

分散キャッシュ技術の 大規模システムへの適用事例について

野村総合研究所
共通基盤推進部 主任テクニカルエンジニア

高木 大輔 (たかぎ だいすけ)

実システムにおけるさまざまな課題を解決するため、新技術の調査、アーキテクチャの検討、および、新技術をスムーズに適用するためのフレームワークの設計・開発を行う。現在は、開発生産性を向上させるための研究開発に従事。

1. はじめに
2. 分散キャッシュ技術の概要
3. 実システムへの適用
4. まとめ

要旨

近年、大量のデータを扱う大規模システムでは、データのアクセス性能を向上させるために分散キャッシュ技術が必須のものとなってきている。キャッシュとは頻繁に使われるデータをメモリ内に保持し処理の高速化を図ること、またはそのメモリのことをいう。

従来より、高速なデータアクセスを実現するにはキャッシュ技術が有効であったが、システムで扱うデータ量の増大に伴い、1台のサーバに搭載可能なメモリ量では必要なデータをすべて保持することができなくなってきた。その解決策として、分散キャッシュ技術が注目を集めている。

本稿では、まず、分散キャッシュ技術のメリットや分散キャッシュ製品が備える機能の概要を紹介し、さらに、この技術を実際に数百万ものユーザが利用する大規模システムに適用した際の事例をもとに、設計上考慮すべき点や適用効果について説明する。

※このレポートに記載された会社名、製品・サービス名はそれぞれ各社の商標もしくは登録商標です。

1. はじめに

大量のリクエストを受け付ける大規模システムではデータベース(DB)処理が性能向上のボトルネックとなることが多い。

比較的簡単に台数を増やすことのできるWebサーバに比べ、DBサーバの台数を増やすことは難しく、サーバの台数を増やして性能を向上(スケールアウト)させるのではなく、1台のサーバを増強して性能を向上(スケールアップ)させる必要があるためだ。スケールアップの手法は高コストであり、また、向上させられる性能にも限りがある。

この問題を解決するには、DB処理の負荷を軽減しつつ、スケールアウトで性能向上させられるような仕組みが必要である。

DB処理の負荷を軽減するにはキャッシュ技術が有効だ。DBに問い合わせた結果をメモリ内に保持(キャッシュ)し、同じ問い合わせに対してはキャッシュから結果を返すことにより、DBへのアクセスを減らすことができる。

ただし、大規模なシステムでは扱うデータ量も多いため、1台のサーバに搭載可能な物理メモリ量では必要なデータをすべてキャッシュすることができない。

そこで、複数台のサーバのキャッシュを論理的に統合し、1つの巨大なキャッシュとして扱えるようにする分散キャッシュと呼ばれる技術が注目を集めている。

本論文では、分散キャッシュ技術について

説明するとともに、この技術のある証券会社のオンライントレードシステム(以降、本システムという)に適用した事例について説明する。

2. 分散キャッシュ技術の概要

(1) 分散キャッシュ技術のメリット

分散キャッシュ技術には、通常のキャッシュ技術に比べて次のようなメリットがある。

① キャッシュ可能なデータ量の向上

複数台のサーバを利用するため、1台のサーバに搭載可能な物理メモリのサイズを超えるデータ量を保持することができる。

また、データ量が増加してキャッシュの容量が足りなくなった場合でも、新たにサーバを追加するだけで、容量を増やすことが可能だ。

② 耐障害性の向上

1つのデータを複製して複数台のサーバで保持することにより、サーバが障害でダウンしても、他のサーバからデータを取得して処理を継続することができる。

なお、データを複製した場合、同じデータを重複して保持することになるため、キャッシュに必要なメモリ量はその分増加する。1つのデータを何台のサーバで保持するかは必要なサービスレベル(何台までの同時障害に耐えるようにするか)に応じて決定する。

③ 処理性能の向上

大量のリクエストを複数台のサーバで並列

に処理することができるため、サーバの台数に応じて(スケールアウトで)、システム全体のスループットを向上させることができる。

(2) 分散キャッシュ製品の機能

現在、分散キャッシュ技術を適用するためのミドルウェアとしてさまざまなソフトウェアがあるが、本システムではVMware社(旧GemStone社)のGemFire Enterprise(以降、GemFire)を採用した。この製品をもとにして分散キャッシュ製品が備える機能について説明する。

① データの格納/取得

分散キャッシュはいわゆるKey-Valueストアの一種であり、キーとデータを1対1に関連付けてキャッシュ内のデータを管理している。

キャッシュにデータを格納したり、キャッシュからデータを取得したりするには、まずRegionと呼ばれるキャッシュ領域を生成し、そのRegionに対してキーを指定してデータの書き込み(put)および読み出し(get)を行う。指定するキーおよびキャッシュに格納するデータには、単純な数値や文字列だけでなく、アプリケーションで定義したクラスのオブジェ

クトも使用可能である。

GemFireには、データの物理的な配置方法の違いにより次の3種類のRegionが用意されている(図1)。キャッシュするデータの特性に合わせて適切なRegionを選択することが重要だ。以降、それぞれのRegionについて説明する。

● ノーマルRegion

各サーバでそれぞれ独立したデータを保持するRegionである。このRegionにputしたデータは当該サーバのメモリ内に保持される(これは、分散キャッシュでない通常のキャッシュと同じである)。

データの参照、更新共に同一サーバ内のメモリに対する問い合わせ(以降、ローカルアクセス)となるためパフォーマンスが高いが、単一サーバのみでデータを保持するため耐障害性は低く、サーバ障害時はデータが消失する。また、1台のサーバに搭載可能なメモリ容量以上のデータを保持することはできない。

このRegionはクライアント-サーバ接続時のクライアント側のキャッシュとして利用することが多い。

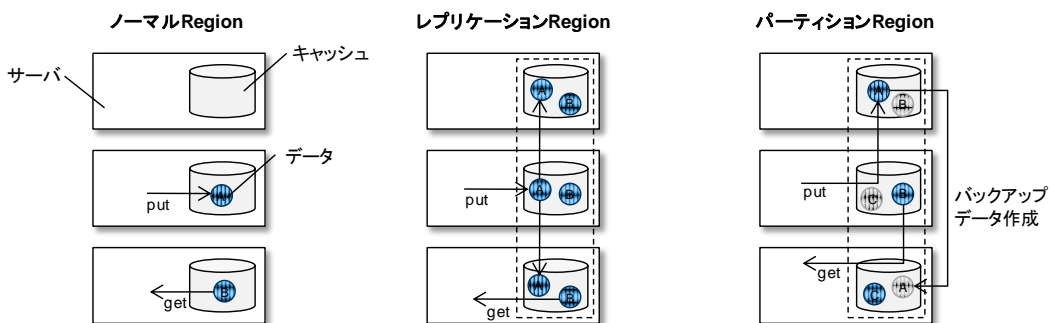


図1 Regionの種類

● レプリケーションRegion

全サーバで同じ内容のデータを保持するRegionである。このRegionにputしたデータは、分散キャッシュを構成するすべてのサーバに複製が生成されて配置される。どれか1台のサーバ上でこのRegionのデータを更新すると、その更新内容がネットワークを介して全サーバに反映される。

データの参照時は、どのサーバから参照してもローカルアクセスとなるためパフォーマンスが高い。反面、更新時は全サーバに対して反映処理を行うためパフォーマンスが低くなる。全サーバで複製を保持するため耐障害性は高いが、必要なメモリ使用量も多く、1台のサーバに搭載可能なメモリ容量以上のデータを保持することはできない。

このRegionには、データ量が少なく、更新頻度に比べて参照頻度が極めて高いデータを格納する。

● パーティションRegion

大量のデータを各サーバで分担して保持するRegionである。このRegionではキーの値に応じて担当するサーバが決まっており、ある

サーバでデータをputすると、そのキーの担当サーバにネットワーク転送されて保持される。

また、putしたデータの複製を作成して他のサーバに配置することにより、耐障害性を高めることも可能である。

目的のデータが配置されているサーバ上でデータの参照/更新を行ったときはローカルアクセスとなるためパフォーマンスが高いが、他のサーバ上に配置されていた場合は、そのサーバに対してネットワークを介して問い合わせ(以降、リモートアクセス)を行う必要があるため、パフォーマンスが低くなる。

サーバを追加することによりキャッシュの容量を拡張できるため、このRegionには大容量のデータを格納する。

各Regionの違いを表1にまとめた。

② データの検索

キーを指定してデータを取得する方法のほか、データの属性値を条件として検索を行う機能が用意されている(クエリ機能)。

検索対象の属性にインデックスを作成することによりクエリの性能を高めることが可能

表1 各Regionの違い

	ノーマルRegion	レプリケーションRegion	パーティションRegion
データの保持	putしたサーバで保持	全サーバで保持	担当サーバおよびバックアップサーバで保持
put時の動作	ローカルアクセス	全サーバへのネットワークアクセス	担当サーバおよびバックアップサーバへのネットワークアクセス (担当サーバ上でputしたときはバックアップサーバへのネットワークアクセス)
get時の動作	ローカルアクセス (他サーバのデータは取得できない)	ローカルアクセス	担当サーバへのネットワークアクセス (担当サーバ上でgetしたときは、ローカルアクセス)
耐障害性	なし	最大	バックアップ数による
容量の拡張性	なし	なし	あり

であるが、インデックスを作成した場合でもキーによるアクセスと比較するとパフォーマンスが大幅に劣るため、クエリを使用する箇所は最小限に抑える必要がある。

③ イベントハンドリング

データの登録、更新、削除などのイベントを契機として独自のロジックを呼び出すことが可能だ。

また、データの参照時にキャッシュ内に目的のデータが存在しなかったとき、独自の処理を呼び出すようにすることも可能であり、その処理でDBや他のキャッシュを参照して目的のデータを生成して返すようにすることができる(生成したデータは自動的にキャッシュに格納される)。

④ データの差分更新

キャッシュ内のデータを更新する際に、更新したデータ全体をネットワーク送受信するのではなく、更新前と更新後の差分情報だけを送受信する機能だ。データを更新するとき、実際に変更されるのはデータの一部分だけであることが多い。データ全体ではなく差分情報だけを送受信することにより、ネットワーク通信量を大幅に抑えることができる。

なお、更新前のデータと更新後のデータの差分情報を抽出する処理や、受信した差分情報を更新前のデータに適用する処理はミドルウェア内で自動的に行われるわけではなく、アプリケーション側で実装する必要がある。

⑤ 不要なキャッシュデータの破棄

一定時間アクセスされなかったデータをキャッシュから自動的に破棄する機能、および、プロセス内のメモリ使用率が一定値を超えたときに、あまりアクセスされていないデータを破棄またはディスクに退避する機能が備わっている。

3. 実システムへの適用

(1) 適用したシステムの特徴

今回、分散キャッシュ技術を適用したのは、株式などの有価証券をインターネット上で売買する証券オンライントレードシステムである。このシステムには以下のような特徴があり、分散キャッシュ技術が高い効果をもたらすことが期待できた。

① 高速なレスポンス、高いピーク性能

オンライントレードシステムでは、有価証券の頻繁な値動きに応じてすばやく取引を行えるかどうかはユーザの利益に直結している。そのため、高速なレスポンスを実現することが重要であり、証券会社間の差別化要因の1つにもなっている。

また、取引所での売買が開始される午前9:00にアクセスが集中し、ピーク時には1秒間に数千ものリクエストを処理する必要があるなど、高いピーク性能が求められる。

このような要件を満たすためには、高速なデータアクセスを実現するキャッシュ技術が有効と考えられた。

② 膨大なユーザ数

このシステムは数百万もの口座を持つオンライントレードシステムであり、また、口座数は日々増加しつづけている。そのため、頻繁にアクセスされるデータだけでも数百GBものサイズになる。当然、1台のサーバに搭載可能なメモリ容量ではすべてのデータをキャッシュすることはできず、分散キャッシュ技術が必要であった。

③ 高い参照比率

ユーザからのアクセスのうち、データの更新を伴う処理(更新系サービス)とデータの参照のみを行う処理(参照系サービス)のリクエスト数の比率を調査したところ、全体の約95%は参照系サービスへのリクエストであった。

参照比率が高いということは、あるデータが更新されてから次に更新されるまでの間に何回も参照されていることになる。このようなシステムにキャッシュ技術を導入すると、一度キャッシュしたデータが何回も参照されることになるため、キャッシュの効果が高いと見込まれた。

④ 一部のサービスに負荷が集中

サービスによってシステムにかかる負荷(処理時間×リクエスト数)に大きな差異があり、全部で130以上のサービスの中で上位の数サービス(いずれも参照系サービス)による負荷だけでシステム全体の負荷の80%以上を占めていた。

一部のサービスでシステム全体の負荷の大部分を占めているということは、それらのサービスに分散キャッシュ技術を適用するだけで高い効果をえられるといえた。

(2) 適用時の考慮点

① 使用できるのはキーによるget/putのみ

基本的に、データアクセス時に使用できるのはキーによるgetまたはputのみである。クエリを使用することも可能であるが、キーによるアクセスに比べてパフォーマンスが大幅に低下するため、できるだけキーによるアクセスだけでロジックを組み立てる必要がある。

キーによるアクセスだけでロジックを組み立てるということは、DBでいえばプライマリキーのみを使用したselect文だけで処理内容を記述するようなものである。実際の業務処理をプライマリキーによる検索のみで実現することは難しいため、データの検索処理をDBのようにストレージ側だけで実行するのではなく、プログラム側でも実行する必要がある。

本システムでは、キャッシュに格納するデータの粒度を大きくし、キーによるアクセスでキャッシュから取得した後で、プログラム内でデータ内の必要な部分を取得するようにした(図2)。

② データをアトミックに更新できない

DBの場合、1トランザクション内で複数のテーブルのレコードを更新しても、トランザク

ション外からは、すべて更新されているかまったく更新されていないかのどちらかの状態しか見えず、更新途中の状態が見えることはない(これをアトミック性という)。

分散キャッシュのようなKey-Valueストアでは、アトミック性が保証されるのは単一データに対する単一の操作(getまたはput)のみである。複数のデータに対する更新はアトミック性が保証されないため、2つのデータをアトミックに更新したいと思っても、処理の途中で外部からアクセスされると、一方のデータだけが更新されているような状態が見えてしまう。

本システムでは、アトミックに更新する必要のあるデータは1つの大きなデータにまとめ、1回のputで更新できるようにした。

③ 単一データでも複数の操作はアトミックに実行できない

アトミック性が保証されるのは単一の操作のみであるため、あるデータをgetし、その値を変更してputし直すというような2つの操作からなる処理をアトミックに行うことはできない。このような処理が複数のクライアントから同時に行われると、最悪の場合、一方の

更新内容がもう一方の更新内容で上書きされてしまう。先程の例と異なり、単一のデータに対する操作でこの問題が発生するため、データの粒度をより大きくしても問題を解決することはできない。

対応策としては、取得したデータが他のクライアントから更新されていない場合のみ更新を許すCAS(Compare-And-Swap)と呼ばれる条件付き更新機能をサポートする分散キャッシュ製品を採用するか、同一データは同時に1つの処理(スレッド)からしか更新されないように制御する必要がある。

本システムでは、キャッシュのデータを更新しているのはDBの更新内容をキャッシュに反映する処理のみであり(後述)、すでに同一データは同時に1スレッドからしか更新しないようにしていた。よってこの問題は発生しなかった。

(3) 適用内容

① 負荷の高い一部のサービスにのみ適用

DBを参照していた処理に分散キャッシュ技術を適用する際は、既存のプログラムを大幅に書き直す必要がある。負荷の低いサービスに分散キャッシュ技術を適用しても適用効果



図2 データの検索処理をプログラム内で実行

はわずかであり、改修コストを上回るメリットを享受できない。

そこで、負荷の高い一部の参照系サービスのみ分散キャッシュ技術を適用することとした。

② DBの更新内容をキャッシュに反映

キャッシュしたデータの元となるDBデータが更新された場合、キャッシュ内のデータも更新する必要がある。そのため、キャッシュ技術を適用する際は既存の参照処理だけでなく更新処理も改修する必要があり、DBのデータを更新すると同時にキャッシュ内のデータも更新するように改修する必要がある。

ただし、DBのデータを更新する処理は更新系サービス、外部システム、バッチ処理など多岐に渡り、これらすべてを改修することは難しい。また、改修しない更新処理が1つでもあった場合、その経路によるデータの更新内容がキャッシュに反映されなくなってしまう。

そこで、個々の更新処理自体を改修するのではなく、DBに対して何らかの更新が行われた際に、その更新内容を検知して自動的にキャッシュに反映する仕組みを新たに構築した。

本システムはオンラインシステムであり、ユーザが行った更新内容が即座に参照できるようにする必要がある。そのため、データの更新処理を行ったトランザクションのコミット後、その内容が数ms～数百ms以内にキャッシュに反映されるようにした。

③ データの特性に応じてキャッシュの種類と配置場所を決定

システム内にはさまざまなデータが存在するが、その中のどのデータをキャッシュし、どのデータはキャッシュしないのか、また、キャッシュするデータを物理的にどのサーバ上に配置するのか決める必要がある。

キャッシュの効果を高めるには、システム内のデータを何でもキャッシュすれば良いわけではなく、データの取得コストが高く、かつ、更新頻度に比べて参照頻度が高いデータのみをキャッシュする必要がある。

取得コストが高いデータというのは、DBのような「重い」ストレージから取得するデータであるとか、複雑なビジネスロジックによって算出するデータなどのことである。ビジネスロジックとは業務データに対するチェックや計算等の業務ルールを実行する処理のことをいう。このようなデータをキャッシュのような「軽い」ストレージに格納することにより、次回以降は低いコストで取得できるようになる。

また、参照頻度が高いデータをキャッシュするのは、一度キャッシュしたデータが何回も参照され、キャッシュの適用効果が高くなるためである。逆に更新頻度が高いデータをキャッシュした場合、キャッシュしたデータがすぐに破棄されて参照回数が少なくなるため、キャッシュの効果は低くなる。

以上のことをふまえ、本システムでは以下の3種類のデータをキャッシュすることとした。

● Domain データ

DBに格納されているデータのうち、口座情報、注文情報など、ユーザに紐づくテーブルのデータだ。

この種類に属するデータは単一の処理で同時に更新されることが多い。そこで、アトミックに更新できるようにするため、複数のテーブルのレコードを口座単位にまとめて1つの大きなデータとし、口座IDをキーとしてキャッシュに格納するようにした(Domainキャッシュ)。

ユーザ数の増加にともなってデータサイズが増加していくため、DomainキャッシュはパーティションRegionとして定義している。

● Master データ

DBに格納されているデータのうち、株の銘柄情報(株価は含まず)など、ユーザに紐づかないテーブルのデータだ。

この種類のデータに関しては、テーブルごとに別々のRegionを用意し、それぞれのテーブルの主キーをキーとしてキャッシュに格納している(Masterキャッシュ)。

様々なビジネスロジックから参照されるため、MasterキャッシュはレプリケーションRegionとして定義している。

● View データ

参照系のビジネスロジックの処理結果であ

り、DomainキャッシュおよびMasterキャッシュから必要なデータを抽出、編集して生成する。

ビジネスロジック内で株価のように更新頻度の高いデータを扱っていた場合、算出される結果の更新頻度も高くなり、キャッシュの効果が低くなってしまふ。キャッシュするデータの更新頻度を下げるため、1つのロジック内の処理を株価に関係する部分と関係しない部分の2つに分離し、株価に関係しない部分の処理結果のみをキャッシュしている(株価に関係する部分の処理は、リクエストごとにキャッシュから株価に関係しない部分の処理結果を取得した後に実行)。

分散キャッシュ技術適用前の旧システムの構成を図3に、適用後の新システムの構成図を図4に示す。また、各キャッシュの設定内容を表2に示す。

Viewデータは画面表示をつかさどるプレゼンテーションロジック(PL)サーバと参照系ビジネスロジック(BL)サーバに配置し、PLサーバではノーマルRegion、参照系BLサーバではパーティションRegionに格納している(Viewキャッシュ)。ユーザからリクエストを受け付けたときは、まず、PLサーバ上のViewキャッシュからデータの取得を試みる。PLサーバ上のViewキャッシュに目的のデータが存在しなかった場合は、参照系BLサーバ上のViewキャッシュからデータを取得する。参照系BLサー

バ上のViewキャッシュにも目的のデータが存在しなかった場合は、参照系サービスのビジネスロジックを呼び出し、DomainキャッシュおよびMasterキャッシュの内容を元にViewデータを生成する。

なお、参照系サービスのビジネスロジックは、取得しようとしたViewデータのキーを担

当するサーバ上で実行される。Viewデータを作成する際にDomainデータおよびMasterデータをローカルアクセスで取得できるようにするため、同じ口座IDのViewデータとDomainデータは同じサーバ上に配置されるようにした（Masterデータは全サーバ上に配置されているため、常にローカルアクセスとなる）。

旧システム

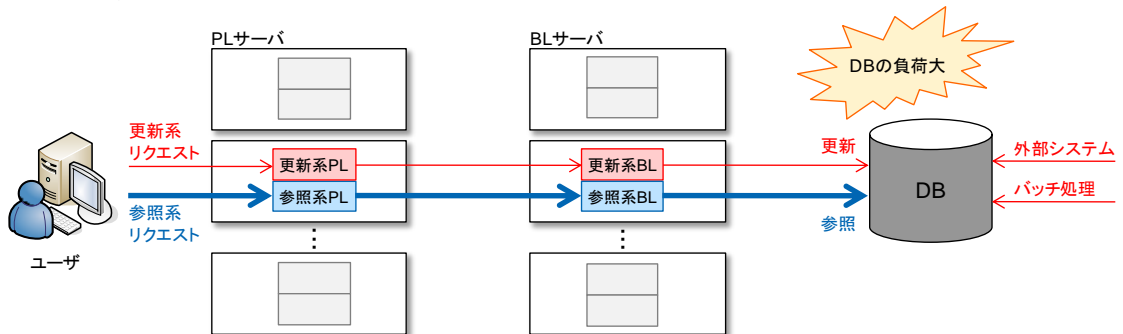


図3 旧システムの概要

新システム

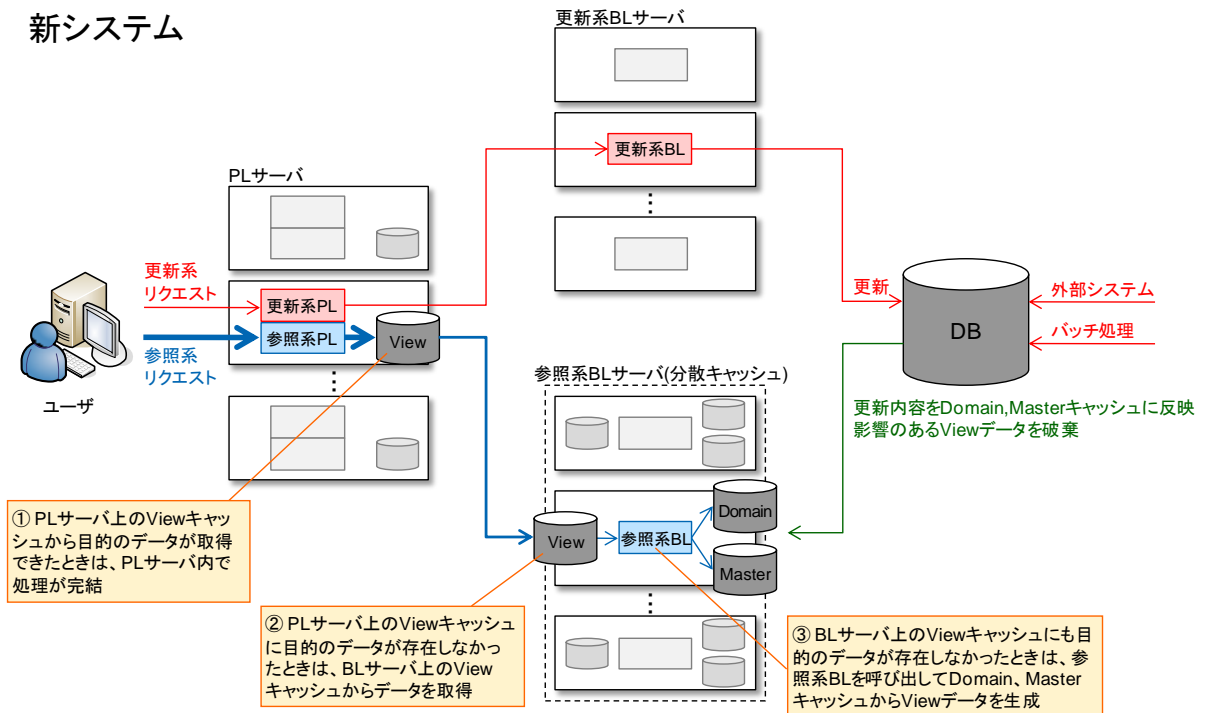


図4 新システムの概要

表2 各キャッシュの設定内容

	View(PL)	View(BL)	Domain	Master
配置先サーバ	PLサーバ	参照系BLサーバ		
Regionの種類	ノーマル	パーティション	パーティション	レプリケーション
バックアップ数	-	0	1	-
一定時間アクセスがなかったとき	破棄	破棄	なにもしない	なにもしない
メモリ使用率が一定値を超えたとき	破棄	破棄	ディスクに退避	なにもしない

(4) 適用結果

① レスポンスタイム

分散キャッシュ技術の適用前(旧システム)と適用後(新システム)の平均レスポンスタイムを示す(図5)。分散キャッシュ技術を適用することにより、レスポンスタイムが大幅に減少したことが分かる。

キャッシュヒット率はほぼ90%以上であることから、分散キャッシュ技術を適用した参照系サービスへのリクエストの90%以上はPLサーバ内で処理が完結し、参照系BLサーバへの問い合わせを行わずにレスポンスを返していることがわかる。

② キャッシュヒット率

PLサーバ上のViewキャッシュのキャッシュヒット率のグラフをに示す(図6)。東証の取引時間である9:00~11:00、12:30~15:00のキ

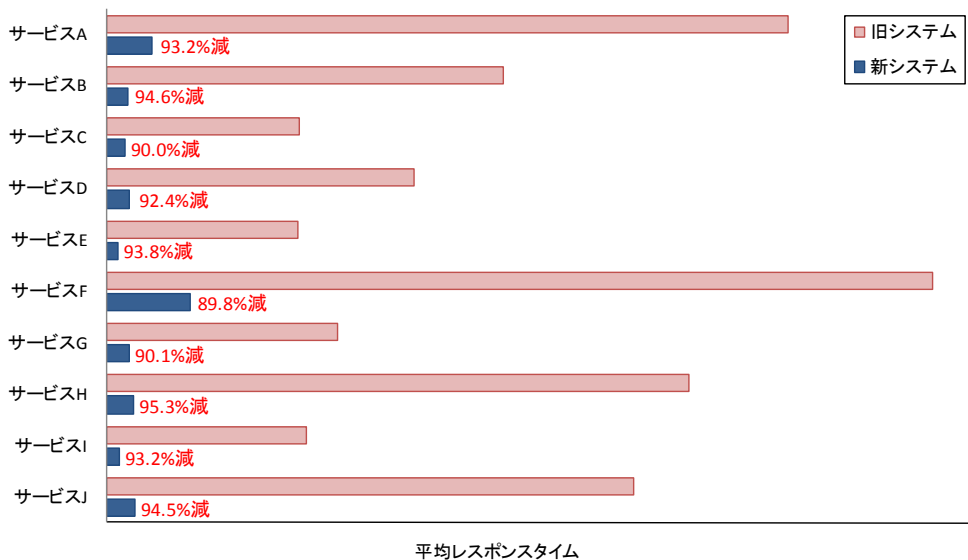


図5 新旧システムの平均レスポンスタイム

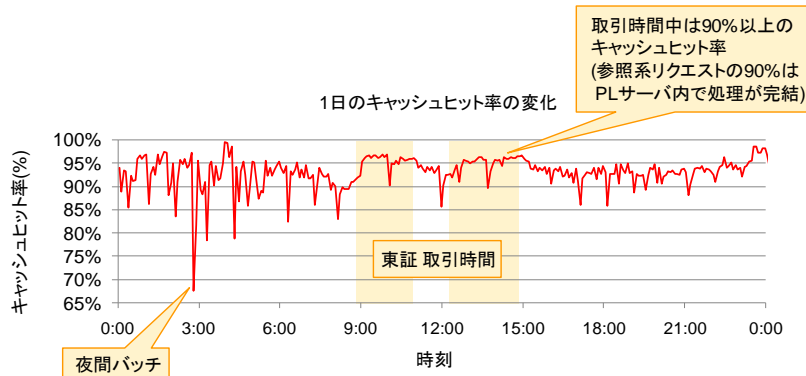


図6 Viewキャッシュのヒット率

4. まとめ

分散キャッシュ技術は、大規模システムで高いパフォーマンスを実現するために必須の技術となってきた。本システムでも分散キャッシュ技術を適用することにより、高いパフォーマンスを実現することができた。

ただし、どのようなシステムでも分散キャッシュ技術が適用できるわけではない。

分散キャッシュ技術には高いパフォーマンスと引き換えにデータアクセス方法やアトミック性に制限があるため、複雑な検索条件でデータを検索する必要があったり、さまざまなデータをアトミックに更新する必要があったりするシステムでは分散キャッシュのメリットを生かすことは難しい。

また、分散キャッシュ技術に適した特性を備えるシステムであっても、分散キャッシュ技術を導入する際には、キャッシュするデータの選定やロジックの書き換えなどの作業が必要になる。単純に分散キャッシュ製品を導入しさえすればシステムのパフォーマンスが

向上するわけではなく、システムの特徴に合わせて分散キャッシュ技術を適用することが重要である。