DEIM Forum 2009 E2-3

分散メモリキャッシュにおける 遅延許容型メンテナンス手法の検討

三上　啓太†　　鬼塚　真†　　森下　慎次†　　鷲崎　誠司†

† 日本電信電話株式会社　NTT サイバースペース研究所

E-mail: †{mikami.keita,onizuka.makoto,morishita.shinji,suzaki.seiji}@lab.ntt.co.jp

あらまし　近年，Web システムにおける分散メモリキャッシュを利用した DB の負荷軽減が注目されている．しかし，今後のユビキタス社会において進展してゆくと考えられる高頻度なビュー更新環境下においては，既存のキャッシュ利用手法はキャッシュヒット率が低下し，結果としてレスポンスタイムが劣化してしまうという問題があった．そこで本研究では，キャッシュを破棄せずに利用することでキャッシュヒット率を維持すると同時に，データの最新性とのトレードオフを考慮してキャッシュの更新負荷を削減する，効率的なメンテナンス手法を提案する．プロトタイプシステムを用いた評価実験の結果，提案手法を用いることでレスポンスタイムの劣化を防ぐことが可能であり，また遅延件数を適切に設定することで，サーバリソースを安定して無駄なく利用可能であることを示した．

キーワード　Web システム, 分散メモリキャッシュ, メンテナンス

1. はじめに

今日，多数のユーザにサービスを提供する Web システムにおいては，スケールアウトが難しくボトルネックとなりやすい DB における負荷対策が重要となる．中でも分散メモリキャッシュを利用した問い合わせ結果キャッシュは，DB に手を加えることなく実施可能であり，得られる効果も大きいため，高い注目を集めている．

キャッシュ利用手法として最も一般的な手法は，参照時に問い合わせ結果をキャッシュし，DB が更新された際には影響を受けるキャッシュを破棄 (エクスパイア) する手法だが，この手法は DB の更新頻度が高い環境下ではキャッシュヒット率が低下し，結果としてレスポンスタイムの劣化や DB 負荷の増大を招くという問題点がある．

現在，多くの Web システムでは更新に比べ圧倒的に参照の頻度が高いため，エクスパイア方式におけるキャッシュヒット率低下は問題とならないが，今後の社会のユビキタス化に伴う，センサ情報や行動履歴を活用したサービスや，ユーザによるユビキタスな情報発信サービスを実現するためには，現在に比べ遙かに更新頻度の高い環境下でも性能を発揮できるキャッシュ利用技術が必要となる．

そこで，我々は，分散メモリキャッシュ上に DB に対する問い合わせ結果を格納し，DB が更新された際にはキャッシュを破棄することなく，最新の状態にメンテナンスすることで，参照時には常にキャッシュからデータを取得する方式を検討する．この際，キャッシュを利用するサービスとしては，前述のセンサ情報・行動履歴を活用するサービスや，ユーザによる情報発信サービスを想定し，これらの情報が高頻度に追記される DB 上のテーブルに対する，JOIN 処理を含むような問い合わせの結果を，問い合わせのパラメータであるユーザ ID 等を key として分散メモリキャッシュ上に格納するモデルを対象とする．(これによりキャッシュは追記更新可能であるとする)

この方式では，参照時は常に分散メモリキャッシュからデータを取得し，DB に対する問い合わせは行わないため，DB における参照処理による負荷は一切発生せず，また，常にキャッシュを利用した高速なレスポンスと，ハッシュ分散による高いスケーラビリティの恩恵を受ける事ができる．

しかし，このようにキャッシュを常時メンテナンスする方式では，DB に更新があった場合，そのデータが参照されるかどうかに関わらず，分散メモリキャッシュ上のデータのメンテナンスを行うため，キャッシュのメンテナンス処理の負荷は非常に大きい．特に我々が想定する高頻度の更新環境下では，キャッシュメンテナンス処理の負荷が DB の処理能力の限界を超え，キャッシュメンテナンスが DB の更新に追いつかないだけではなく，DB のパフォーマンス低下や，サーバーダウンに繋がる恐れがあるため，キャッシュメンテナンス処理の負荷を削減する必要がある．

そこで本研究では，キャッシュの最新性とのトレードオフにより，キャッシュメンテナンスの負荷を削減する，遅延許容型キャッシュメンテナンス方式を提案する．

さらに，ミニブログサービスで用いられる「フレンドタイムライン処理」を対象として，分散メモリキャッシュ上に問い合わせ結果キャッシュを構築するプロトタイプシステムを実装し，実際のミニブログサービスからクロール

した半年間分，約 110 万件の投稿を利用した評価実験を行った．実験では従来手法と提案手法の，キャッシュヒット率，参照のレスポンスタイム，更新のスループットについて比較を行い提案手法の有効性を確認したほか，提案手法のパラメータである遅延件数を適切に設定することで，データの最新性とトレードオフにメンテナンス負荷を削減可能であることを示した．

2. 課題の詳細化

本研究では，Web システムのバックエンドにおいて，分散メモリキャッシュ上に，DB への問い合わせ結果をキャッシュして利用することにより，DB に対する参照負荷の軽減を行うモデルを対象とする．

また，1. 章で述べた，「高頻度に追記されるデータの JOIN 結果を含む問い合わせ」を利用するサービスの代表的な例として，「フレンドタイムライン処理」を題材とする．

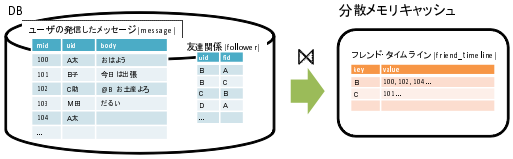

図 1　フレンドタイムライン処理

フレンドタイムライン処理は，SNS やミニブログ等でよく用いられる処理で，全ユーザの発言とユーザ毎の友人関係のテーブルを結合 (JOIN) して，あるユーザの全ての友人の最近数件の発言を集約して表示するためのビューを作成する．

全ユーザの投稿した日記などの記事を表す message テーブルと，ユーザ同士の友人関係を表す follower テーブルを元に，フレンドタイムライン処理結果のビュー friend_timeline を作成する処理をリレーショナル代数によって記述すると，以下のようになる．

フレンドタイムライン処理

$$message(id, user\_id, date)$$
$$follower(follower\_id, followee\_id)$$

$$friend\_timeline(X, follower, message) =$$
$$\sigma_{follower.followee\_id=message.user\_id}($$
$$(\sigma_{follower\_id=X}\, follower) \times message))$$

上記 $friend\_timeline(X, id, author)$ を，パラメータとなるユーザ ID である X を key として，図 1 のように分散メモリキャッシュ上に格納し，ユーザによる参照時には DB における上記処理を行うことなく，キャッシュ上のデータを返却することで，DB への問い合わせによる負荷を軽減し，ユーザの問い合わせに対するレスポンスタイムを短縮することができる．

すなわち本研究では，このビューの作成タイミングやメンテナンスアルゴリズムを工夫することで，

- ユーザに対するシステムのレスポンスタイム
- CPU やキャッシュ等のシステム資源利用効率

の二つの面からの性能向上を目指す．

3. 既存技術

3.1 エクスパイア方式

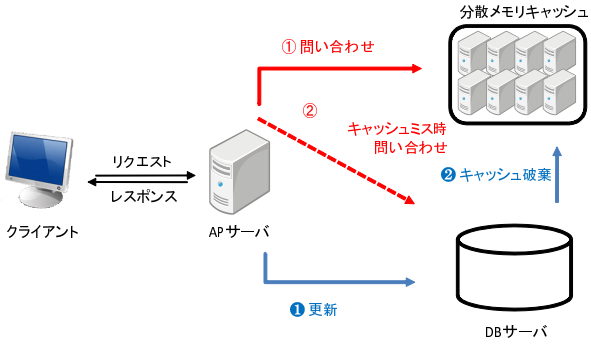

図 2　エクスパイア方式

図 2 は，分散メモリキャッシュの利用方式としては最も一般的なエクスパイア方式の図である．参照時，まずキャッシュに問い合わせを行い，キャッシュにデータが存在しなかった場合 (キャッシュミス時) のみ DB に問い合わせを行うことにより，DB へのアクセス回数を低減し，DB の参照負荷を削減している．エクスパイア方式では，ビューの作成はキャッシュミスした場合に初めて行われ，それ以降同一のビューへの問い合わせはキャッシュのみを利用して処理される．必要とされたビューのみをキャッシュするため，キャッシュスペースの利用効率に優れており，また，こうしてキャッシュされたビューを利用する場合には高速なレスポンスが可能である．しかし，キャッシュミス時には DB に問い合わせを行い，一からビューを作り直すため，レスポンスタイムが悪化してしまう．

また，ビューの元となるデータが更新された場合，キャッシュされているビューをそのまま利用し続けると，古いデータを返してしまうことになるため，当該キャッシュを破棄 (expire) する必要がある．従ってビューの更新頻度が上昇すると，キャッシュのエクスパイアが多発し，結果としてキャッシュヒット率が低下し，レスポンスタイムも悪化してしまうという問題がある．

実験の詳細は 6.2 節にて後述するが，更新頻度とキャッシュヒット率の関係を調べた予備実験の結果を図 3 に示す．

キャッシュヒット率が 90%を超すような領域は，更新比

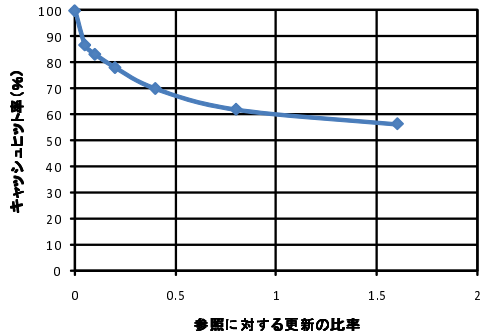

図 3　更新頻度とキャッシュヒット率の関係 (expire)

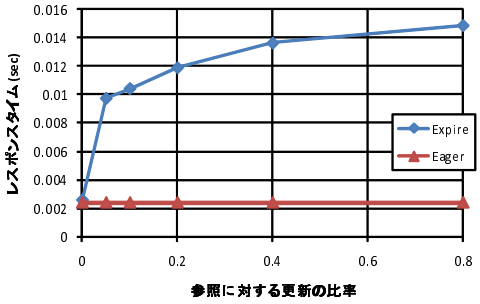

図 5　更新頻度とレスポンスタイムの関係

率の低いごく一部だけであり，また，更新比率が低い領域ではグラフの傾きが急であることから，サービスにおけるデータの更新頻度の増加が，大きくキャッシュヒット率を低下させてしまう可能性がある．

特に，本研究で対象とする，高頻度でビューの更新が発生する環境下では，エクスパイア方式は十分な性能を発揮できないという問題がある．

3.2　常時メンテナンス方式

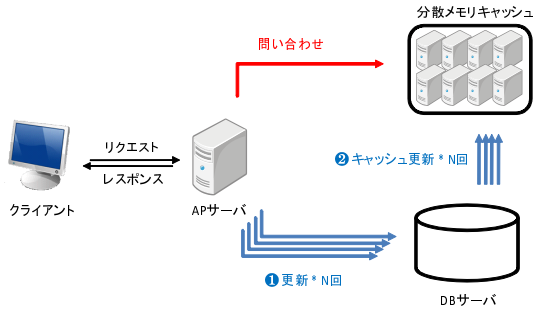

図 4　常時メンテナンス方式

エクスパイア方式の問題点である，高頻度更新環境下でのキャッシュヒット率低下を回避する手法として，DB が更新された時にキャッシュ上のデータを破棄せず，常にメンテナンスし続ける方式が考えられる．(常時キャッシュのメンテナンスを行うことから，この方式を「常時メンテナンス方式」と呼称することにする．)

図 3 同様，実験の詳細は 6.2 節にて後述するが，エクスパイア方式 (Expire) と常時メンテナンス方式 (Eager) における，更新頻度とレスポンスタイムの関係を図 5 に示す．

更新頻度の増加とともに大きくレスポンスタイムが悪化しているエクスパイア方式に対して，常時メンテナンス方式では，ビューは常にキャッシュ上に存在し続けるため，DB の更新頻度に関わらず常に高速なレスポンスが可能となるというメリットがある．

しかし常時メンテナンス方式の場合，DB の更新の回数に比例した回数のキャッシュ更新処理が発生することになり，DB の更新頻度が高い環境では，キャッシュの更新処理がボトルネックとなってしまう．実際，予備実験では，DB 更新のスループットを上昇させていくと，DB サーバの CPU 使用率が 100% になった時点でシステムのスループットが限界に達した．

また，常時メンテナンス方式では，エクスパイア方式と異なり，問い合わせが行われていないビューでも，分散メモリキャッシュ上に格納するため，キャッシュスペースの利用効率が悪いという問題がある．

4.　提 案 手 法

本研究における提案手法である，遅延許容型メンテナンス方式について説明する．

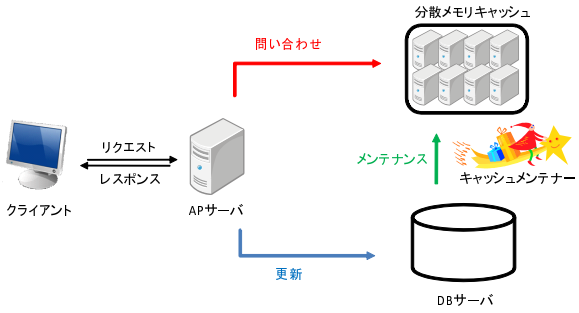

図 6　遅延許容型メンテナンス方式

遅延許容型メンテナンス方式では，図 6 のように，キャッシュのメンテナンス処理を参照や更新処理から切り離し，新たに追加したキャッシュメンテナーが適切なタイミングでキャッシュのメンテナンスを行う．

キャッシュメンテナーは一定期間の更新をまとめて更新命令を作成し，一度にキャッシュに反映することで，更新時の DB 問い合わせ処理及び，分散メモリキャッシュとの通信処理のコストを削減する．

ビューは常にキャッシュ上に存在するため，高速なレスポンスが可能となる．

4.1　見かけ上のキャッシュヒット率の維持

エクスパイア方式において問題となったのが，更新頻度の上昇に伴う，キャッシュヒット率の低下とレスポンスタ

イムの悪化であった．常時メンテナンス方式では，常にキャッシュを最新に保つことでこの問題を解消できるものの，キャッシュの更新処理コストが問題となってしまった．

そこで提案手法では，DB が更新され，キャッシュ上のビューが古くなったとしてもそれを許容し，指定量まで更新処理が蓄積されるまではキャッシュメンテナンスを行わず，キャッシュヒット率及びレスポンスタイムを，常時メンテナンス方式と同じ水準に維持する．この際のキャッシュヒットは，実際は DB の更新により最新のビューでは無い可能性もある，言わば「見かけ上の」キャッシュヒットだが，実際のサービスでは DB が更新されたからといってすぐにそのデータが参照されるとは限らず，データの利用用途によっては多少データの更新の反映が遅れても問題ないと考えられる．例えば，多くのミニブログサービスは Web を介した API を提供しているが，API 経由でミニブログのメッセージを取得するクライアントの多くは，数分おきに API に問い合わせを行う疑似 PUSH 方式を採用しており，書き込みはリアルタイムでもその参照はリアルタイムではなく，1〜2 分の遅れは許容されている．

本手法ではこのように，指定レベルの遅延を許容する代わり，キャッシュヒット率とレスポンスタイムは常時メンテナンス方式の水準を維持する．

4.2 更新処理のコスト削減

遅延許容型メンテナンス方式では，具体的には下記の三つの処理に関して，処理コストの削減を図る．

4.2.1 メンテナンスの度に一定量発生する処理

SQL のパースや分散メモリキャッシュとの接続などの，メンテナンス毎に毎回発生する処理のコストは，純粋にメンテナンスを行う回数に比例して発生する．従って，例えば遅延許容型メンテナンス方式により，N 回分の更新処理をまとめて反映する場合，この部分に属する処理のコストは 1/N に削減することができる．

4.2.2 分散メモリキャッシュの更新処理

提案手法では，一般的な分散メモリキャッシュの利用方式同様，分散メモリキャッシュの一エントリーに，一つのビューを格納する．この場合，追記型の更新などの特別な工夫を行わない限り，ビューの更新を行うためには，一つのビューを丸ごと書き換える必要がある．この場合，一つのビューに対する N 回分の更新処理をまとめて反映する処理コストは，1 回分の更新処理のコストと大きくは変わらない．

従って，遅延期間中に蓄積した更新処理の中の，同じビューに対する更新をひとまとめにして反映することで，同じビューに対する更新の回数を N 回とすると，N-1 回分のビュー更新コスト削減が可能となる．このコスト削減効果は，蓄積する更新処理内にどれだけ同じビューに対する更新が含まれるかに依存するため，対象とするサービスの利用傾向や友人関係の偏りによって変わる．

また，この際，複数のビューから同時にデータを取得する命令 (memcached における get_multi 命令) や，複数のビューの同時更新命令 (memcached には set_multi 命令は存在しない) を活用したコスト削減も可能となる．

4.2.3 ビューの更新内容を決定する処理

「ビューの更新内容を決定する処理」とは，具体的にはユーザによる発言テーブルと友人関係テーブルの JOIN を行い，各ビューの更新内容を決定する処理である．フレンドタイムライン処理

$$friend\_timeline(user, follower, message) = \sigma_{follower.followee\_id=message.user\_id}(\sigma_{follower\_id\in user}follower) \times message)$$

において，message に追記する差分を Δmessage とすると，更新後のビューは

$$friend\_timeline(user, follower, message+\Delta message) = friend\_timeline(user, follower, message) + friend\_timeline(user, follower, \Delta message)$$

と表す事ができ，このうち第一項はメンテナンス前のビューとしてキャッシュ上に格納されているため，この第二項を求め，差分としてキャッシュに追記を行う．

この処理のコストが遅延許容により削減可能かは，メンテナンスするビューの定義によるが，例えばビューの定義がフレンドタイムラインの「最新 20 件」であったとする場合，この 20 件制限による処理の足切りを JOIN 処理の前に行うような実行プランを用いることで，効率良くビューの更新内容を決定できる．

5. CAMEL の設計と実装

4. 章の内容を踏まえ，提案手法の有効性を検証するためのプロトタイプシステムを作成し，既存手法であるエクスパイア方式と常時メンテナンス方式，及び提案手法である遅延許容型メンテナンス方式の比較を行った．

本プロトタイプシステムは，遅延許容型メンテナンスを行うキャッシュメンテナーということで，CAMEL(CAche Maintainer with tolerable dELay) と命名した．

5.1 システムの概観

図 7 に示したように，CAMEL は AP サーバ，DB サーバ，分散メモリキャッシュの 3 層から成る．各層のハードウェア構成を表 1 に示す．

	台数	spec
AP Server	1	Xeon X5470(3.33GHz)×1, 32GB RAM, 1.5TB HDD(RAID 10), ruby 1.8.5, MySQL 5.0
DB Server	1	Xeon X5355(2.66GHz)×2, 16GB RAM, 1.5TB HDD(RAID 10), MySQL 5.0, libmemcached 0.23
Memory Cache	10	Celeron 430(1.8GHz)×1, 2GB RAM, 500GB HDD, memcached 1.2.6

表 1 実験環境

AP サーバは，実際の Web システムにおけるクライアントと AP サーバの役割を模しており，テスト用のアクセ

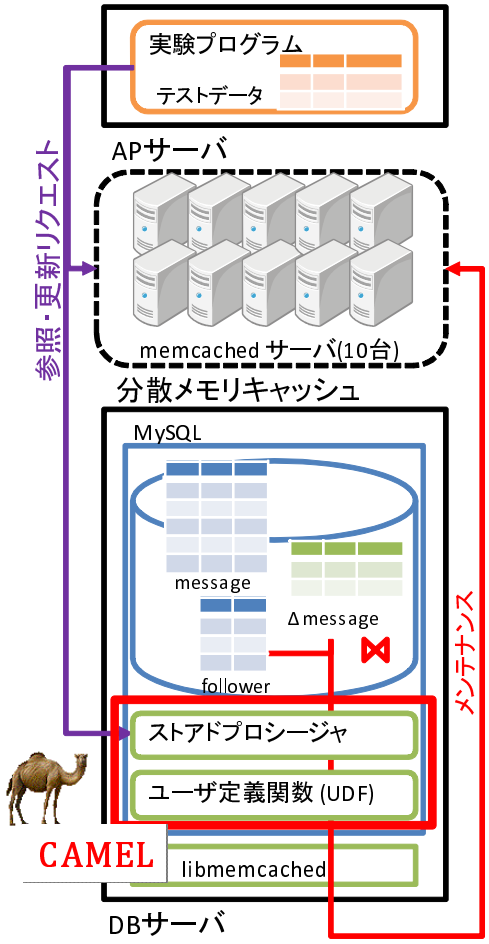

図 7　CAMEL

スパターンのデータを元に，DB サーバに対して参照・更新のリクエストを行う．この実験用のプログラムの詳細については，6. 章で述べる．

分散メモリキャッシュとしては，1 台あたり 1GB のメモリを割り当てた memcached が動作するサーバ 10 台を，容量 10GB の memcached プールとして用い，memcached クライアント libmemcached を介してアクセスする．

DB サーバは，Web システムにおける DB サーバと，キャッシュメンテナーを含む．DBMS としては MySQL を採用し，また，純粋に各手法の性能の比較を行うため，各手法における参照・更新処理およびメンテナーは MySQL のストアドプロシージャとして実装した．この際，MySQL を拡張し，ストアドプロシージャから libmemcached を介して memcached 上のビューを操作・参照可能とするユーザ定義関数を実装した．

評価実験の対象は，フレンドタイムラインの作成処理とし，それにあわせて DB のスキーマと memcached 上のビューを定義した．

5.2　処理の流れ

本節では，CAMEL における，評価する 3 つの方式それぞれの処理の流れを説明する．いずれの方式も，参照時は 1 つのフレンドタイムラインを取得し，更新時は DB の message テーブルに対して 1 件のエントリーを追加する点は共通している．また，以下で説明する処理は MySQL のストアドプロシージャとして記述されているため，AP サーバと DB サーバとの通信は，各参照・更新処理ごとに，ストアドプロシージャをコールする 1 回のみである．

5.2.1　エクスパイア方式

参照時は，まず memcached に対して，求めるフレンドタイムラインが存在するかどうかを問い合わせ，存在した場合はそれを取得して終了する．存在しなかった場合，図??のビュー定義の SQL を実行してフレンドタイムラインを取得し，その後，そのフレンドタイムラインを memcached に格納する．

更新時はまず DB の message テーブルにエントリーを INSERT する．その後 follower テーブルを参照してその INSERT によって影響を受ける memcached 上のビューを特定し，該当するビューを破棄 (Expire) する．この Expire 命令は，ビューの内容を参照せずとにかく破棄の命令を送信するもので，そのうえビューの破棄は，次回データが参照される時か，キャッシュの容量が溢れた時に一括して行われるため，他の処理に比べて実行コストが少ない．

5.2.2　常時メンテナンス方式

参照時は求めるビューは必ず memcached 上に存在するため，memcached に対してのみ問い合わせを行う．

更新時はまず DB の message テーブルにエントリーを INSERT する．その後 follower テーブルを参照してその INSERT によって影響を受ける memcached 上のビューを特定し，該当するビューをメンテナンスする．

5.2.3　遅延許容型メンテナンス方式

参照時は求めるビューは必ず memcached 上に存在するため，memcached に対してのみ問い合わせを行う．

更新時は DB の message テーブル及び dmessage テーブルにエントリーを INSERT する．

こうして dmessage テーブルには新たに追加されたエントリーが蓄積されてゆくが，dmessage テーブルのサイズ，もしくは前回のキャッシュメンテナンスからの一定時間の経過をトリガーとして，dmessage テーブルと follower テーブルを JOIN し，その結果をもとに該当するビューをメンテナンスし，その後，更新の終了した dmessage テーブルをクリアする．

6.　評　　価

6.1　評価用データ

CAMEL を用いて実際に各手法の評価を行うためには，AP サーバが用いる参照・更新のテストデータ，DB サーバが用いる message テーブル及び follower テーブル用のデータを用意する必要がある．これらはできるだけ，想定するサービスに即したデータを用いることが望ましい．本研究では，SNS サービス “goo ホーム” [10] 内のミニブロ

グサービスである “goo ひとこと” を対象としてクローリングを行い，ユーザ同士の友達関係と，各ユーザの投稿した「ひとこと」のデータを取得した．クローリングにより取得したデータの情報を表 2 に示す．

message	1128300 件
follower	70803 件
有効ユーザ ID	16596 人
対象期間	2008 年 5〜11 月

表 2　goo ひとことから取得したデータ

友人関係のデータはそのまま，follower テーブルとして利用した．「ひとこと」データは，そのうちの 100 万件を DB にあらかじめ格納されているデータとして，message テーブルに格納した．また，残りの「ひとこと」データから適宜エントリーを選び出し，更新処理用のテストデータとした．

参照用のテストデータとして用いることができるデータは，システム外部からのクローリングでは得られないため，機械的に生成した．この際，参照用データは，「多くのエントリーを投稿するユーザほど，高い頻度で参照を行う」および「多くのユーザを follow するユーザほど，高い頻度で参照を行う」という二つの仮定を立て，式 (1) で表される Zipf 確率分布を用いて，ある程度偏ったビューにアクセスするようにした．

$$f(k;N) = \frac{1/k}{\sum_{n=1}^{N} 1/n} \tag{1}$$

N : 全ユーザ数,　k : 順位

6.2　更新頻度がエクスパイア方式に及ぼす影響

本研究では，更新頻度の高い環境におけるエクスパイア方式の性能劣化を問題としたが，実際に更新頻度の増加が，エクスパイア方式と常時メンテナンス方式にどの程度の影響を与えるかを調べる予備実験を行った．

エクスパイア方式及び，遅延許容型メンテナンス方式で更新を行う CAMEL に対して，10 分間に 10000 回 (16.6qps) の参照を行い，この 10 分間に行う更新の回数を 0 回から 16000 回まで変化させた時のキャッシュヒット率及びレスポンスタイムを測定した結果を図 8 及び図 9 に示す．

更新頻度が低く，キャッシュヒット率が高い場合，エクスパイア方式もほぼ DB を参照せずに済むため，高い性能を示している．しかし，図 8 からわかるように，更新の比率の増加に対するキャッシュヒット率の低下はかなり敏感であり，図 9 での，キャッシュヒット率の低下に対してのレスポンスタイムの劣化も敏感である．例えば，キャッシュヒット率が 90% を割った時点でレスポンスタイムは約 4 倍に劣化している．

分散メモリキャッシュを採用しているサービスでは 90%

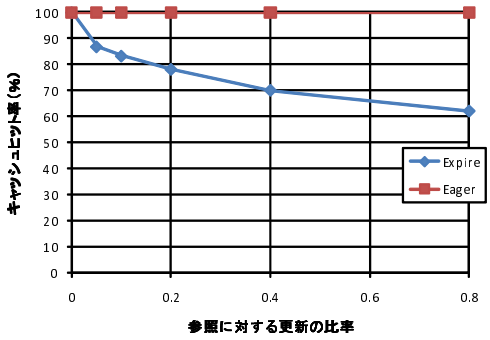

図 8　更新頻度とキャッシュヒット率の関係 (再掲)

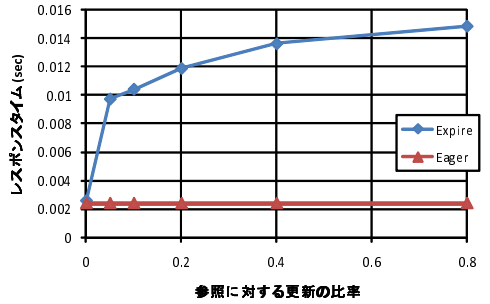

図 9　更新頻度とレスポンスタイムの関係 (再掲)

以上の高いキャッシュヒット率で memcached を運用しているが [4]〜[6]，これは図 8，9 では左端の，キャッシュヒット率が高く，レスポンスタイムも良好な領域であり，この領域での運用であれば従来のエクスパイア方式は非常に優れている．

しかし，1. 章で述べた様に，より更新頻度の高いビューをキャッシュするケースでは，容易にキャッシュヒット率が低下し，それに伴ってレスポンスタイムが劣化する．

現在の goo ホームのトップページの一日の参照件数は 4 万件，サービス開始からの 6ヶ月間での goo ひとことへの投稿件数は約 100 万件なので，大まかに見積もった場合，更新の比率は 0.12 となる．実際はフレンドタイムラインのように更新頻度が高いビューだけでなく，より更新頻度が低く，キャッシュヒット率の高いデータを積極的にキャッシュするため，現在のサービスの特性であればかなり高い (90%以上の) キャッシュヒット率でキャッシュを利用できると考えられる．しかし，例えばサービスが OpenSocial [9] に対応し，Web における行動履歴情報のような，発信頻度の高い情報がアクティビティとして流入した場合，データの更新頻度は数倍〜10 倍以上に増加すると考えられ，この場合はキャッシュヒット率及びレスポンスタイムは，図 8，9 で見て右側の，エクスパイア方式がうまく機能しない領域に入ってしまう．

また，トータルでは高いキャッシュヒット率を維持できても，サービス単位・ビュー単位では更新頻度が高く，

キャッシュヒット率が低下する部分が発生するため，従来からのエクスパイア方式に加えて，部分的に提案手法を併用するハイブリッド方式であれば，極度に更新頻度の高いサービスでなくとも，提案手法を導入するメリットがあると考えられる．

6.3 更新処理のスループット

次に，各方式の更新処理のスループットを測定した．エクスパイア方式 (Expire)，常時メンテナンス方式 (Eager)，遅延許容型メンテナンス方式 (Lazy) の各更新方式で，2 分間高負荷で CAMEL の更新を行った場合の限界スループット (件/s) を表 3 に示す．(遅延許容型メンテナンス方式は，遅延許容により蓄積する件数が 1 件と 10 件の場合)

更新方式	スループット
Expire	105.3
Eager	89.6
Lazy(1)	86.0
Lazy(10)	213.4

表 3　各手法の限界スループット

Expire と Eager 差はもっと大きくなると想定していたが，実際は 15%程度の差であり，想定していたほどの差は見られなかった．これは，更新処理全体から見た場合，同じビューにアクセスして更新を行うか，あるいは破棄するかの違いはあまり大きな違いではないことを示している．遅延件数を 1 件とした Lazy は，処理内容が Eager と同じであるため，スループットもほぼ Eager と等しくなる．遅延件数を 10 件とした Lazy は，Eager，Expire の倍以上のスループットを示した．

6.4 遅延許容によるスループット向上

遅延許容型メンテナンス方式は，一度に処理する件数を多くするほどスループットが向上することが期待される．遅延件数と限界スループットの関係を見るために，6.3 節と同様の方法で，遅延許容型メンテナンス方式の遅延件数を変化させた時の限界スループットを図 10 に示す．また，遅延許容型更新方式における，各処理の負荷の内訳を調べるため，呼び出しても何も行わないストアドプロシージャ (nop) の限界スループットも同時に測定した．

図 10 の Lazy の推移に注目すると，遅延件数が 10 件までとそれ以降とでグラフの傾向が異なることが見て取れる．対数グラフ上で，10 件までは緩やかな傾きで限界スループットが上昇し，以降スループットの伸びが 2500 近辺で止まるまでの区間では，ほぼ直線を描いている．

このような推移になる理由は nop の推移を見るとわかりやすい．遅延が 10 件となった時点で nop はほぼ上限に近いスループットまで達しており，以降は遅延件数が増えても限界スループットは上昇しない．この限界スループットが Lazy の限界スループットと一致しているのは，message テーブルへの INSERT の後はストアドプロシージャの呼び出しのみしか行わない nop のスループットが，更新処理のスループットの限界であるためである．

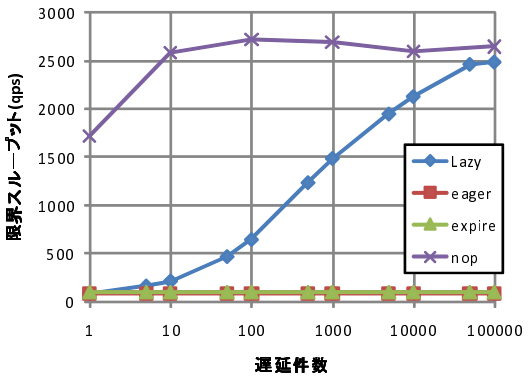

図 10　遅延許容型メンテナンス方式における遅延件数と限界スループットの関係

遅延件数 10 件まではストアドプロシージャ呼び出しまでの処理もある程度の部分を占めているが，遅延件数 100 件以上の領域ではそれらの処理はもうほぼ無視してよく，memcached の更新処理が処理の大部分であることがわかる．以降，この memcached の更新処理部分も無視できる程度となる遅延 10 万件まで，限界スループットは伸び続ける．遅延を許容しさえすれば，DB 側の限界まで分散メモリキャッシュ更新のスループットを向上できることが示された．

最後に，遅延許容型メンテナンス方式を用いた場合，実際にどの程度最新ではない「古い」ビューを参照することになるかを測定した．図 11 は遅延許容型メンテナンスを行う CAMEL に対して，一定期間に参照を 100 万回，更新を 50 万回行い，遅延件数毎に参照時に参照したビューが最新であった比率 (フレッシュネスと呼ぶこととする) をプロットしたものである．

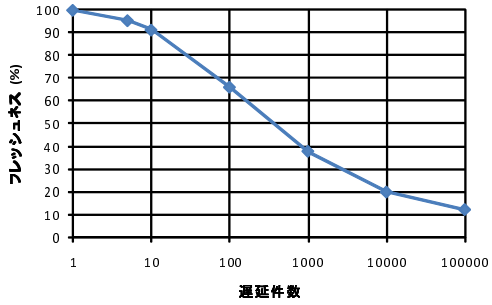

図 11　遅延許容型メンテナンス方式における遅延件数とフレッシュネスの関係

対数グラフにプロットしての結果が直線に近いことから，遅延件数がごく少ない領域では急な傾きを示し，遅延件数が大きくなるにつれて傾きが緩やかになる，いわゆる

べき乗則に従うようなトレードオフ関係が存在していることがわかる．例えば，フレッシュネスを 90%に保とうとするなら，遅延件数は 10 件程度にとどめる必要がある．

しかし，実際にどの程度の遅延が許容されるかは，今回測定したフレッシュネスに加え，最新のビューを取得できなかった場合，そのビューが最新のビューからどの程度の時間遅れたビューであるのかといった，時間的な尺度が重要になると考えられる上，その要求水準もサービス毎に決まるため，この結果のみでの判断は困難である．よりユーザの使用感に立った遅延の評価は今後の課題としたい．

また，分散メモリキャッシュの負荷がボトルネックとなって更新処理が停滞しない限りにおいては，そもそもわざわざフレッシュネスを犠牲にしてまでシステムの更新スループットを伸ばす必要は無い．実際は，システムが安定して動作するスループットを確保できるラインに遅延件数・遅延期間を設定することで，サーバリソースを安定して，無駄なく利用することが可能になると考えられる．

7. 関 連 研 究

DB における実体化ビューのメンテナンスをベーステーブルの更新処理から切り離し，効率的な更新処理を提案している研究としては [1] がある．しかし, [1] ではベーステーブルの更新処理からビューの更新処理を切り離した場合の serializability や snapshot isolation の保障を大きなポイントとし，システムのアイドル時間を利用することで負荷の平準化を図っているのに対し，本研究では Web システムにおける分散メモリキャッシュを対象とし，フレッシュネスとのトレードオフによる更新コスト削減を図っており，そのアプローチと対象領域が異なる．

また，キャッシュを実体化ビューとして扱い，DB を含むシステムのスループットとスケーラビリティの改善を図っている研究としては [2] がある. [2] はアプリケーションからの透過性を重視し，バックエンド DB のサブセットをキャッシュしているのに対し，本研究では JOIN 結果をキャッシュしている点が異なる．また本研究は key-value によるアクセスのみを対象とすることで，分散メモリキャッシュの高いスケーラビリティの活用を図っており，この点で [2] を含む多くの実体化ビューの研究と異なっている．

8. ま と め

高頻度更新環境下においても，キャッシュヒット率の低下による性能劣化が発生せず，また，遅延件数を設定することにより，メンテナンスの処理コストを制御可能なキャッシュ利用方式として，遅延許容型メンテナンス方式を提案した．

提案手法と従来手法を比較し，その性能特性を調べるため，AP サーバ，DB サーバ，に加えて分散メモリキャッシュを利用して，フレンドタイムライン処理を行うプロトタイプシステム CAMEL を設計・実装した．

実データを用いた評価実験により，データの更新頻度が増加した場合，既存技術ではキャッシュヒット率が容易に低下することと，そういった環境においては提案手法を用いることでレスポンスタイムの劣化を防ぐ事ができることを確認した．

また，提案する遅延許容型メンテナンス方式において，遅延の量による，更新スループットとデータのフレッシュネスのトレードオフを確認した．実際のサービスに適用する場合は，負荷の面からどの程度の遅延が必要か，またサービスの面からどの程度の遅延が許容されるかにより適切な遅延の量を設定することで，サーバリソースを安定して，無駄なく利用することが可能になると考えられる．

9. 今後の展望

本資料における評価では “goo ひとこと” のみを対象とした実験を行ったが，今回のデータを元にモデルを構築し，ユーザ規模や友人関係を変化させてのシミュレーションなどを行うことで，今後はより汎用的な評価を行いたい．

一方で，今回の実験を通じて，大規模なシステムを対象としなくても，従来のキャッシュ利用方式に加えて，部分的に提案手法を併用することで，更新頻度の高い一部のビューにおける性能劣化を効率良くカバーできるのではないかという感触を得ることができた．今後は, [3] のように部分的にビューを実体化する手法や，従来手法を適用するビューと提案手法を適用するビューを組み合わせる，ハイブリッド手法についても検討を進めてゆく．

文 献

[1] Jingren Zhou, Per-Ake Larson, Hicham Elmongui. Lazy Maintenance of Materialized Views. In Proc. of VLDB Conference, 2007.

[2] Per-Ake Larson, Jonathan Goldstein, Hongfei Guo, Jingren Zhou. MTCache: Mid-Tier Database Caching for SQL Server. IEEE Data Eng. Bull. 27(2): 35-40, 2004.

[3] Jingren Zhou, Per-Ake Larson, Jonathan Goldstein, Luping Ding. Dynamic Materialized Views. In Proc. of ICDE Conference, 2007.

[4] Bill Howard. Analyzing Online Social Networks. CACM vol.51. no.11., 2008. http://www.acm.org/press-room/news-releases/cacm-reports-0811

[5] 松信嘉範. Facebook のデータセンターに見る MySQL 活用事例. マイコミジャーナル, 2008. http://journal.mycom.co.jp/articles/2008/04/28/mysql/001.html

[6] 長野雅広. memcached in mixi. YAPC::Asia 2008. http://alpha.mixi.co.jp/dist/memcached_in_mixi.pdf

[7] Danga Interactive. memcached. http://www.danga.com/memcached/

[8] Microsoft. Velocity. http://code.msdn.microsoft.com/velocity

[9] Google. OpenSocial resources at Google. 2007. http://code.google.com/apis/opensocial

[10] NTT レゾナント. goo ホーム. http://home.goo.ne.jp/