．

5.2.2.2 無効化用ハッシュインデックス

照会クエリ用のインデックスと異なり，無効化用のインデックスは，インデックスから無効化すべき所望の ID リストが一意に特定されずとも，その ID リストを含んだ ID のリストを用いて無効化すれば，データベースとのデータの整合性を損ねずに無効化可能である．この性質を利用して，あらかじめ指定した無効化用インデックスサイズ k の中に，インデックスがおさまるように，適当なハッシュ関数を適用して DataCache 内の全エントリの ID 情報を入れる．

サイズ k のインデックスを生成するためには，ある更新クエリの WHERE 句で指定されたパラメータの型値を整数化し，k でその値の mod をとることによって，そのパラメータ値で更新される ID リストを，インデックス k 個分割領域のいずれかに格納すれば良い．パラメータが複数指定される場合は，指定された値全てを用いて整数化する．

例えば図 5.16 にて，更新クエリ U1 にて”Brown”が指定された場合，無効化用インデックスでは ID=1，13 のデータを DataCache から無効化し，INDEX_U1 からもその ID リストを削除する．対して，無効化用ハッシュインデックスでは，”Brown”というパラメータからハッシュ関数を適用して，”1”という値に変換されると，Hash_INDEX_U1 のキー1 に属している ID リストを参照し，それらの ID を持つデータ全て(ID=1,13,20,24 のデータ)を DataCache から無効化する．Hash_INDEX_U1 からもその ID リストを削除する．

ハッシュインデックスに ID が登録されるタイミングは，データベースから取得したデータが DataCache に格納される時である．取得したデータからハッシュ関数の計算に必要なパラメータ値を参照してハッシュ値を計算し，ハッシュインデックス中の当該箇所の ID リストに ID 値を追加する．すなわち，DataCache にデータが

存在する場合は必ずハッシュインデックスの ID リストにも該当する ID が格納されている.

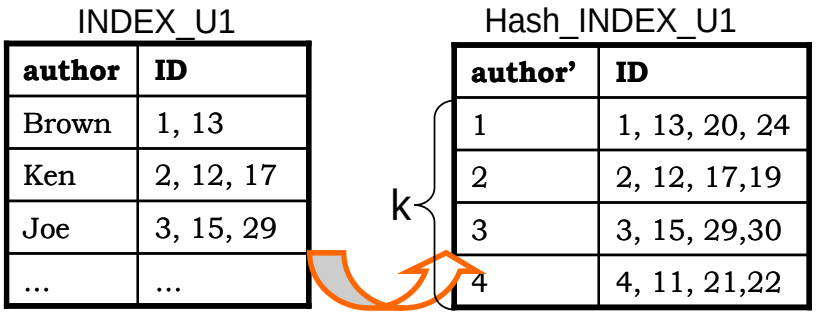


図 5.16　無効化用インデックスからハッシュインデックスへの変換

5.2.2.3 無効化用重み付ハッシュインデックス

無効化用ハッシュインデックスにより，限られた空間内で ID リストを保持できるようになった．ところがハッシュで分割されるために ID リストはインデックス内に均一に分配される．この場合，もし頻繁に参照される ID_1 と，頻繁に更新される ID_2 が同じハッシュの領域に入った場合, ID_1 のデータは頻繁に参照されるものの頻繁に無効化されてしまい，キャッシュの恩恵が受けられにくくなってしまう.

現実のアプリケーションの多くは全てのデータに対して均等にアクセスするのではなく，一部のごく限られたデータに対してアクセスを行うという特性がある．例えば，インターネットオークションでは，人気商品やオークション時間期限間近の商品が頻繁にアクセスされる傾向がある．ソーシャルメディアデータでは，特定のトピックや流行のキーワードのみを対象に頻繁にアクセスされる等が考えられる.

したがって，もしアプリケーションがデータの照会・更新の頻度や分布に偏りがある場合，それらの特性を考慮して無効化用ハッシュインデックスのデータ分配

を最適化した無効化用重み付ハッシュインデックスを提案する．これにより高いヒット率とスループットを維持できることを目指す.

この重み付ハッシュインデックスでは, ハッシュインデックスの各分割領域に対する更新頻度と照会頻度の情報をもとに，ハッシュインデックスの保持の仕方を変更する．具体的には，ハッシュインデックスの各分割領域に対し，頻繁に更新されるデータを持つ分割領域が複数存在した場合，それらはキャッシュの効果を得にくい領域であると判断して，小さな領域にまとめる．これにより空いた領域を利用して，照会の割合が高い分割領域をさらに広い領域を割り当てることにより他のデータが更新されたときの無効化の影響を受け難くする.

以下に重み付ハッシュインデックスの作成手順を述べる．まず，各分割領域の更新と照会頻度から重み値を用いて各分割領域の重要度を計算する．次に，その値により領域を再配置する.

(1) 重み値の計算

ある分割領域 i の重み値は以下のように表される．r_i は照会トランザクションの数で，w_i は更新トランザクションの数である.

$$Weight_i = F(r_i, w_i)$$

$F(r_i, w_i)$ のシンプルな例としては，r_i / w_i となる．その場合，重み値は照会トランザクションの数が多いほど大きくなり，更新トランザクションが多いほど小さくなる.

(2) 重み値をもとにしたハッシュインデックスの再構成

各分割領域の重み値がだいたい等しければ, ハッシュインデックスはそのデータアクセスパターンに対して最適な構造になっていると言える．そこで我々は，ま

ず分割領域数が *k* のハッシュインデックスの領域を *2k* に拡張する．図 5.17 に例を示す．サイズ *2k* のインデックスを生成するためには，ある更新クエリの WHERE 句で指定されたパラメータの型値を整数化し, *2k* でその値の mod をとるように変更する．*weight'*の値には *weight* の値をコピーする

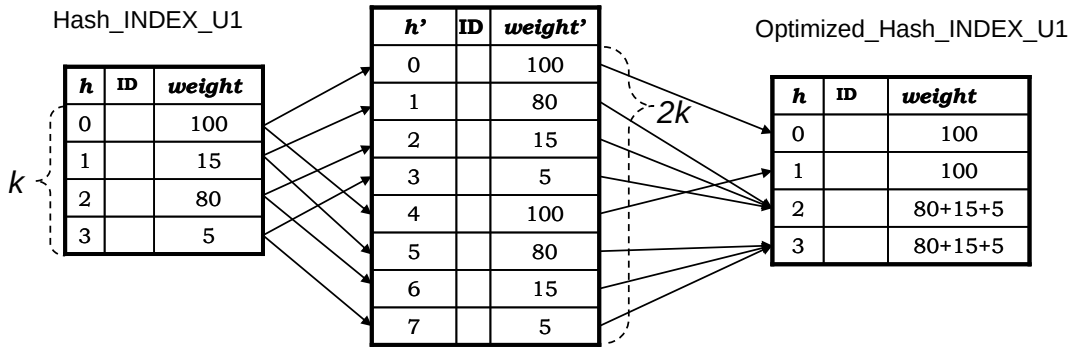


図 5.17　ハッシュインデックスから重みつきハッシュインデックスへの変換

重み値のリスト{*weight'*$_0$, *weight'*$_1$, . . . , *weight'*$_{k2-1}$}　をもとに，{*h'*$_0$, *h'*$_1$, *h'*$_2$, .., *h'*$_{2k-1}$}の領域を，再び k 個の領域数のインデックス{h_0, h_1, h_2, .., h_k}に以下の式を最小化するように配置する．

$$\max_i\{\sum_{h'_j \in h_i} weight'_j\} - \min_i\{\sum_{h'_j \in h_i} weight'_j\}$$

しかしながらこれは，multi-way partitioning problem [Mic79] であり，NP 完全であることが知られている. partitioning problem とは, 与えられた n 個の整数 $a_1,...,a_n$ を二つの集合に分け，各々の集合内の数の和がもう一方の集合内の数の和と等しくなるようにできるかどうかを判定する問題である．これに対して，ヒューリスティ

ックな手法が存在するため[Kar82] [Kor98]，これをハッシュインデックスの再構成に用いる．

図 5.16 の右のインデックスはこの手法を適用した後のインデックスである．まず，*weight'* の値を降順に並べ{100, 100, 80, 80, 15, 15, 5, 5}，る．*k*=4 とする．h_0 = (100), h_1 = (100), h_2 = (80), and h_3 = (80) をセットする．次の *weight'* の値である 15 は，最も値の小さい h_2 にセットされて，h_2 = (80, 15) になる．このように，次の重み値は，各領域の重み値の合計が最も小さい場所に配置されていく．最終的に，h_0 = (100), h_1 = (100), h_2 = (80,15,5), and h_3 = (80,15,5)　となる．*h* から *h'* への再配置するための紐付けは，別途格納して記憶しておく．

この最適化されたハッシュインデックスを用いたキャッシュ無効化の手順を以下に示す．

```
1.  //The hash key calculated from the parameters of the WHERE clause
2. hash_key = param_value % 2k;
3.  //Return the key of optimized hash index
4. index_key = getReference(hash_key);
5. invalidation_id_list = optimized_index.get(hash_key);
6.  //invalidate cache entries in the DataCache
7. datacache.invalidate(invalidation_id_list);
```

重み付ハッシュインデックスの変換は，ハッシュインデックスを生成してアプリケーションを稼動してから，ある一定時間経過した後のタイミングで行われる．ハッシュインデックスの各領域のデータの照会・更新数をプロファイルするモニタリングプロセスが別途存在し，そこから計算した重み値をもとに，変換手続きを実施する．また，変換後はアプリケーションのアクセスの偏りが変化していないか，DataCache のキャッシュヒット率を監視し，ハッシュインデックス使用時よりもヒット率が下回った場合，重み付インデックスを再生成する指令を出す．この状況はデータパターンが変化したということであるため，前回の重み付ハッシュインデック

ス生成のために用いた照会・更新数と重み値も同時にリセットする．再びハッシュインデックスの状態に戻した後は，各領域のアクセス頻度を取得して重み付インデックスを再生成することを繰り返す．データアクセスの偏りが頻繁に変化するアプリケーションでは，インデックス再編成のコストがオーバーヘッドになってしまう可能性があるが，例えば，インターネットショッピングに於いて頻繁に照会や更新される，新商品やベストセラーの商品群は，数分単位で激しく変化するものではないため，日単位でインデックスを再生成するなどしても効果を維持できると考えられる．

5.2.2.4 無効化用インデックスのサイズの推定

無効化用インデックスを導入することにより，細粒度のキャッシュメンテナンスが期待されるが，これにより本来 DataCache と QueryCache が利用するメモリサイズを使用することになる．そこで，無効化用インデックスにどれ程のサイズが割り当て可能かを計算する．無効化用インデックスに使用できるサイズを $S_{invalidate}$ とすると，以下のように表される．

$$S_{invalidate} = S_{total} - S_{DataCache}$$

S_{total} は，キャッシュに使用可能な総メモリサイズである．DataCache に使用される領域($S_{DataCache}$)は以下の式によりもとめられる．

$$S_{DataCace} = \sum_{i} (T_i \times n_i)$$

(T_i x n_i) は，テーブル i の DetaCache のサイズである．T_i はテーブル i の 1 レコードあたりのデータサイズである．n_i は DataCache に格納されたレコード数であり，最大値はテーブル i の総レコード数となる．基本的に，アプリケーションからアクセスされたテーブルの全てのデータはキャッシュされることになる．

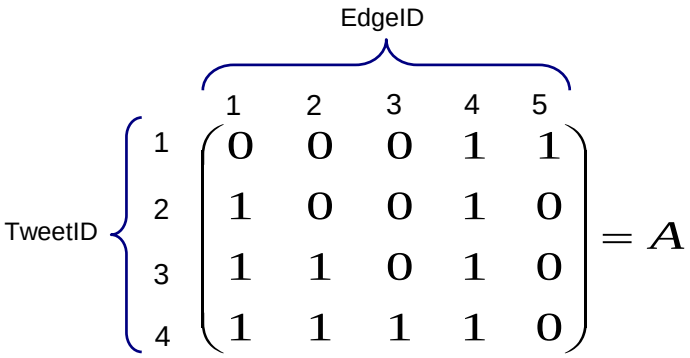
無効化用インデックスは，アプリケーションの更新クエリのパターンごとに生成されるため，$S_{invalidate}$ のサイズをそのパターン数のインデックスでシェアすることとなる．

5.3 グラフデータのエッジ情報のインデックスの導入

ここまでは，データベースへのクエリの結果のデータをキャッシュする機構とそのメンテナンス最適化手法について述べた．ここで，構造化されたテーブルに格納されているデータのほかに，3 章で述べたような代表的な問合せパターンの中の，拡散ネットワークのようなグラフ構造データを取得する場面も多く存在する．このようなデータもテーブルに格納されてはいるが，グラフデータを高速に取り出すためには，ビットマップインデックスを使うことができる．ビットマップのインデックスは，OpenJPA のキャッシュで使用されているようなハッシュマップ構造よりもサイズが小さく，ビットマップ間の論理積や論理和を高速に演算できるため，たとえば複数のツイートの拡散ネットワークを取得する場合に効果的である．そこで，エッジ情報のインデックスをキャッシュ機構に追加することを検討する．

5.3.1 エッジインデックス

エッジ情報を含むようなインデックスを生成すると，以下のような行列で表現される．



行は TweetID, 列は EdgeID であり，各 TweetID のツイートに対して，各 Edge がリツイートしていれば 1, していなければ 0 が入る．すなわち，例えば TweetID=’1’のツイートは，EdgeID=’4’と EdgeID=’5’ がリツイートしたエッジとして存在するということである．各エッジは 3 章で示した RETWEET テーブルの ”Src”と”Dst”のペアに該当する．EdgeID がどのペアで構成されているかは別途格納しておく必要がある．

複数の TweetID が指定されて, 各エッジのリツイートの頻度情報も必要な場合は，該当する TweetID の行データを，列方向に足し合わせることで，各エッジのリツイート頻度情報を得ることができる． この行列 A を，アプリケーションサーバー側の Data access layer に保持しておく．分析モジュールからの問合せに対し，Data access layer にてこのインデックスへの問合せ操作が入ることになるが，インメモリの操作なので高速に処理できると考えられる．

5.3.2 インデックスの圧縮

インデックスとして導入したこの行列は，アプリケーションサーバーのメモリ内に配置されるため，インデックスサイズは可能な限り小さくメモリ消費量をおさえることが望ましい．非常に密な行列の場合はこのような行列表現が有効であるが，ほぼすべての Tweet に対してリツイートするユーザーは，ごく一部存在する可能性がある程度であり，他ユーザーは数回リツイートする頻度であることを想定すると，大規模な疎行列になる可能性が高い．

また，この行列は，行数に比べて，列数が大きくなることが予想される．行に入る TweetID ごとに，数百から数千，万規模のリツイート数とすれば，そのリツイートしたエッジのパターン分の列数となる．そこで，インデックスに圧縮手法を適用して，格納サイズを圧縮する．

5.3.2.1 疎行列の圧縮

行列圧縮手法にはいままで様々な圧縮手法が提案されてきている．その中でも疎行列の圧縮として代表的なのは，圧縮行格納方式(Compressed Row Storage, CRS) [Com]である．CRS は，行成分に圧縮して，非ゼロの要素の位置のみを格納している．前節の行列 A を例にとると，まず非ゼロの要素を左上から右へ並べた配列を生成する．

A = [1 1 1 1 1 1 1 1 1 1 1]

さらに，各要素がある行と列の位置を並べた配列を生成する．

IA = [1 1 2 2 3 3 3 4 4 4 4]　i 行

JA = [4 5 1 4 1 2 4 1 2 3 4]　j 列

IA は各数字が初めに出現する位置だけを覚えるようにして，

IA’ = [1 3 5 8]

とさらに小さい配列で表現できる．CRS ではこの A,IA’,JA の組を用いて行列を表現するが，本論文でのエッジインデックスの行列に入る値は 0 か 1 なので，A は常に 1 が並ぶだけになるため，保持する必要はない．

5.3.2.2 連長圧縮

行列 A に入る値は 0 か 1 であり，リツイート数が多いツイートほど 1 が多く入ることになる．列には Edge が並んでいるが，ちょうどリツイートしたユーザーの Edge が連続して並んでいる場合は，1 が連続する．逆にリツイートしなかったユーザーが並べば 0 が連続する．したがって，同じ値が連続して出現するほど圧縮効果がある連長圧縮(ランレングス圧縮　Run Length Encoding , RLE)[Com2]もエッジインデックスの圧縮に効果があるのではないかと考えられる．連長圧縮は，連続したデータをそのデータ一つ分と，それが連続した長さで表現して圧縮を実現している．

行列 A を例にとると,1 行目は 0 が 3 個続いた後に 1 が 2 個続いているので[03 12]，2 行目は同様に[11 02 11 01]と表現される．このインデックスの値は前述のように 0 か 1 なので，各行の要素の数は必ず 0 の出現個数から数えるようにする，という決まりにしておけば, 要素数だけを保持しておくだけで復元できる. したがって, 行列 A の 1 行目は[3 2]，2 行目は[0 1 2 1 1]となる.

ここで，2 行目の圧縮後の要素数をみると 5 であり，これは圧縮前の要素数と変化がない．このように連長圧縮では，異なる値が交互に出現すると圧縮効果が小さくなってしまう．したがって，連長圧縮の効果を得るためには，できるだけ同じ値が連続して出現していることが望ましい.

5.3.2.3 インデックスの列の並べ替え

連長圧縮の効果をあげるために，連長圧縮前の前処理として，インデックスの列を並べ替える.

頻出アイテム集合

Edge の並びをかえて，できるだけ 1 が連続するようにすることで，行方向に連長圧縮を実施したときに，連長圧縮の効果が大きくなる行が存在することになる．そこで，各 TweetID の EdgeID が 1 のリストをアイテム集合とみなして，頻出する Edge の集合を発見し，その Edge 集合をまとめて並べることで，その Edge 集合をもつ TweetID の行は, 少なくとも発見した EdgeID が並ぶ部分は 1 が連続することになる. Edge 集合の頻度(=support 値)と，集合のサイズ(Edge の数)が大きいほど，効果が高いといえる．このような Edge 集合を重複なく複数発見し，その集合ごとに Edge を並べて列の並びを決定する.

例えば以下の行列では，1, 2 行目に該当する TweetID には共通の Edge が 3 つ存在していて，3,4 行目には共通の Edge が 2 つ存在していたことを発見し，Edge の順を並べ替えた後の行列を表している．

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

頻出アイテム集合の発見は幅優先探索型アルゴリズムの代表的なアプリオリアルゴリズム[Ag94]や深さ優先探索型アルゴリズムの代表的な Eclat[Mo97]などが存在するが，頻出アイテム集合を，数千〜万のエッジ集合に対して実施するには，計算コストが高く，計算時間が長くなるというデメリットがある．

リツイート頻度順

各 Edge を，列方向に和を計算して，降順(= リツイートした回数が多い順)に Edge を並べることで，1 の出現が多い Edge 同士が並ぶことになり，行方向に見ても 1 が連続しやすくなる可能性がある．頻出アイテム集合発見のように，1 が連続することを保証できるものではないが，計算時間としてはそれぞれの Edge の和を計算してソートするだけなので，頻出アイテム発見の計算よりは計算コストを大幅に削減できる．

以上，本節ではグラフデータのデータアクセスを高速化するため，エッジインデックスを導入した．また，サイズ削減のために効果が期待される行列圧縮手法について紹介した．圧縮することにより，圧縮と復元のオーバーヘッドが発生することが考えられるが，それらの影響については次章の実験にて検証する．

5.4 まとめ

本章では，蓄積されたデータを処理する機構において，蓄積された大規模データを高速に問合せ処理するためのキャッシュの導入と，データ更新の影響を出来るだけ抑えるためのキャッシュメンテナンス手法を提案した．我々のシステムは Java で構築されているため，POJO ベースの O/R マッピングアーキテクチャであり，JDBC を直接使用したアプリケーションよりも少ないプログラムコードでデータにアクセスすることができる JPA(Java Persistence API)をデータベースアクセス部分に用いる．

JPA 実装の一つである Apache OpenJPA は，キャッシュレイヤーを提供しており，頻繁にアクセスされるデータをキャッシュすることにより，データアクセスを高速に行えることが期待される．キャッシュされたデータはデータベースのデータが更新されると一貫性を保持するために，更新するか無効化しなくてはならない．OpenJPA の提供するキャッシュメンテナンス手法は，テーブル単位でデータを無効化するため，メンテナンスコストは小さいがキャッシュされるデータ数も少なくなってしまうことがある．本研究で提案しているシステムは，ストリーム処理されたデータが定期的に挿入されてくるため，このような粗粒度なメンテナンスでは，ほとんんどキャッシュにデータが残らずにキャッシュの恩恵をうけることができない．

そこで，OpenJPA のキャッシュをより細粒度にメンテナンスする手法を提案した．データ更新時に多少のメンテナンスオーバーヘッドが発生すると考えられるが，分析ユーザーが発行するのは主に参照クエリである．更新が数秒遅延したとしても，主にはオフライン分析対象となるデータベースへのデータ挿入時への影響である．

したがって，多少更新のオーバーヘッドが高くなったとしても，細粒度なキャッシュメンテナンスにより，参照クエリの性能を改善することを優先する．

そこで本章では，OpenJPA の提供する QueryCache をクエリの依存性解析結果を用いて列単位，値単位で更新の影響が及ぶキャッシュのみを無効化する，より細粒度のキャッシュメンテナンス手法を提案した．もう一つのキャッシュである DataCache には，無効化すべきデータを特定するためのインデックスを導入した．さらに，与えられたメモリ領域内で無効化用インデックスが効果を維持できるよう，メモリ領域の容量にあわせてサイズ変更可能な無効化用ハッシュインデックスと，さらにアプリケーションのデータアクセスの特性にあわせてインデックスの構成を最適化した無効化用重み付ハッシュインデックスを提案した．

また，ソーシャルメディアの情報拡散ネットワークのようなグラフデータには，ビットマップのインデックスを使用することが効果的である．このインデックスは大規模な粗行列になることが想定されるため，行列圧縮の手法を適用してより小さいサイズでインデックスが生成されるようにした．ソーシャルメディアデータの，情報が拡散されやすい経路が存在する特性を考慮して，圧縮率をさらに高めるための行列並び替えを提案した．

第6章　データベースアクセスの最適化の検証

本章では，まずは OpenJPA の基本性能を評価する．具体的には，OpenJPA を利用した Java の Web アプリケーションと従来の JDBC 直接使用時との性能比較による OpenJPA のオーバーヘッドの検証，さらに OpenJPA のキャッシュ機能を有効にした時の効果測定をベンチマーク(TPC-W)を用いて行う．また，更新トランザクション時の JPA からの更新方法を変更することによる OpenJPA のキャッシュメンテナンス方法の違いとそれに伴うパフォーマンスへの影響についても比較し考察する．

続いて，キャッシュを効果的に利用するための，クエリの依存性解析を用いた QueryCache のメンテナンス手法(5.2)と，無効化用インデックスを用いた DataCache のメンテナンス手法(5.3 節)の効果を検証する．

最後に，グラフデータ用のインデックスであるエッジインデックスを圧縮したとき(5.4 節)のサイズの削減率と，そのオーバーヘッドを検証する．

6.1 測定環境

本評価において使用するベンチマーク TPC-W は，Transaction Processing Council（TPC）が仕様を策定した，ベンチマークアプリケーションである．本アプリケーションは，オンラインブックストアに関する 14 の Web インタラクションから構成される．我々は，TPC-W の，Java 実装版を使用し[Har11]，Java EE サーバー上で稼動させる．

また，TPC-W では，Emulated Browser（EB）から，上記 Web インタラクションを要求する．要求のワークロードは，3 つの mix（Shopping，Browsing，Ordering）によって定義され，それぞれ更新，照会トランザクションの割合が異なる．Shopping mix は最も一般的なワークロードとされており、更新 20%，照会 80%である．Browsing mix は照会の多いワークロード（更新 5%，照会 95%），Ordering mix は更新の多いワークロード（更新 50%，照会 50%）である．それぞれの評価は，各 Web インタラクションごとに規定されたレスポンス時間以内に返された，1 秒当たりのインタラクション数（WIPS）で示される．なお，本稿では，item 数 10,000 (6.3 節以降は 100,000)，クライアント数 100 用の TPC-W 用データベースを利用した．また，HTTP によって EB と Web アプリケーションサーバ間の通信が行われるようにした．

測定に利用したデータベースは，DB2 v9.129，J2EE サーバは，WebSphere Application Server v7.0 である．OpenJPA は OpenJPA 1.2.0 を使用した．測定に利用するサーバーとして，クライアントには 2 プロセッサ（Dual Core Opteron2218），4GB メモリ のマシンを利用した． Web アプリケーションサーバー，データベースサー

バーには 2 プロセッサ（Dual Core Opteron2222），8GB メモリ構成のマシンを利用した．なお，全ての OS は，ReadHat Linux 4 を利用した．

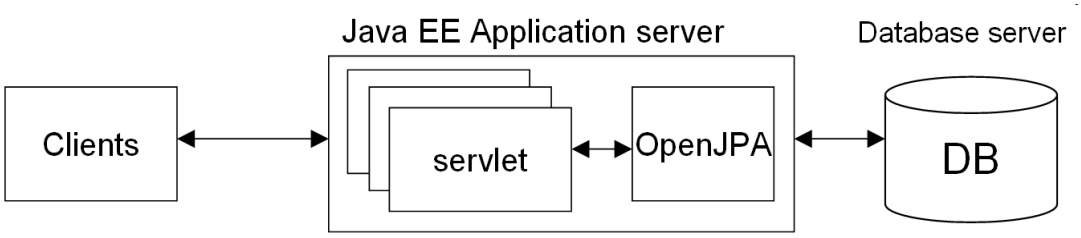


図 6.1　実験システム構成

図 6.1 は，実験のシステム構成図である．クライアントは 1 台のマシンから，マルチスレッドで複数の EB をたちあげてリクエストを送信する. アプリケーションサーバはサーブレットでリクエストを処理するが，データベースアクセス部分を OpenJPA からデータにアクセスするように変更した．もとのプログラムでは，JDBC を直接使用してデータベースにアクセスしている.

エッジインデックスの実験ではグラフ形式のデータを必要とするため，4 章と同じように，選挙関連の Twitter のデータセットを用いてインデックスを生成する.

6.2 OpenJPA 基本性能結果

6.2.1 JDBC 直接使用と，OpenJPA アクセス(キャッシュ OFF) の性能比較

まず, OpenJPA のキャッシュを OFF にして利用したアプリケーションと従来の JDBC を直接使用したアプリケーションの TPC-W の性能比較を行った．図 6.2 は，TPC-W の各 mix においてクライアント数を変更して測定した時のスループット最大値，図

6.3 は各 mix でのアプリケーションサーバ，データベースサーバの CPU 使用率を表している．

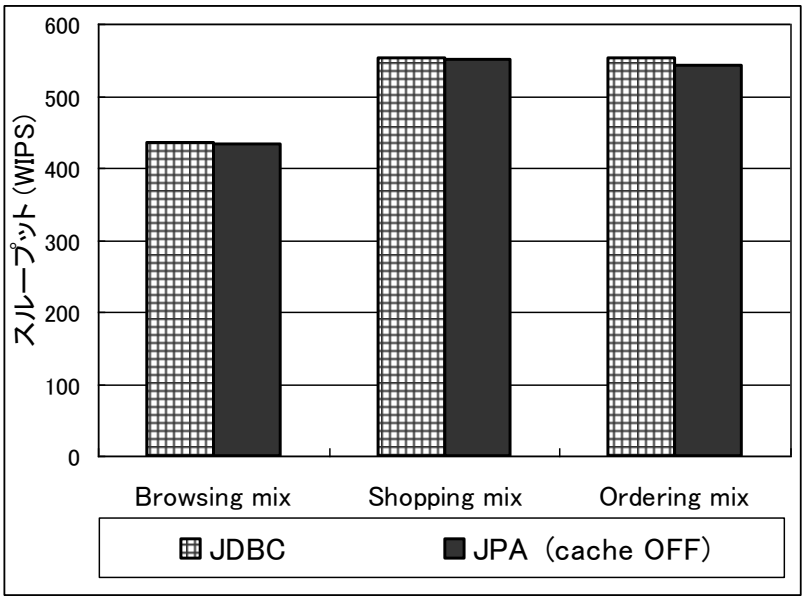


図 6.2　JDBC アクセスと JPA アクセス(cache OFF)の比較

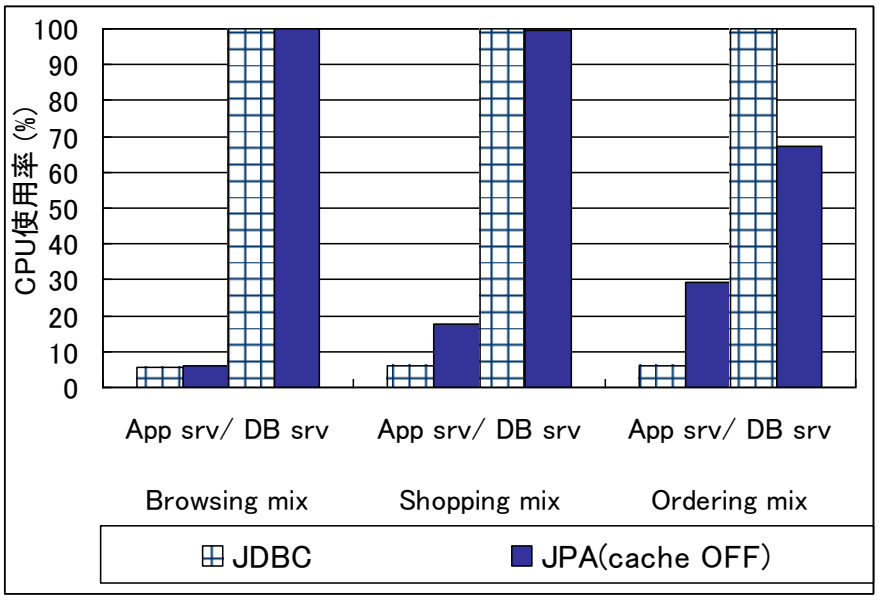


図 6.3　アプリケーションサーバ，データベースサーバの CPU 使用率

図 6.2 に見られるように，各 mix において，性能の差はほとんど無い．したがって，JDBC を直接使用せず，OpenJPA を利用してデータをエンティティオブジェクトとして扱う場合でも，そのオーバーヘッドは無視できることが分かる．Browsing mix のスループットが Shopping mix よりも低いのは, 本のベストセラー表示等の SQL 処理が複雑でデータベース処理時間のかかる照会トランザクションが，より多く含まれているためである．

また，CPU 使用率は JDBC のアクセスのデータベースサーバの CPU 使用率がすべての mix において 100%に達しており，データベースサーバがボトルネックになっている．OpenJPA は，Browsing mix, Shopping mix はデータベースサーバが 100%に達しているが，Ordering mix では，60%でスループット最大になっている．一方，アプリケーションサーバの CPU 使用率は JDBC が 10%以下であるのに対し，OpenJPA は 25%前後である．Ordering mix において，OpenJPA のデータベースサーバの CPU

使用率が 100%に達する前にスループット最大に達しているのは，更新トランザクションを処理する OpenJPA にてロックが衝突する等により待ちが発生している可能性がある．

6.2.2 OpenJPA アプリケーションのキャッシュ ON・OFF の性能比較

OpenJPA のキャッシュ OFF 時と，DataCache，QueryCache を ON にした時の TPC-W の性能結果を以下に示す(図 6.4, 6.5).

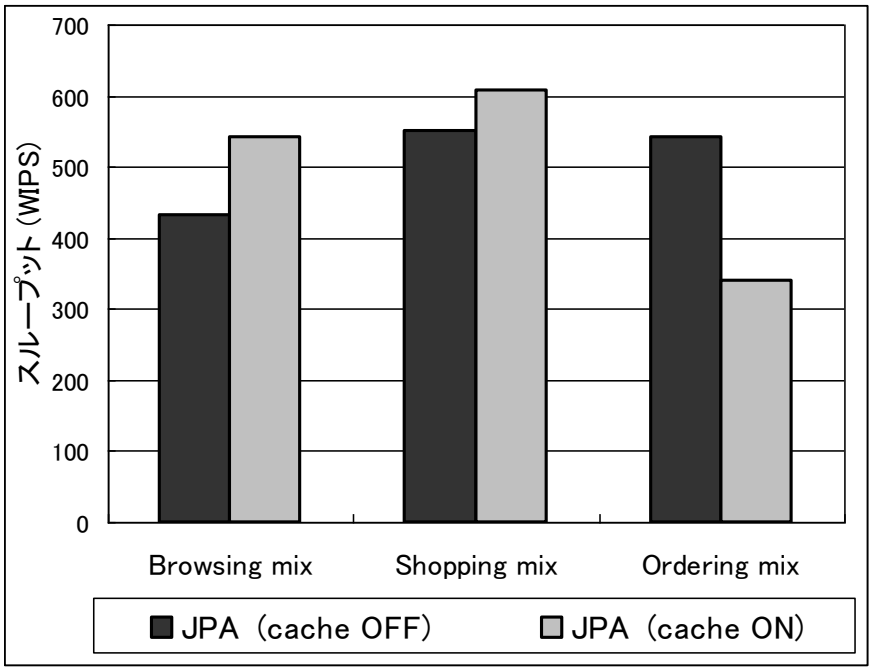


図 6.4　OpenJPA アクセス キャッシュ ON・OFF の性能比較

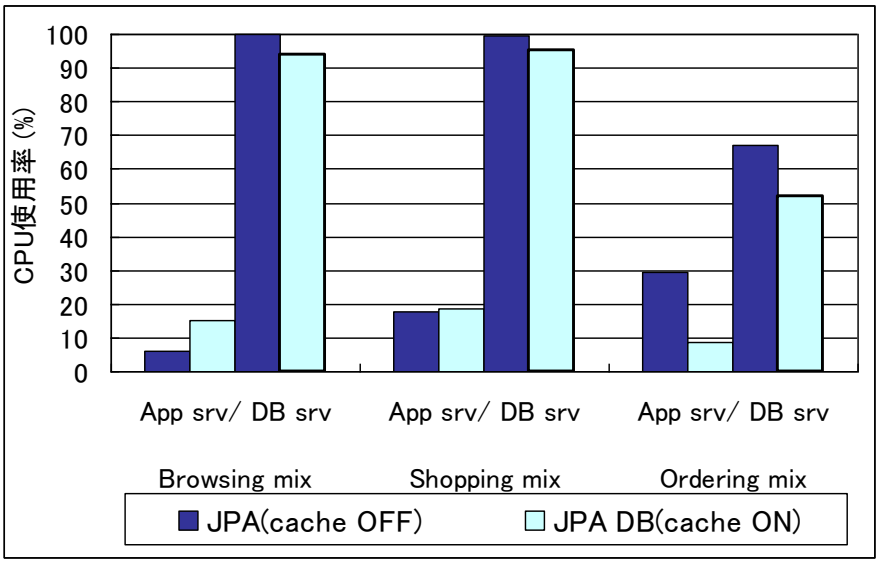


図 6.5　アプリケーションサーバ，データベースサーバの CPU 使用率

キャッシュ ON のスループットは OFF 時と比較して，Browsing mix では 1.25 倍, Shopping mix では 1.1 倍のスループットにあがる一方，Ordering mix では，0.6 倍のスループットに落ちている．Shopping mix のスループットが Browsing mix より更新が多いのにも関わらずスループットが高いのは, 前節と同様の理由による．また，両者の CPU 使用率は Browsing mix, Shopping mix はデータベースサーバがボトルネックになるまで CPU を使い切っているが，Ordering mix では，CPU を使い切れていない．

照会トランザクションが多いシナリオではキャッシュ有効時の方が高いスループットを得ることができるが，更新トランザクションが半分の割合を占めるようなアプリケーションでは，逆にキャッシュのメンテナンスのコストがオーバーヘッドとなり，スループットが落ちてしまうことが分かる．

以上の結果より，照会トランザクションの割合が多いアプリケーションでは，キャッシュを有効にした時が最も高いスループットを得ることが示された．一方，更新トランザクションが半数を占めるようなアプリケーションでは，キャッシュのメンテナンスのコストが逆にオーバーヘッドになってしまうことが分かった．したがって，キャッシュ有効時のアプリケーションはデータ更新に伴うオーバーヘッドを最小にすることが重要となる．

6.2.3 異なるキャッシュメンテナンス手法によるオーバーヘッドの比較

前章で述べたように，JPA のデータ更新方法の違いにより，OpenJPA キャッシュのメンテナンス手法も異なる．この手法の差は，更新トランザクション時のオーバーヘッドに影響する．図 6.6 は EntityManager のメソッドのみを使用した場合と，JPQL の更新が存在する場合との TPC-W の性能比較を表している．

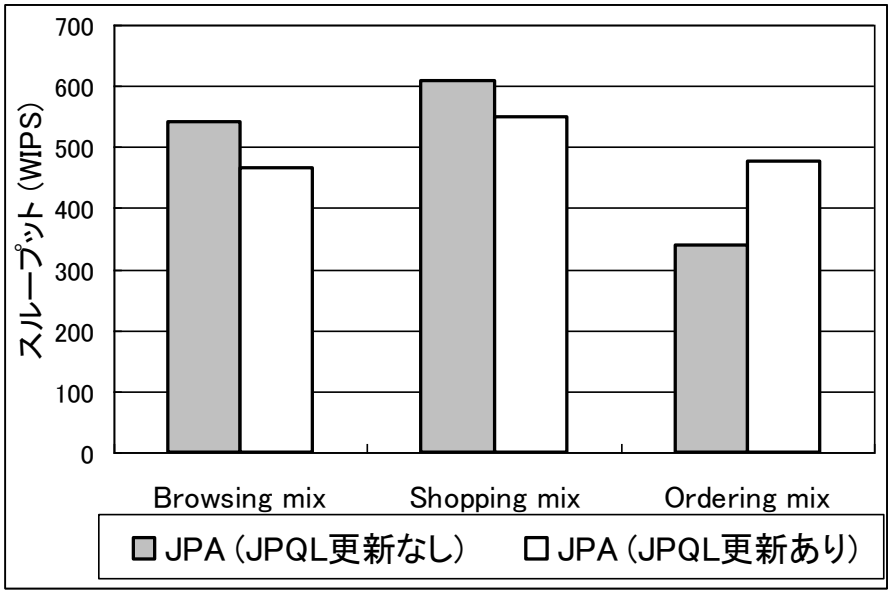


図 6.6　異なるキャッシュメンテナンス手法による性能比較

Browsing mix, Shopping mix では EntitiManager のメソッドから更新したほうが高いスループットであり，Ordering mix では JPQL からの更新のほうが高いスループットとなっている．これは，更新トランザクション時に EntityManager メソッドを使用すると DataCache のデータは無効化されないため，キャッシュヒット率が高くなる．しかし更新の多いアプリケーションになると，更新時メンテナンスのコストが逆にオーバーヘッドになり，JPQL からの更新時に DataCache をテーブルごと無効化した方が，高いスループット値になっている．図 6.6 の JPA(JPQL 更新なし)の結果は図 6.4 の JPA(cache ON)の結果と同じものである．

JPA のデータ更新の方法について，どちらを選択すれば高いスループットを得られるかは，アプリケーションのデータアクセスパターンに依存する．TPC-W では，単一のデータを更新するトランザクションが多くを占めるために，JPQL からの更新を行わない方が照会トランザクションの多いシナリオで高いスループットを得ることができた．

もし，一度の更新トランザクションで，テーブルのデータを複数更新するトランザクションが多いアプリケーションの場合，JPQL を用いてデータベース側のみに更新を反映させ，OpenJPA 側は DataCache のデータを削除してしまった方が，照会トランザクションが多くを占めるアプリケーションの場合でもスループットが高くなる可能性もある．

以上， OpenJPA のベンチマーク測定比較により，従来の JDBC 直接使用の Web アプリケーションから OpenJPA 使用へ変換してもそのオーバーヘッドは無視できる程であることを確認し，さらに OpenJPA のキャッシュ機能を有効にすることで，照会トランザクションの多いアプリケーションではより高いスループットを得られることを示した．また，JPA の更新手法により OpenJPA の DataCache のメンテナン

ス手法が異なり，この違いが更新トランザクション時のキャッシュメンテナンスのオーバーヘッドに影響することを明らかにした.

6.3 クエリ依存性解析を用いた QueryCache のメンテナンス手法の測定結果

比較手法として(1) NoCache: OpenJPA キャッシュなし，(2) Base: テーブル単位のキャッシュ無効化，OpenJPA のオリジナル機能，(3) Column-based: 更新クエリの列情報を利用したキャッシュ無効化，(4)Value-based: 更新クエリの値情報を利用したキャッシュ無効化　と用いる．図 6.7 は各 mix での TPC-W の最大スループットを表している．表 6.1 に QueryCache のキャッシュヒット率を示す.

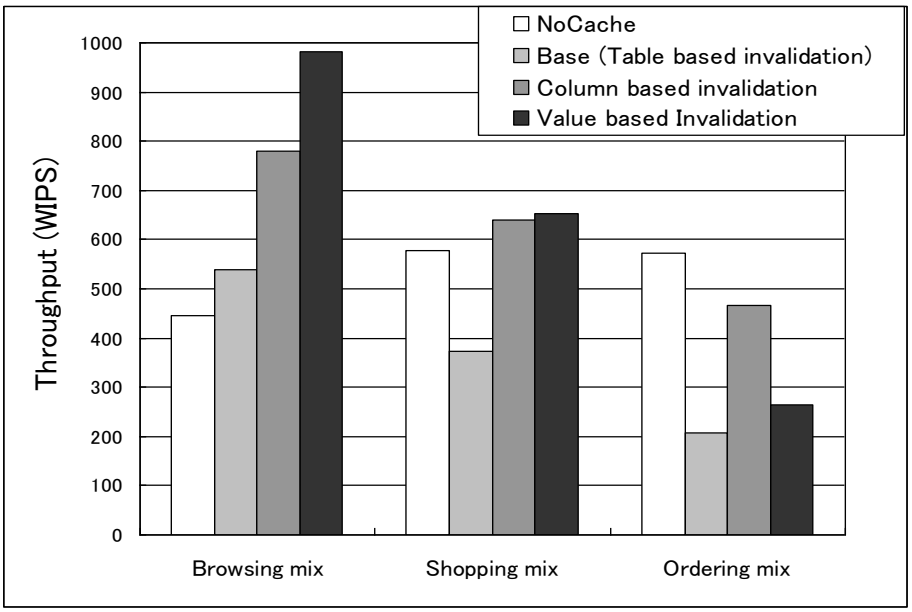


図 6.7　異なるキャッシュメンテナンス手法による性能比較

Column-based と Value-based の結果はすべての mix において，Base よりも高い結果となっている．Browsing mix での Value-based の無効化は，NoCache の結果と比較して，約 2 倍のスループットを得ており，QueryCache のヒット率は，17%から 92%へ大きく向上している．しかしながら，Column-based と Value-based のスループットは更新頻度が上がるにつれて，スループットが低下している．これは，細粒度のキャッシュメンテナンスのオーバーヘッドが徐々に大きくなっているためと考えられる．Ordering mix の Value-based のキャッシュヒット率は依然として 71%と，Column-based よりも高い値となっているが，スループットは Column-based のほうが高い結果になっている．

表 6.1　QueryCache のキャッシュヒット率

	Browsing mix			Shopping mix			Ordering mix		
	Base	Column	Value	Base	Column	Value	Base	Column	Value
QueryCache	17%	24%	92%	8%	14%	82%	4%	21%	71%

次に，Browsing mix と Ordering mix の間の照会と更新クエリの割合をより詳細に変更して，スループットを測定する (図 6.8)．x 軸は照会と更新クエリの割合であり，y 軸の値は，各割合での NoCache のスループットの値を 1 としたときの，3 種類のキャッシュメンテナンス手法での相対値を示している．この結果により，Base のスループットは更新クエリの割合が 10%を超えた付近にて，NoCache のスループットを下回っている．すなわち，NoCache の場合は，更新クエリの割合が 10%以上のアプリケーションではキャッシュの効果を発揮できなくなるということである．

一方，Column-based と Value-based の場合は，更新クエリの割合が 25%程度までは NoCache より高いスループットを維持している．また，更新クエリの割合が 20%

を超えた付近にて，Column-based のスループットは Value-based の結果と同程度になり，それ以降は Value-based よりもやや高いスループットになっている．これは，前章で述べたように，Value-based の無効化コストが全キャッシュメンテナンス手法の中で最も高いことが原因であると考えられる．

どのキャッシュメンテナンス手法を利用するかは，アプリケーションの照会・更新クエリの割合によって決定することで，キャッシュの効果を発揮できるといえる．

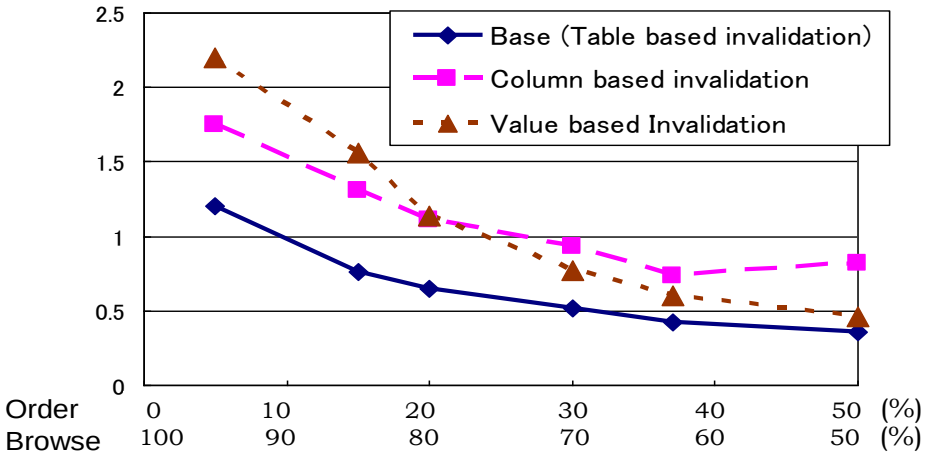


図 6.8　参照・更新クエリの割合を変更したときの性能比較

6.4 無効化用インデックスを用いた DataCache のメンテナンス手法の測定結果

無効化用インデックス，無効化用ハッシュインデックス，無効化用重み付ハッシュインデックスの効果を測定するために，TPC-W のデータを用いたマイクロベンチマ

ークにより，スループットと DataCache のキャッシュヒット率を計測する．TPC-W で使用される ITEM テーブル(主キー: i_id，100,000 レコード)を用いて，更新クエリは，ITEM テーブルの 1 属性である author の id (a_id)を指定して該当するレコードのデータを更新する．照会クエリは ITEM テーブルの id (i_id) を指定してレコードを取得するクエリである. 無効化用インデックスは, この更新クエリ用に生成される. author id の最大数は 25,000 である.

6.4.1 測定シナリオ

以下に本実験での問合せパターンを紹介する．これらのパターンは，実 Web アプリケーションのアクセスパターンに即するために，パレートの法則[Av05]をもとに(= 80 対 20 の法則)して生成されている.

Scenario 1: **Uniformly access**

照会・更新クエリ共に一様に ITEM テーブルのデータにアクセスする.

Scenario 2: Read transaction concentration

照会クエリのうち，80%のデータアクセスを，ITEM テーブルの 20%のデータに集中させる．これは，例えば ITEM のデータの中の 20%のデータが人気があり頻繁に照会されるということを表す.

Scenario 3: Read and Write transaction concentration at the same region

照会・更新クエリ共に，80%のデータアクセスを，ITEM テーブルの 20%のデータに集中させる．これは，例えば ITEM のデータの中の 20%のデータが人気があり頻繁に紹介されると共に，購入されて更新が発生するということを表す.

Scenario 4: Read and Write transaction concentration at overlapped region

ITEM テーブルの中の 20%に更新・照会のアクセスが集中, 別の 10%に照会のみアクセスが集中する．つまり，全データの 30%の領域に対し全照会クエリ数の 80%を集中させ, その 30%の領域中の 2/3 の領域に全更新クエリ数の 80%を集中させている．これは，例えばインターネット上で販売されている商品のテーブルがあるとして，人気がある売れ筋の商品は頻繁に参照されると共に商品購入によるデータ更新の頻度が共に高く，またキャンペーン商品や新規入荷商品には閲覧頻度が高く参照のみのアクセスが集中するところがあることを想定している.

アクセスデータの偏りは上記のシナリオで固定し, その状態で得た無効化用ハッシュインデックスの各領域の重み値をもとに，無効化用重み付ハッシュインデックスを生成して測定する．測定中にアクセスパターンは変化しないため，無効化用重み付ハッシュインデックスの再編成は発生しない.

6.4.2 実験結果

まず最初に，Scenario 1 において，無効化用インデックスを導入したときと，しないときの結果を比較する．図 6.9 は最大スループットを，表 6.2 は DataCache のヒット率を表している．”No Index”は無効化用インデックスのない，オリジナルの OpenJPA の実装である．“Index (size 25000)” は，メモリサイズの制限をせずに無効化用インデックスを生成した時の結果である. インデックスのキーのサイズが 25,000 であり，これは，全 a_id の値をキーにしてインデックスが生成されていることを示す．すなわち, 無効化用インデックスを最大限使用した最高スループットとなる．“Index (size 12500)”は，キーサイズ 25,000 のインデックスの半分のキー数で生成されたインデックスである．ITEM テーブルのデータを DataCache に格納した時のメモリサイズは 81.3 Mbyte であり，“Index (size 25000)”の無効化用インデックスが消費したメモリサイズは 34.1 Mbyte であった.

図 6.9 と表 6.2 の結果により，スループット，キャッシュヒット率共に無効化用インデックスを導入することで，オリジナルの OpenJPA の性能を大きく上回る結果となっている．“Index (size 25000)”のスループットは，オリジナル(”No Index”)の OpenJPA と比較して約 9.7 倍向上している. ”No Index”のキャッシュヒット率が 0.1%と低いのは，更新クエリが発行されるたびに，ITEM テーブルのデータを全て無効化してしまうためである.

対して，“Index (size 25000)”のキャッシュヒット率は 78%という高い結果となっている．サイズが半数である“Index (size 12500)”の場合でも，キャッシュヒット率は 65%であり，これは“Index (size 25000)”のヒット率と比較して，約 88%のヒット率を維持できている．なお，Secnario 1 のアクセスパターンは一様であるため，無効化用重み付ハッシュインデックスは生成されていない.

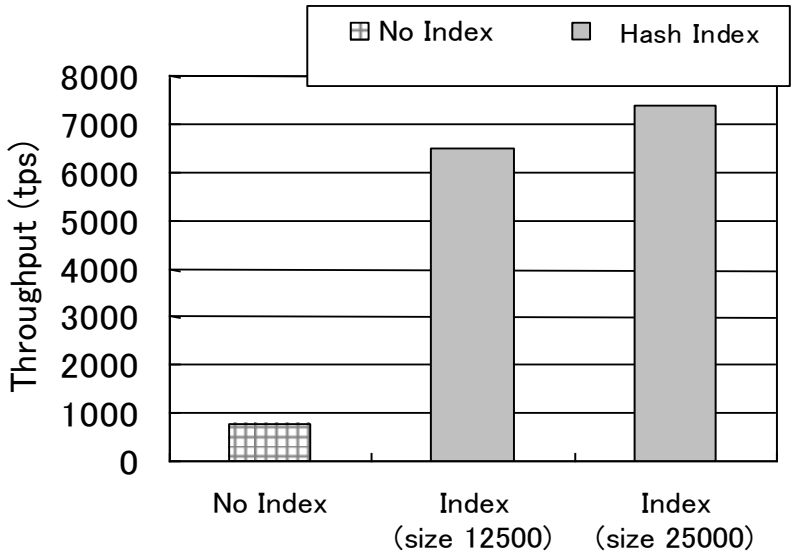


図 6.9　Scenario 1 のスループット

表 6.2　Scenario 1 の DataCache のキャッシュヒット率

No Index	Index (12500)	Index (25000)
0.1%	65%	78%

次に，Scenario 2, 3 and 4 の，無効化用ハッシュインデックス，無効化用重み付ハッシュインデックス(= Optimized Hash Index)のスループットとキャッシュヒット率を図 6.10 と表 6.3 に示す．インデックスのサイズは，12,5000 としている．重み付ハッシュインデックスは，ハッシュインデックスよりも，全シナリオにて高いスループットになっている．例えば，Scenario 2 では 24%の性能向上である．これはアクセスの偏りを考慮して再構成した重み付のハッシュインデックスが効果があることを示している．

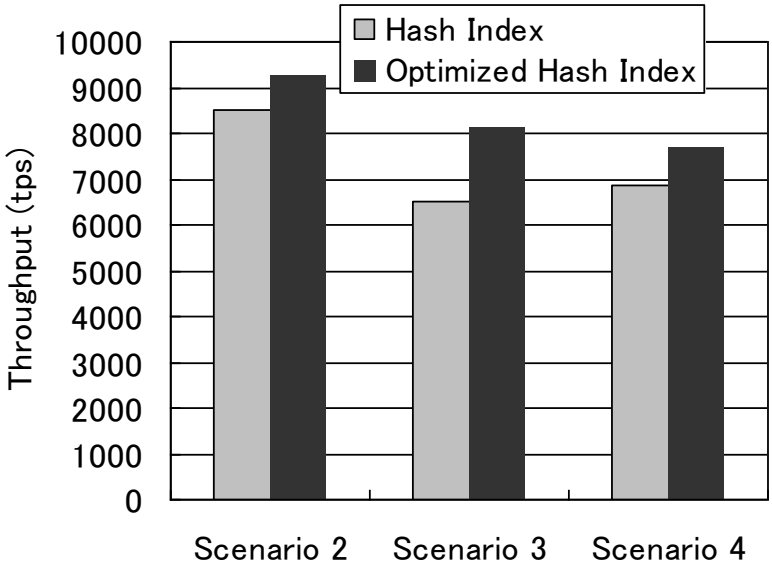


図 6.10　　Scenario 2, 3 and 4 のスループット

表 6.3　Scenario 2, 3 and 4 の DataCache のキャッシュヒット率

Scenario	2	3	4
Hash Index	75%	65%	68%
Optimized Hash Index	77%	71%	70%

続いて, 無効化用インデックスのサイズを様々な値に変更した時の, Scenario 4 での無効化用ハッシュインデックスと，無効化用重み付ハッシュインデックスの性能測定結果を図 6.11, 表 6.4 に示す. インデックスの指定サイズを, 2500, 8000, 12500,

17000, 20000 に変化させて性能を測定した．したがって，ハッシュインデックスのコリジョン数は，1 領域あたり約(25000/指定サイズ)となる．

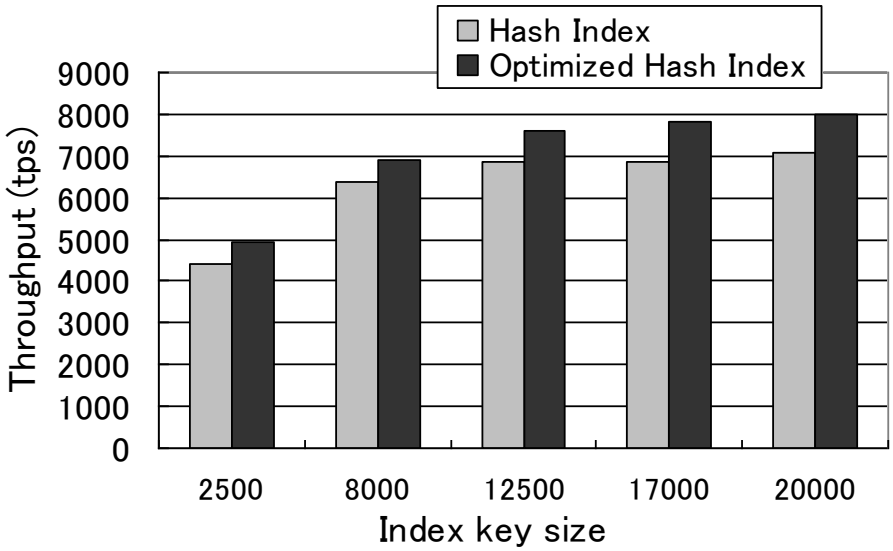


図 6.11　インデックスのサイズを変更した時のスループット

表 6.4　図 6.11 の各結果での DataCache のキャッシュヒット率

Index Key Size	2500	8000	12500	17000	20000
Hash Index	34%	62%	68%	69%	70%
Optimized Hash Index	42%	64%	70%	73%	75%

インデックスのサイズが変化しても重み付ハッシュインデックスの方が常に高いスループットとなり，その差は最大で 16%の差(サイズ 17,000)となっている．ま

た，表 1 のキャッシュヒット率も同様に最大で 8%(サイズ 2,500)高い結果である．重み付ハッシュインデックスでは，アプリケーションのデータアクセスの偏りに合わせて参照頻度の高い領域をより細粒度にメンテナンスすることが可能となり，ヒット率がより向上したと考えられる．

また，サイズ制限無しの場合(25,000)のスループットは 8545tps であったため，重み付ハッシュインデックスではインデックスサイズが半分（12,500）の状態で約 90%のスループットを維持できている．サイズが 1/10 の 2,500 では約 57%のスループット維持となり，無効化用インデックス無しの OpenJPA の性能と比較すると約 6 倍のスループットを得ている．

使用メモリに対して，どの程度の性能が得られているのかを測るために，ハッシュインデックスの実メモリサイズと, メモリ 1Mbyte あたりのスループット(図 6.11 の重み付ハッシュインデックスのスループット/実メモリサイズ)を表 6.5 に示す．インデックスの実メモリサイズ削減率は，無効化用インデックスの key となるパラメータの種類数とそのテーブルのデータ数により変化する．value 数に対する key 数の割合が高いほど，ハッシュインデックスに変換した時に key のサイズを削減させることができ，全体のインデックスサイズ削減率がより高くなる．本実験では，インデックスサイズ 8000 の 1Mbyte あたりのスループット値が最高となり，インデックスサイズ削減に対する効果が最も高かった．

表 6.5 インデックスのメモリサイズと 1Mbyte あたりのスループット

Index Key Size	2500	8000	12500	17000	20000
Size (Mbyte)	14.9	18	21.8	26	29
Throughput (tps/Mbyte)	297	353	314	264	244

6.5 エッジインデックスの圧縮手法適用結果

6.5.1 測定シナリオ

前章にて紹介した各行列圧縮手法にてエッジインデックスを圧縮したときの圧縮率について，実データを用いて比較する．また，圧縮するときのコスト，データ問合せ時に復元するときのコストも検証する．

実権に用いるエッジインデックスは，日本の首相(ユーザーA とする)と，選挙の立候補者であり，Twitter を利用して情報を頻繁に発信していたユーザーB とユーザーC が発信したツイートから生成した拡散ネットワークの集合を対象としてそれぞれインデックスを生成する．拡散ネットワークは，各ユーザーが発信したツイートの，リツイート数が多い順の 50 件のツイートに対して生成する．その中で最もリツイート数が多いツイートは約 2000 件，最も少ないツイートでは約 50 件のリツイート数であった．

6.5.2 行列圧縮率

上記 3 ユーザーごとに拡散ネットワークからインデックスを生成した．各ユーザー50 ツイートの拡散ネットワークを生成しているので, インデックスの行数は 50 となる. 列数は, それぞれのユーザーの 50 件の拡散ネットワークで出現した, エッジ({リツイート元のユーザー名，リツイートしたユーザー名}が 1 エッジ)の総数になる．

比較対象は, 行は TweetID, 列は EdgeID の行列で表現したインデックス (オリジナル)，圧縮行格納方式を用いて圧縮したインデックス(CRS)，連長圧縮を用いて圧縮したインデックス(RLE)，頻出アイテム集合発見を前処理として生成してから連長圧縮で圧縮したインデックス(RLE_freq)，Edge をリツイート頻度順に並べ替えてから連長圧縮したインデックス(RLE_orderRT)の合計 5 つのインデックスである．

サイズの測定方法は，各インデックスを，int 型の二次元配列として格納したときの，総配列要素数とする．オリジナルのサイズは，行数×列数となる．CRS の場合は，3.2.1 に示した IA’,JA の要素数の和となる．

表 6.6 に各ユーザーの拡散ネットワークデータに対して生成したインデックスと，その圧縮率を示す．今回実験した 4 手法はいずれももとの行列のサイズと比較して，90%以上の圧縮率となった．最も高い圧縮率だったのは CRS であった．

RLE の結果は，RLE のみの結果よりも，RLE_freq と RLE_orderRT のほうが圧縮率が少し高くなっており，行方向に 1 が続くようにエッジを並べ替えた効果があったと言える．また，RLE_freq と RLE_orderRT の圧縮率は RLE_orderRT のほうがやや高い結果となった．前処理の時間は RLE_orderRT のほうが小さいため，シンプルにリツイート順で並べるだけでも効果を得ることができたことが分かる．

表 6.6 拡散ネットワークに対するインデックス圧縮率

ユーザーA	オリジナル	CRS	RLE	RLE_freq	RLE_orderRT
サイズ	291650	12623	23662	21058	19769
圧縮率(%)		**95.7%**	91.9%	92.8%	93.2%

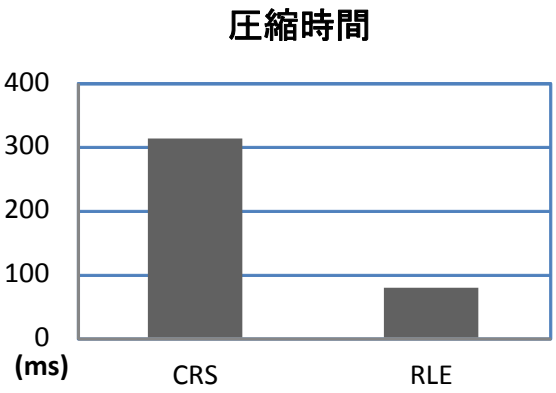
ユーザーB	オリジナル	CRS	RLE	RLE_freq	RLE_orderRT
サイズ	701700	24876	46803	45177	44087
圧縮率(%)		**96.5%**	93.3%	93.6%	93.7%

ユーザーC	オリジナル	CRS	RLE	RLE_freq	RLE_orderRT
サイズ	251100	8328	14409	13765	13445
圧縮率(%)		**96.7%**	94.3%	94.5%	94.6%

本実験データにおいては，おおむね CRS のほうが高い圧縮率であった．CRS は，1 の位置のみを記憶するため，1 の出現数が少ないほど圧縮率が高くなる．1 の出現数やリツイートしたユーザーの重複度によっては，インデックス内に 1 が連続するパターンが増加し，RLE が有利になる可能性もある．したがって，拡散ネットワークデータの性質により，有効な圧縮手法が変化する場合があると考えられる．

6.5.3 圧縮/復元のオーバーヘッド

次に，オリジナルのインデックスから CRS または RLE を適用して圧縮した時間と，任意の TweetID を 1 つ指定したクエリが発行されたときに，インデックスを該当行のみ復元し，データベースに問合せるべき EdgeID のリストを取得するまでの処理を 1000 回繰返したときの実行時間を検索時間として測定した結果を図 6.12 に示す．



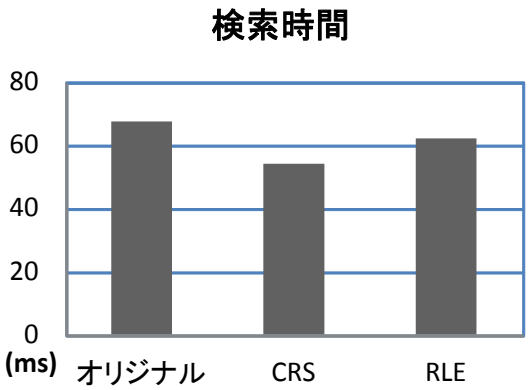


図 6.12 圧縮時間と検索時間の比較

圧縮時間は，CRS と比べて，RLE のほうが約 4 倍はやい結果となった．一方，検索時間は両者ともオリジナル行列とほぼかわらなかった．むしろオリジナルよりも高速に検索できているのは，オリジナルの場合は各列の値が 0 か 1 かを順に比較するのに対し，圧縮した行列は，1 の箇所のみをピックアップして Edge リストを取得することができるからであると考えられる．

したがって，インデックスの生成タイミングをどのような頻度で実行するかによるが，生成コストを重視したい場合は，圧縮率が少し下がっても RLE で圧縮した方が良いといえる．

6.6 まとめ

本章では，提案システム内の蓄積されたデータを処理する部分の，データアクセスの処理性能の評価し，提案したデータアクセス最適化手法の効果を検証した．データアクセスには，POJO ベースの O/R マッピングアーキテクチャである JPA を利用するため，まずその基本性能を評価した．データアクセス処理の高速化には，キャッシュの利用が効果的であるが，更新クエリの割合が増えてくると，従来のキャッ

シュメンテナンス手法では性能が低下することを確認し，我々の提案手法である列単位，値単位で更新の影響が及ぶキャッシュのみを無効化する，より細粒度のキャッシュメンテナンス手法により，性能が改善することを示した．また，無効化すべきデータを特定するための無効化用インデックスが効果を維持できるよう，メモリ領域の容量にあわせてサイズ変更可能な無効化用ハッシュインデックス導入し，キャッシュヒット率を大幅に向上できることを示した．クエリアクセスに偏りがある場合は，無効化用重み付ハッシュインデックスを導入することにより，さらに性能を高く維持できることを示した．

これらの提案手法は，本論文で述べているソーシャルデータを対象にしたシステムだけに限る話ではない．本研究で提案している，リアルタイムなストリームデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えたシステムであれば，どのようなストリームデータを処理したとしても，定期的にリアルタイム処理から退避されてきたデータを受け取るため，本提案手法を適用可能である．

最後に，ソーシャルメディアの情報拡散ネットワークのようなグラフデータアクセスの高速化のため，ビットマップのインデックス導入した．このインデックスは大規模な粗行列になることが想定されるため，実際に Twitter のデータを用いて，行列を圧縮してサイズ削減の効果を評価した．更新時の再圧縮や，検索時の復号のオーバーヘッドを考慮すると，RLE で圧縮するほうが良い．さらに，ソーシャルメディアデータの特性である，情報拡散と拡散されやすい経路等を考慮して，あらかじめ行列を並び替えることにより，さらに圧縮率を高めることが出来ることを示した．

第7章　結論

7.1 まとめ

近年，様々なデバイスから位置情報・気象・ソーシャル・センサーデータなど，膨大な量のデータが日々発信され続けており，地球上で生成されるデータは 2013 年から 2020 年の間で 4 兆 4,000 億ギガバイトから 44 兆ギガバイトへと 10 倍の規模に拡大すると予想されている[EMC]. これらのデータは綺麗に構造化されたデータではなく，欠損を含んでいたり，自然言語で書かれているものも多く，このような不確かなデータが全データの 80%を占めるともいわれている．このような膨大なデータを利用して関連性を見つけたり，変化を検知したりと，様々な観点から分析する要求は年々高まっており，それらを実現するための処理基盤は大規模なデータをリアルタイムに，また高速に扱えることが重要となる．

このようなストリームデータをリアルタイムに分析するため，2000 年以降ストリームデータ処理を主にした DSMS (Data Stream Management System)が登場した．従来の DBMS (DataBase Management System) と異なり, 連続的に発信されるデータを対象にリアルタイムにデータ加工や分析結果を返し続けるシステムである．

従来の DBMS は，データがあらかじめデータベースに格納されている状態から，そのデータを対象に SQL によるクエリが発行され，所望のデータが返されるアーキテクチャである．発行されてくるクエリのパターンは様々である．一方，DSMS は流れてくるストリームデータに対して，連続的にクエリが実行される．データへの問合せ部分に着目した時，あらかじめ指定したウィンドウと呼ばれる単位にデータを区切り，その範囲内に到着したデータに対してクエリの演算を施すという特徴がある．また，ストリームデータはリアルタイムにデータが発信されてきており，その流量が時間や何かのタイミングで変化する場合は，常に一定の性能を保って処理し続けるシステムを保証することは難しい．そこで，インプットデータは，ランダムにサンプリングする等のフィルタリングを実施する．

DSMS で演算処理された後のデータは，一般的に破棄されていた．センサーデータはリアルタイムに到着し続けるため，全データをストレージに格納することは現実的ではなく，そもそもリアルタイムに異常検知したり，定期的に平均値を返すことなどを目的にしていたため，オリジナルデータを保持する必要もなかったのである．ところが，近年のストレージの格納量の大規模，高速化にも伴い，DSMS のオリジナルデータもストレージに格納され，それをオフラインで分析対象にすることも増えてきた．しかしながら，ストリームデータのリアルタイムの分析とストレージに格納したオフラインでの分析を同時に考えたシステムは検討されていない．

そこで本研究では，リアルタイムなストリームデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えたシステムを提案し

た．本研究ではストリームデータのリアルタイム性の収束のタイミングを議論し，リアルタイム分析とオフライン分析との使い分けの手法を検討した．

また，扱うストリームデータにおいては，近年，あらゆるものがインターネットに繋がり，ネットワークを介して様々な情報を交換するようになる IoT (Internet of Things) の拡大や，スマートフォンの普及も影響して，世界中の各個人がアカウントを持ちテキストや写真，動画を送受信するソーシャルネットワーキングサービス，車等のデバイスからの GPS や機器情報のセンサーの発信など，世界中のデータは今も増加し続けて大規模になっており，一般にリアルタイムデータといっても，多岐にわたることが分かる．これらの情報を利用することにより，ストリーム処理を分析する際に，各データの特性を考慮した最適化が行えるのではないかと考えられる．そこで本研究では，大規模リアルタイムデータの一つであるソーシャルメディアデータを対象として，ソーシャルメディアデータがもつ特徴を利用したデータアクセスの最適化手法を，**逐次的にストリーム処理する機構**と，**蓄積されたデータを処理する機構**それぞれに提案した．

リアルタイム分析を実施するための，**逐次的にストリーム処理をする部分**については，出来るだけフレッシュなデータをストリーム処理内に維持するために，各メッセージの流行が収束するタイミングを拡散モデルを用いて早期に推定し，時間ウィンドウ幅をカスタマイズしてメンテナンスする手法を提案し，Twitter の実データを用いてその効果を示した．次に，代表的なリアルタイムの問合せパターンを実際に測定し，インメモリなデータアクセスは高速にクエリが実行できることを示した．しかし，複雑な問合せパターンは演算のオーバーヘッドがあり，単純なクエリより応答時間が長くなっていた．そのようなクエリに対しては，計算の中間結果を保持したビューを導入して問合せの高速化が実現できることを示した．

また，データバースト時のクエリの応答時間の劣化を回避するため，各メッセージに付随する情報に着目して重要度を計算し，重要度が低いと判断されたメッセ

ージをフィルタリングして処理するデータ量をコントロールする手法を検証した．これにより，従来のランダムなフィルタリングと比較して，同程度のサンプリング量において，より問合せ時間が短く処理できていることを示した．また，問合せ結果の精度が高く維持できていた．

提案システム内の**蓄積されたデータを処理する部分**については，データアクセスの処理性能の評価し，提案したデータアクセス最適化手法の効果を検証した．データアクセスには，POJO ベースの O/R マッピングアーキテクチャである JPA を利用するため，まずその基本性能を評価した．データアクセス処理の高速化には，キャッシュの利用が効果的であるが，更新クエリの割合が増えてくると，従来のキャッシュメンテナンス手法では性能が低下することを確認し，我々の提案手法である列単位，値単位で更新の影響が及ぶキャッシュのみを無効化する，より細粒度のキャッシュメンテナンス手法により，性能が改善することを示した．また，無効化すべきデータを特定するための無効化用インデックスが効果を維持できるよう，メモリ領域の容量にあわせてサイズ変更可能な無効化用ハッシュインデックス導入し，キャッシュヒット率を大幅に向上できることを示した．クエリアクセスに偏りがある場合は，無効化用重み付ハッシュインデックスを導入することにより，さらに性能を高く維持できることを示した．

また，ソーシャルメディアの情報拡散ネットワークのようなグラフデータアクセスの高速化のため，ビットマップのインデックス導入した．このインデックスは大規模な粗行列になることが想定されるため，実際に Twitter のデータを用いて，行列を圧縮してサイズ削減の効果を評価した．更新時の再圧縮や，検索時の復号のオーバーヘッドを考慮すると，RLE で圧縮するほうが良い．さらに，ソーシャルメディアデータの特性である，情報拡散と拡散されやすい経路等を考慮して，あらかじめ行列を並び替えることにより，さらに圧縮率を高めることが出来ることを示した．

本論文では，大規模リアルタイムデータの一つであるソーシャルメディアデータを対象として，ソーシャルメディアデータがもつ特徴を利用したデータアクセスの最適化手法を，逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構それぞれに提案した．しかしながら，提案したデータアクセスの最適化は，他の種類のストリームデータにも適用可能であると考える．

逐次的なストリーム処理では，例えば，ある有料道路区間を走る車のリアルタイムな位置情報や運転速度等のセンサーデータは，時間帯や天気，事故の有無等の道路状況によって，データの流量が大きく変化すると考えられる．これらも一種のバースト性を引き起こすと考えられ，キャパシティの制御が必要になったときには分析したい観点によって，優先度の高い車の重み値を高くしてフィルタリングすることで，問合せ精度の低下を防げることが期待できる．また，それぞれの車が，有料道路に入ってから出るまでをストリームのウィンドウとしてカスタマイズすれば，その車の時系列データを分断させることなくリアルタイム分析対象にすることができる．

蓄積されたデータを処理では，本研究で提案している，リアルタイムなストリームデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えたシステムであれば，どのようなストリームデータを処理したとしても，定期的にリアルタイム処理から退避されてきたデータを受け取るため，本提案手法を適用可能である．

7.2 今後の課題

本論文では，ソーシャルメディアデータを対象にリアルタイムなストリームデータを逐次的にストリーム処理する機構と，蓄積されたデータを処理する機構を兼ね備えたリアルタイムデータアクセス処理システムの，データアクセス処理の最適化について議論し，提案手法を評価した．従来のストリーム処理システムに対して，ソーシャルデータのもつ拡散の特徴や，付随するユーザーの情報を利用することにより，より最適なデータアクセス処理が可能になることを示した．

使用したデータセットは主に選挙関連のトピックデータであったが，リアルタイムな分析としては災害時や突発的なイベント等，他のトピックでもシナリオが考えられる．これらのデータセットを用いた場合でも同様の効果が得られるかを検証したい．

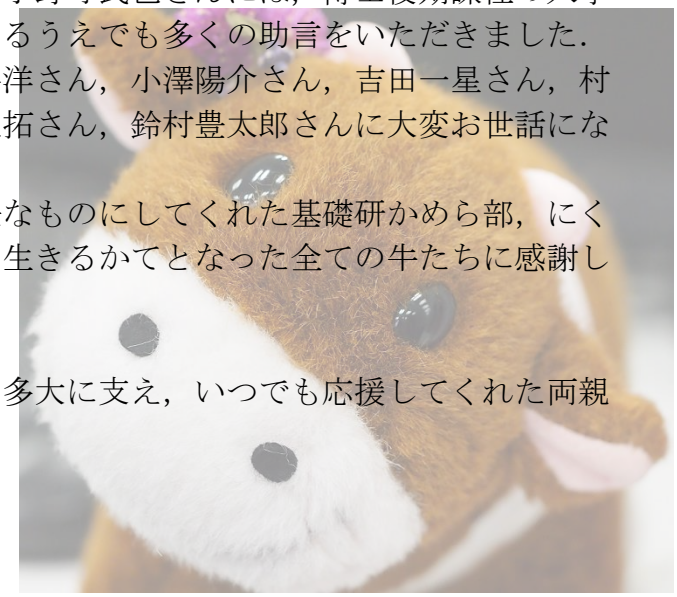
また，分析ユーザーが増えてくると，分析対象としたいトピックは様々になると考えられ，それぞれのユーザーが所望とするデータセットをすべて一つのシステムでまかなうにはデータが膨大になる可能性がある．さらに，他のユーザーには必要の無いデータセットまで問合せデータ中に混在することは性能的にも無駄である．そこで，多くのユーザーが自由な期間，タイミングで軽量に分析できるよう，システムのインスタンスをクラウド化して並列処理可能になるようにしたい．また，その時に性能上課題となる点等を考察していきたい．

謝辞

本研究を進める過程で多くの方々のご指導，ご支援を賜りました．心から感謝致します．
主任指導教員である，お茶の水女子大学理学部の小口正人教授には，本研究を進めるにあたりご指導いただき，また社会人博士ということで柔軟なご対応をいただき，大変お世話になりました．深く感謝致します．
副指導教員を引き受けてくださいました，お茶の水女子大学理学部の小林一郎教授には，本研究に関する大変有益なコメントを度々いただきました．深く感謝致します．
お茶の水女子大学理学部の吉田裕亮教授，椎尾一郎教授，伊藤貴之教授には論文審査員として大変有益なコメントをいただき感謝いたします．
小口研学生の皆さん，研究室ゼミや学会等，研究室生活においてお世話になりました．いつ見ても皆可愛くてしっかりしていて，大変心の癒しとなりました．
社会人として博士課程に通うにあたり，お茶の水女子大学増永良文名誉教授，渡辺知恵美先生には何度もご相談にのっていただき，どうもありがとうございました．

本論文執筆において，日本アイ・ビー・エム（株）東京基礎研究所の皆様に，非常にお世話になりました．小野寺民也さんには，博士後期課程の入学を快諾していただき，研究を進めるうえでも多くの助言をいただきました．
また，研究を進めるうえで，堀井洋さん，小澤陽介さん，吉田一星さん，村上明子さん，伊川洋平さん，井上拓さん，鈴村豊太郎さんに大変お世話になり，ありがとうございました．
また，研究生活を彩り豊かで愉快なものにしてくれた基礎研かめら部，にく部の皆さん，同期の皆さん，私の生きるかてとなった全ての牛たちに感謝します．

最後に，今まで生活面，精神面を多大に支え，いつでも応援してくれた両親に深く感謝します．



参考文献

[EMC] “Digital Universe Invaded By Sensors”
http://japan.emc.com/about/news/press/2014/20140409-01.htm

[Tw] Twitter https://twitter.com/

[TC] “Twitter hits 400 million tweets per day, mostly mobile”
http://www.cnet.com/news/twitter-hits-400-million-tweets-per-day-mostly-mobile/

[Enoki01] M. Enoki, I. Yoshida and M. Oguchi, “A Practical Framework for Real-time Diffusion Analysis in Social Media”, ACM International Conference on Computing Frontiers (CF), 2015

[Enoki02] M. Enoki, I. Yoshida and M. Oguchi, “Performance of System for Analyzing Diffusion of Social Media Messages in Real Time”, IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2015

[Enoki03] M. Enoki, Y. Ozawa and T. Onodera, “Performance Improvement of OpenJPA by Query Dependency Analysis”, Database Systems for Advanced Applications (DASFAA), 2010

[Enoki04] M. Enoki, Y. Ozawa, H. Horii, and T. Onodera, “Memory-Efficient Index for Cache Invalidation Mechanism with OpenJPA.”, Web Information Systems Engineering (WISE), 2012

[Enoki05] 榎美紀，小澤陽介，堀井洋，小野寺民也“メモリ効率の良い無効化用インデックスを用いたOpenJPAの性能改善”日本データベース学会論文誌, Vol.10, No.1, pp.73-78, 2011.

[Enoki06] 榎 美紀，小口 正人，“ソーシャルメディア上の情報拡散ネットワークのデータ格納サイズの圧縮”，第5回ソーシャルコンピューティングシンポジウム（SoC）, 2014

[Tn] “Twitterの利用状況/企業情報” https://about.twitter.com/ja/company

[Ke] “企業を襲う インターネットの“炎上”” http://www.nhk.or.jp/ohayou/marugoto/2014/04/0416.html

[St] “ソーシャルメディア運用・分析・監視ツール” http://dmc-navi.sendenkaigi.com/keyword/view/3

[Mat10] M. Mathioudakis and N. Koudas, “TwitterMonitor: trend detection over the twitter stream.” In Proceedings of the 2010 international conference on Management of data, pages 1155–1158. ACM, 2010.

[Asur11] S. Asur, B. A. Huberman, G. Szabo, and C. Wang, “Trends in social media - persistence and decay.” In 5th International AAAI Conference on Weblogs and Social Media, 2011.

[Lee12] Lee, C.-H, “Mining spatio-temporal information on microblogging streams using a density-based online clustering method”, Expert Syst. Appl., Vol. 39, No. 10, pp. 9623-9641, 2012.

[Nasu13]那須川 哲哉,西山 莉紗, 金山 博,吉田 一星,大野 正樹, “一人称所有格を用いたプロフィール推定” 言語処理学会 第 19 回年次大会, 2013

[Cu] “キュレーター”

http://ja.wikipedia.org/wiki/%E3%82%AD%E3%83%A5%E3%83%AC%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3#.E3.82.A4.E3.83.B3.E3.82.BF.E3.83.BC.E3.83.8D.E3.83.83.E3.83.88.E3.81.AB.E3.81.8A.E3.81.91.E3.82.8B.E3.82.AD.E3.83.A5.E3.83.AC.E3.83.BC.E3.82.BF.E3.83.BC

[Enoki07] 榎 美紀, 村上 明子, レイモンド ルディー, 小口 正人 “ソーシャルメディア上の情報拡散分析”, データ工学と情報マネジメントに関するフォーラム (DEIM), 2014

[HB] HBase http://hbase.apache.org/

[Neo] Neo4j http://www.neo4j.org/

[Gu11] M.Gupte, P.Shankar, J. Li, S. Muthukrishnan, L. Iftode, “Finding hierarchy in directed online social networks”, Proceedings of the 20th international conference on World wide web (WWW), pp. 557-566, 2011.

[In] “インメモリデータベース”

http://ja.wikipedia.org/wiki/%E3%82%A4%E3%83%B3%E3%83%A1%E3%83%A2%E3%83%AA%E3%83%87%E3%83%BC%E3%82%BF%E3%83%99%E3%83%BC%E3%82%B9

[Id] “インメモリー時代の DB 構築”

http://coin.nikkeibp.co.jp/coin/sys_ranking10/img/sample3_2.pdf

[Fagin01] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware. “ In Proceedings of the twentieth ACM SIGMODSIGACT SIGART symposium on Principles of database systems, pp.102–113, 2001.

[Jain08] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Cetintemel,U., Cherniack, M., Tibbetts, R., and Zdonik, S., “Towards a streaming SQL standard.” Proc. VLDB, 2008.

[Pol07] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E.Frantzell, “Supporting streaming updates in an active data ware-house,” IEEE International Conference on Data Engineering (ICDE), pp. 476–485, 2007.

[Babu01]S. Babu and J. Widom,”Continuous queries over data streams”, ACM SIGMOD Record, v.30 n.3, September 2001

[Ara06] A. Arasu , S. Babu , and J. Widom, “The CQL continu-ous query language: semantic foundations and query execution”, The International Journal on Very Large Data Bases (VLDB), v.15 n.2, p.121-142, 2006

[Matsu13] 松澤 有,セーヨー サンティ, 鳥海 不二夫, 陳 昱, “リツイート時系列の 3 パラメータ混合対数正規分布による分析”, 人工知能学会全国大会, 2013

[Kw10] Haewoon Kwak, Changhyun Lee, Hosung Park, Sue B. Moon, “What is Twitter, a social network or a news media?” pp. 591-600, WWW 2010.

[Gal10] Galuba, W., Chakraborty, D., Aberer, K, Despotovic, Z., and Kellerer,W., “Outtweeting the Twitterers – Predicting Information Cascades in Microblogs.”, In 3rd Workshop on Online Social Networks (WOSN), 2010.

[Asur11] Asur, S., Huberman, B. A., Szabo, G., and Wang, C., “Trends in social media: persistence and decay.” In Proceedings of the fifth International AAAI Conference on Weblogs and Social Media (ICWSM), 2011

[R] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

[E6] EJB 3.0 Expert Group, JSR 220: Enterprise JavaBeans Version 3.0 Java Persistence API, Sun Microsystems, Santa Clara, CA, 2006.

[Em02] Emmanuel ,C., Julie, M., Willy, Z. : "Performance and scalability of EJB applications", In Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '02),pp246-261 ,2002.

[Ap] Apache OpenJPA http://openjpa.apache.org/

[Hi] Hibernate http://www.hibernate.org/

[To] TopLink Essentials

https://glassfish.dev.java.net/javaee5/persistence/

[Pat07] Patrick, L., Mark, P., "An in-depth look at the architecture of an object/relational mapper", Proceedings of the ACM SIGMOD international conference on Management of data, pp.889-894 ,2007.

[Levy93] A. Y. Levy and Y. Sagiv. "Queries independent of updates", In Proc. International Conference on Very Large Data Bases (VLDB), 1993

[Gar08] Garrod, C. et. al., "Scalable query result caching for web applications", In Proc. International Conference on Very Large Data Bases (VLDB), pp. 550-561, 2008

[Mic79] Michael R. Garey and David S.Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness." W. H. Freeman, New York, NY, ,1979

[Kar82] N. Karmarkar and R. Karp, "The differencing method of set partitioning." Technical Report UCB/CSD 82/113, University of California, Berkeley, CA, 1982

[Kor98] R. Korf., "A complete anytime algorithm for number partitioning." Artificial Intelligence, 106(2):181–203, 1998

[Com] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst , “Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition”, SIAM, 1994

[Com2] 連長圧縮

http://ja.wikipedia.org/wiki/%E9%80%A3%E9%95%B7%E5%9C%A7%E7%B8%AE

[Ag94] R. Agrawal, R. Srikant: “Fast Algorithms for Mining Association Rules in Large Databases.” In Proc. International Conference on Very Large Data Bases (VLDB) pp.487-499 , 1994

[Mo97] M. Javeed Zaki, S. Parthasarathy, M. Ogihara, and W. Li: “New Algorithms for FastDiscovery of Association Rules.” KDD pp283-286 1997

[Har11] Harold, W., Ravi, R., Mowwis, M.Mikko ,H., “An Architectural Evaluation of Java TPC-W”, Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), pp.229 ,2001

[Av05] A. Narula, U. Publishers Distributors Ltd, “80/20 Rule of Communicating Your Ideas Effectively”, 2005

[Rat11] Ratkiewicz, J., Conover, M. Meiss, M. Gonc¸alves, B. Patil, S.; Flammini, A.; nd Menczer, F.. Truthy: Mapping the spread of astroturf in microblog streams. In Proc. 20th Intl. World Wide Web Conf. (WWW) 2011 .

[Mc13] McKelvey K, Menczer F “Truthy: Enabling the study of online social networks.” In: Proc. CSCW 2013.

[Gupta12] A. Gupta and P. Kumaraguru, “Credibility ranking of tweets during high impact events,” in Proceedings of the 1st Workshop on Privacy and Security in Online Social Media,. PSOSM, 2012

[Mar12] A. Marcus, Michael S. Bernstein, O. Badar, David R. Karger, S. Madden, and Robert C. Miller. “Processing and Visualizing the Data in Tweets.” ACM SIGMOD Record 40, no. 4, 2012

[Adam11] A. Marcus, Michael S. Bernstein, O. Badar, David R. Karger, S. Madden, and Robert C. Miller. “Twitinfo: aggregating and visualizing microblogs for event exploration.” In Proceedings of the 2011 annual conference on Human factors in computing systems (CHI). ACM, New York, NY, USA, 227- 236, 2011

[Gupta99] A. Gupta and I. S. Mumick, editors. Materialized Views: Techniques, Implementations and Applications. MIT Press, June 1999.

[Hri01] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: “A system for the efficient execution of multi-parametric ranked queries.” In Proc. of the ACM SIGMOD 2001.

[Cet16] Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Meehan, J. and Zdonik, S, “S-Store: A Streaming NewSQL Sys-tem for Big Velocity Applications.” In Proc. International Conference on Very Large Data Bases (VLDB), 2014.

[Wei06] Wei, Y., Prasad, V., Son, S. H., and Stankovic, J. A. “Prediction-based QoS management for real-time data streams.” In 27th IEEE International Real-Time Systems Symposium (RTSS), pp. 344-358, 2006.

[Pa01] Paul, B., Shuping, R, “Entity Bean A, B, C’s: Enterprise Java Beans Commit Options and Caching”, Middleware, pp.36-55, 2001.

[Em02] Emmanuel ,C., Julie, M., Willy, Z., “Performance and scalability of EJB applications”, In Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA ‘02),pp246-261 ,2002.

[Avr03] Avraham, L., James, T., “Improving Application Throughput With Enterprise JavaBeans Caching”, Proceedings of the 23rd International Conference on Distributed Computing Systems(ICDCS’03), pp.244-251 ,2003.

[Ben08] Ben ,W., Ali, I.,William, R., “Interprocedural query extraction for transparent persistence”, Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications(OOPSLA’08), pp.19-36 ,2008.

[Zac08] Zachary, T.,Chris, T., David, S.,Ranjit ,J.,Sorin, L., “Deep typechecking and refactoring”, Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications(OOPSLA’08), pp.37-52 ,2008

[Luo02] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton., “Middle-tier database caching for e-business”, in Proc. ACM SIGMOD International Conference on Management of Data, 2002.

[Ami03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan., “DBProxy: A dynamic data cache for Web applications.” in Proc. International Conference on Data Engineering (ICDE) , 2003.

[Cha08] Charles, G., Amit, M., Anastasia, A., Bruce, M., Todd, M., Christopher, O., Anthony, T., “Scalable Query Result Caching four Web Applications”, in Proceedings of the 34th Very Large Databases(VLDB’08), pp.550-561, 2008

[Lar04] Larson, PA., Goldstein, J. and Zhou, J., “MTCache: Transparent mid-tier database caching in sql server.” In Proc. International Conference on Data Engineering(ICDE), pp.177-188, 2004.

[Gupta93]Gupta, A., Mumick, IS. and Subrahmanian, VS., “Maintaining views incrementally.” In Proc. ACM SIGMOD International Conference on Management of Data, pp.157-166, 1993.

[Ken96] Kenneth A. Ross, D. Srivastava, and S. Sudarshan, “Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time.” in Proc. ACM SIGMOD International Conference on Management of Data, pp. 447-458, 1996.

[Stone98]M. Stonebraker, “The case for partial indexes,” in Proceedings of the 34th Very Large Databases (VLDB), 1998

[Wu11] Eugene, Wu., Sam Madden., “Partitioning Techniques for Fine-grained Indexing.” in Proc. International Conference on Data Engineering(ICDE), 2011

[Maru15] C. Maru, M. Enoki, A. Nakano, S. Yamamoto, S. Yamaguchi, and M. Oguchi,” Network Failure Detection System for Traffic Control using Social Information in Large-Scale Disasters”, ITU Kaleidoscope Conference, 2015

[Sakai10] T. Sakaki, M. Okazaki, and Y. Matsuo., “Earthquake shakes Twitter users: real-time event detection by social sensors.” In Proceedings of the 19th international conference on World wide web (WWW), pages 851–860, 2010.

[Cheng10] Cheng. Z, Caverlee. J, and Lee. K, “You are where you tweet: A content-based approach to geo-locating twitter users.” In Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM), 2010.

[Guan11] Z.Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan: “Assessing and Ranking Structural Correlations in Graphs”, In Proceedings of International Conference on Management of Data (SIGMOD), pp. 937-948, 2011.

[Au03] D. Abadi et al. “Aurora: A New Model and Architecture for Data Stream Management.” VLDB Journal, 12(2), 2003.

[Au05] D. Abadi et al. The Design of the Borealis Stream Processing Engine.In CIDR, 2005.

[STREAM04] A. Arasu et al. “STREAM: The Stanford Data Stream Management System.” In Data Stream Management: Processing High-Speed Data Streams, 2004.

[Ni00] J. Chen et al. “NiagaraCQ: A Scalable Continuous Query System for Internet Databases.” In Proc. ACM SIGMOD International Conference on Management of Data, 2000.

[IB] IBM http://www-03.ibm.com/software/products/en/ibm-streams-to-expire

[Mi] Microsoft StreamInsight https://technet.microsoft.com/ja-jp/library/ee362541%28v=sql.111%29.aspx

[SAS] SAS Event Stream Processing Engine http://www.sas.com/en_us/software/data-management/event-stream-processing.html

[St05] M. Stonebraker, U. C¸etintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” In Proc. ACM SIGMOD International Conference on Management of Data, vol. 34, no. 4, pp. 42–47, 2005

[Go03] Golab, L., Ozsu, M.T.. “Issues in data stream management.”, In Proc. ACM SIGMOD International Conference on Management of Data, vRecord 32 (2), 5–14, 2003

[Esper] Esper, http://esper.codehaus.org/

[Jo11] J. Costa, J. Cecilio, P. Martins, and P. Furtado, “Blending OLAP processing with real-time data streams”, Database Systems for Advanced Applications (DASFAA) Lecture Notes in Computer Science Volume 6588, 2011, pp 446-449

[Ro13] R. Derakhshan, A. Sattar, and B. Stantic, “A new operator for efficient stream-relation join processing in data streaming engines” , Proceedings of the 22nd ACM international conference on Conference on information & knowledge management (CIKM), pp. 793-798, 2013

[Spark] Spark Streaming http://spark.apache.org/streaming/

[Ju] Jubatus http://jubat.us/ja/

[SPL] https://developer.ibm.com/streamsdev/2014/05/06/spl-tumbling-windows-explained/, https://developer.ibm.com/streamsdev/2014/08/22/spl-sliding-windows-explained/

[Azure] Azure http://codezine.jp/article/detail/8261

[15] 15 Mind-Blowing Statistics Reveal What Happens on the Internet in a Minute http://www.inc.com/larry-kim/15-mind-blowing-statistics-reveal-what-happens-on-the-internet-in-a-minute.html

[Fb] https://ja-jp.facebook.com/

[CQL] A. Arasu, S. Babu, and J. Widom. “The CQL continuous query language: Semantic foundations and query execution.” Technical report, InfoLab – Stanford University, October 2003

[SP08] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. “SPADE: the System S declarative stream processing engine.” In Proc. ACM SIGMOD International Conference on Management of Data, 2008