Vol.2013-DBS-157 No.20
Vol.2013-IFAT-111 No.20
2013/7/23

高可用キー・バリュー ストアにおけるデータ鮮度に基づいた一貫性の提案とその実装

堀井洋[†1]　　小野寺民也[†2]

可用性と分割耐性を重視するキー・バリュー ストア（KVS）の場合，その一貫性は犠牲になる．そのため，従来のKVS では，クライアントが複数のレプリカの値を照会し，最新のデータを選択することで一貫性を保つ．しかし，多くの場合，レプリカは同じデータを返すため，冗長な処理が増えることになる．我々は，この冗長な処理を省略して保障可能な，データ鮮度に基づいた一貫性を提案する．データ鮮度は，レプリカ数と時刻によって表現され，指定した数のレプリカが，ある値を，指定した時刻以降に最新であると認識する場合，そのデータ鮮度は保たれていると認識する．このデータ鮮度を利用する一貫性は，レプリカ間で複製状況を同期することで，各レプリカで保障可能か判定できるため，クライアントが複数のレプリカに冗長な照会処理を要求する必要がなくなる．本一貫性を保障するKVS を実装したところ，従来の KVS と比較し，2 倍以上の性能向上が見込まれることが分かった．

Freshness-based Consistency and its Implementation for High-Available Key-Value Store

HIROSHI HORII[†1]　TAMIYA ONODERA[†2]

Key-value stores that emphasize their availability and partition-tolerance sacrifice consistency, as the CAP theorem states. In existing key-value stores, servers detect inconsistencies with versions of values and clients reconcile the inconsistencies while reading values with versions in multiple servers. However, such solutions reduce the throughput caused by duplicated processing for each read in servers. We have devised a new key-value store that provides freshness of value for how up-to-date the value is. Servers recognize freshness of values by exchanging the latest version numbers of values, and clients get the freshness for every read. Whenever the freshness is enough for clients, the clients read a value of only one server. In our evaluation, our key-value store showed up to 2.0 times higher throughputs for reads and competitive throughputs for writes with existing key-value stores.

1. はじめに

キー・バリュー ストア（KVS）[1,2,3,16,17,18,19]は，クラウド環境で動作するアプリケーションにおける，一般的なデータベースになってきている．KVS のサーバーは，キーとバリューを関連づけてデータを格納し，KVS のクライアントは，連想配列（Map）の get や put といった，キーを指定してバリューを取得・更新する素朴な API を通じて，データにアクセスする．KVS は，キーのハッシュ値に応じてキーとバリューを格納するサーバーを分散することにより，サーバー台数に応じたスループットを得ることが可能であり，キーの種類に応じて更新されたバリューを格納するサーバー（レプリカ）を複数用意することにより，アプリケーションに応じた可用性を得ることが可能である．

KVS のようなデータベースは，格納するデータの一貫性を強めると，その可用性は犠牲になる．例えば，レプリカの 1 つを主サーバー（プライマリ）とし，全てのデータアクセスをプライマリが行うことで, Strong Consistency[4]等, KVS は厳密な一貫性を実現可能である．しかし，プライマリの障害時，障害を検知し，1 つのレプリカを新しいプライマリとして選択する処理中は，KVS の処理を制限する必要がある．特に，障害検知は，高いレベルの障害検知器[5]が必要となるため，10 秒以上の時間（Chubby[5]：12 秒，Dryad[7]：30 秒，GFS[8]：60 秒）が必要となる．

一方，KVS の可用性を重視する場合，その一貫性は犠牲になる．例えば，全レプリカへの同等のデータアクセスを可能とし，障害が疑われるレプリカにアクセス中のクライアントが，即座に他レプリカをアクセスする可能とすることで，レプリカの障害に対する可用性は高めることが可能である[9]．しかし，複数のレプリカで同時にバリューが照会，更新されるため，KVS は厳密な一貫性を提供することはできない[10]．

可用性を重視しながら，処理性能を犠牲にし，一貫性を向上する KVS も存在する．Dynamo[3]や Voldemort[18]では，照会時，クライアントが複数のレプリカのバリューを照会することで一貫性を向上している．しかし，クライアントの各照会要求を複数のレプリカが冗長に処理するため，システム全体の性能が低下する．

本稿では，データ鮮度に基づいた一貫性を提案し，可用性を重視しながら、冗長な照会処理を削減する KVS を紹介する．本稿では，データ鮮度を，データの最新の度合いを示す値と定義する．全レプリカは，更新したバリューのバージョンを交換することで格納するバリューのデータ鮮度を管理し，クライアントは，データ鮮度を指定してバリューの照会を要求する．多くの場合，各レプリカのバリューはデータ鮮度を満たすため，クライアントは照会時に複数

†1 日本アイ・ビー・エム（株）
IBM Japan

Vol.2013-DBS-157 No.20
Vol.2013-IFAT-111 No.20
2013/7/23

のレプリカにバリューを照会する必要はない.

これより，2 章にて既存の KVS の挙動の詳細を示し，3 章にて本稿で提案する一貫性とその実装，4 章にてその評価結果, 5 章にて関連技術を紹介し, 6 章にてまとめを行う.

2. 既存の高可用 KVS の挙動

本章では，可用性を重視する KVS の Voldemort の挙動を示し，その問題点を議論する.

Voldemort では，キーのハッシュ値によって，そのキーとバリューを管理するレプリカの集合が決定される．クライアントがバリューを更新する際は，その集合内の全レプリカのバリューを更新し，クライアントがバリューを照会する際は, その集合内の複数レプリカのバリューを照会する．バリューのバージョンは各レプリカによって管理され，バージョンを比較することで，各レプリカの格納するバリュー，および，クライアントの照会するバリューが最新に保たれる.

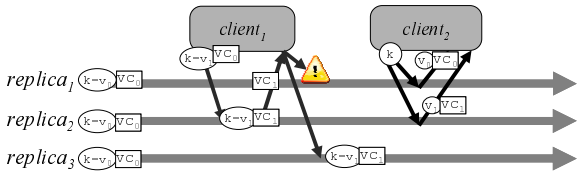

図 1　Voldemort における更新，照会処理（W=2，R=2）

Figure 1　Workflow for read and write in Voldemort

図 1 は，キーk を管理する 3 台のレプリカ $replica_1$, $replica_2$, $replica_3$ に対するクライアント $client_1$ の更新処理と $client_2$ の照会処理を表す.

k のバリューの初期値は，バージョン vc_0 のバリュー v_0 であり，$client_1$ は，バリュー v_1 に更新する．その際，$client_1$ は，まず，1 台のレプリカ（$replica_2$）に vc_0 と v_1 を送信する. $replica_2$ は, 受信したバージョン（vc_0）が，格納するバージョン（vc_0）以上であれば，バージョンを更新（vc_1）し，バリューを更新（v_1）して，更新後のバージョン（vc_1）を返す．その後，$client_1$ は，他のレプリカ（$replica_1$, $replica_3$）に vc_1 と v_1 を送信し，設定された台数（W）以上のレプリカに送信した時点で，更新処理が成功したとみなす．図 1 では，$replica_1$ への送信が失敗しているが，2 台のレプリカへの送信が成功しているため，k の v_0 から v_1 への更新は成功している.

$client_2$ が k のバリューを照会する際は，指定された台数（R）のレプリカの k のバリューとバージョンを照会し，その結果から最新の値を照会する．図 1 では，$client_2$ は，$replica_1$ より v_0 と vc_0 を，$replica_2$ より v_1 と vc_1 を照会しているが，vc_1 は vc_0 よりも新しいため，v_1 を最新の値といて照会する.

このようなシステムでは，R と W の和がレプリカの台数（N）よりも大きければ，照会する値が必ず，クライアントが照会要求した時点における最新の値，もしくは，それ以降に更新された値であることが保障される[11].

Voldemort では, バージョンは Lamport Clock[12]が利用されており，バージョン間は半順序性が保たれている．そのため，バリュー更新時のバージョン比較の際，レプリカはバージョンの順序を決定できない場合がある．その場合，Voldemort では, バージョンの順序を決定できない全てのバリューを保存しておき、クライアントがそのキーを指定した照会をした際は、全てのバリューをクライアントに返す。

クライアントは、上記のように順序性が決定できないバリューを、1 つのレプリカ、および、複数のレプリカから取得する可能性がある。その場合、クライアントは、アプリケーションに対し、それらのバリューから最新のバリューを生成することを要求する．この仕組みは，リードリペア（Read-Repair）と呼ばれる.

このように，R と W の和を N より大きく設定し，また，Read-Repair の実装をアプリケーションに求めることで，Voldemort は，格納する，および，照会するデータの一貫性を保つ．Voldemort は Dynamo のオープンソース版であり，Dynamo も同様の処理が行われている.

しかし，Voldemort の照会処理では，クライアントは必ず R 台のバリューを照会する．例えば，図 1 では，$client_2$ は，$replica_1$ と $replica_2$ のバリューを照会する．ここでは，異なるバリューを照会しているが，もし $client_2$ が $replica_2$, $replica_3$ のバリューを照会した場合は，レプリカは冗長な処理をしていることになる.

一般的に，データベースの処理の大半は照会処理であり, 長期間更新されないバリューを照会することは多い．そのような照会処理に対しては，本質的には，1 台のレプリカからの照会で十分であるため, Voldemort は一貫性の維持に対する多大なオーバヘッドを払っているといえる.

3. データ鮮度に基づいた一貫性と KVS

本章では，データ鮮度を定義し，データ鮮度を用いた KVS の実装を示す.

3.1 データ鮮度

Voldemort で照会可能なバリューは, クライアントの照会要求時点で最新であることを保障するが，クライアントがバリューを受信した時点で最新の値であることは保障しない．例えば，図 1 では，$client_2$ が $replica_2$ から v_1 を受信した時点で, 他クライアントが $replica_2$ の v_1 を更新している可能性がある. すなわち, Voldemort のような KVS を利用するアプリケーションは，照会するバリューが最新ではないことを前提に，記述される必要がある.

一方，アプリケーションは，古すぎるバリューを照会す

Vol.2013-DBS-157 No.20
Vol.2013-IFAT-111 No.20
2013/7/23

ることは許容されない．例えば，Bulletin Board のようなアプリケーションにおいて，5 秒前に更新されたコメントは表示されなくても許容されるが，1 時間前に更新されたコメントは表示される許容されない．つまり，アプリケーションには，許容するバリューの古さの度合いが，潜在的に存在する．

我々は，アプリケーションが許容するバリューの古さの度合いを,「データ鮮度」として指定する一貫性を提案する．データ鮮度は，レプリカ数 r と時刻 t を用いて，以下のように定義する．

定義 データ鮮度 [r,t]

あるキーk のバリューが N 台のレプリカにおいて共有されるシステムにおいて，r（≦N）台のレプリカが時刻 t 以降に，k のバリューとして v が最新であると認識した場合，v はデータ鮮度[r,t] である．

このデータ鮮度を，クライアントが照会処理ごとに指定し，その要求を満たすバリューを取得する．具体例を図 2 に挙げる．

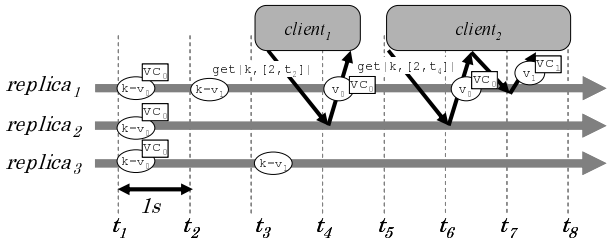

図 2　データ鮮度を用いた照会処理
Figure 2　Read operation with freshness-limit

図 2 は，キーk を管理する 3 台のレプリカ $replica_1$, $replica_2$, $replica_3$ に対するクライアント $client_1$, $client_2$ の照会処理を表す．k のバリュー初期値 v_0 は，$replica_1$ では時刻 t_2 に，$replica_3$ では時刻 t_3 に，v_1 に更新される．

$client_1$ は，データ鮮度$[2,t_2]$を指定して，$replica_2$ のバリューv_0 を取得する．v_0 は，t_2 では，$replica_2$, $replica_3$ において最新である．そのため，v_0 は，$[2,t_2]$ を満たしているため，$client_1$ は，v_0 を照会する．

一方，$client_2$ は，データ鮮度 $[2,t_4]$ を指定して，$replica_2$ のバリューv_0 を取得する．v_0 は，t_4 では，$replica_2$ のみにおいて最新である．そのため，v_0 は，$[2,t_4]$ を満たしておらず，$client_2$ は別のレプリカの値も取得し，要求を満たす値を照会する必要がある．

3.2 時刻の同期とデータ鮮度の保障

データ鮮度を保障するには，各レプリカは他レプリカのバリューの状態を認識する必要がある．例えば，図 2 では，$client_2$ が，照会した v_0 は$[2,t_2]$を満たすかを判定する．その際，$replica_2$ 以外のレプリカが t_2 以降に更新されているか，$replica_2$ は認識する必要がある．

取得したバリューが，指定したデータ鮮度を満たすか判定する仕組みは，レプリカ間で更新されたキーのバージョンを定期的に同期することで，実現可能である．図 3 に，その実装例を示す．

図 3 は，キーk を管理する 2 台のレプリカ $replica_1$, $replica_2$ において，$replica_1$ で更新されたバージョン vc_1 を $replica_2$ と同期する処理する処理を示す．

レプリカは，定期的に他レプリカの各キーに対する最新バージョンを取得する．その際，レプリカは，バージョンと共に，他レプリカがそのバージョンに更新した時刻からの経過時間も取得する．図 3 の場合，時刻 t_1 に，$replica_2$ が $replica_1$ の最新のバージョンを取得要求する．時刻 t_2 に取得要求を受け取った $replica_1$ は，vc_1 と vc_1 に更新した時刻（t_0）からの経過時間 lag_1 を返す．この際，lag_1 は，$t_2 - t_0$ となる．

最新のバージョンと，そのバージョンに更新した時刻からの経過時間を取得したレプリカは，他レプリカの更新時刻を，保守的に認識する．図 3 の場合，$replica_2$ は，「$replica_1$ が時刻 t_1-lag_1 以降に vc_1 に更新した」という更新状況を認識し，t_1-lag_1 を $replica_1$ における更新時刻として認識する．なお，t_1-lag_1 は，通信時間のギャップから，実際の更新時刻 t_0 よりは以前の時刻となる．

クライアントは，バリューをレプリカから取得する際，そのレプリカが認識する，他レプリカの更新時刻からの経過時間も取得する．図 3 では，client が k のバリューを $replica_2$ から取得する際，$replica_1$ が vc_1 に更新した状況 lag_2 も取得する．lag_2 は，t_1-lag_1 から，$replica_2$ が取得要求を受信した時刻 t_4 の経過時間であり，$t_4-(t_1-lag_1)$ である．なお，レプリカ台数分，クライアントは経過時間を取得する．

クライアントは取得した経過時間から，取得した値が要求したデータ鮮度が満たされているか，保守的に判定する．図 3 では，client は，「$replica_2$ は時刻 t_3 に，$replica_1$ は時刻 t_3-lag_2 に，バリューは vc_1 に更新された」として判定する．この情報は，言い換えると，「$replica_2$ は時刻 t_3 以降に，$replica_1$ は時刻 t_3-lag_2 以降に，vc_1 以外のバリューに更新された可能性がある」と認識される．つまり，client が，$[2,t]$ の鮮度を要求する場合，t が t_3-lag_2 以降であれば，取得したバリューはデータ鮮度を保障できないと認識する．

クライアントが要求するデータ鮮度のバリューを取得できなかった場合は，Dynamo や Voldemort と同様，クライアントは他レプリカのバリューを照会する．なお，この取得はデータ鮮度を保障しない．

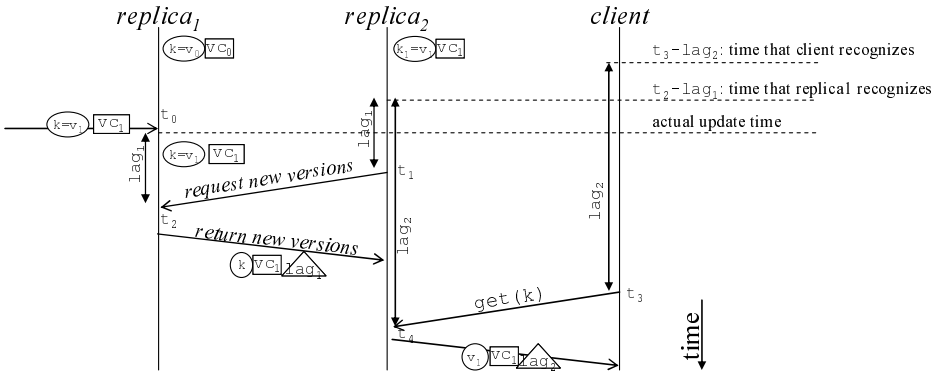

図 3　データ鮮度を保障するための同期処理と照会処理
Figure 3　Version synchronization and read operation to guarantee freshness-limit

4. 評価

提案するデータ鮮度に基づいた一貫性を保障する KVS を，Yahoo! Cloud Serving Benchmark（YCSB）で評価し，その結果より提案手法の特性を議論する. なお, 本評価では，可用性の高い KVS の一貫性を保障するためのオーバヘッドを評価することを目的としているため，全てのレプリカが全てのデータを保有する設定で行われる.

4.1 提案手法の実装

我々は，3 章で示したデータ鮮度を保障する仕組みを備えた KVS を実装した．本 KVS では，全てのキー，バリュー，バージョンはメモリ上に保存され，クライアントは素朴な連想配列（Map）の API，get，put を通じて，キーに関連付けられたバリュー取得，更新する．また，get にはデータ鮮度を指定可能とし，1 台のレプリカで値を取得可能にするための条件として利用する.

4.2 比較する KVS

本評価では，Voldemort を，提案手法の比較対象の KVS とした.

Voldemort は，Dynamo を模したオープンソースの KVS で，LinkedIn で採用されている．キー，バリュー，バージョンの保存場所は，メモリの他にも，RDB や Berkley DB など，多岐にわたる．2 章で示したとおり，更新時には全レプリカに，照会時には複数のレプリカにアクセスし，各レプリカは独立して更新，照会要求を処理可能である．Read-Repair 用のコールバック関数は Map ごとに管理され，アプリケーションは Map の定義時に指定する. データはキーの値により分割され，分割ごとに管理するレプリカの集合が決定される.

本評価では，データの保存場所としてメモリを指定し，全体で 1 つのパーティションを利用することで，全レプリカが全データを保有する形式で評価を進めた.

4.3 Yahoo! Cloud Serving Benchmark

本評価では，Yahoo! Cloud Serving Benchmark（YCSB）を利用して，提案手法の効果を測定した.

YCSB は，Yahoo!が作成した，クラウド上のストレージシステムの性能を計測するためのベンチマークである．YCSB では，仮想的なクライアントが動作し，設定された割合で，照会，および，更新処理を行う．ストレージ内には，設定された数のデータが事前に挿入されており，クライアントはそのデータに対する照会，更新処理を行う．あらかじめ規定のワークロード a) update-heavy, b) read-mostly, c) read-only, d) read-latest, e) short-ranges, f) read-modify-write が存在し，それぞれの照会と更新の割合は，50:50，95:5，100:0，95:5，95:5 となっている．なお，read-latest の更新は挿入であり，short-ranges の照会は範囲指定で複数のデータが照会され，read-modify-write は照会後に同じ値を更新する．本評価で利用する KVS では，範囲指定の照会機能がないため，ワークロード a，b，c，d，f を評価対象とした．また，全ての処理が更新処理（0:100）の w) write-heavy も，併せて評価した．なお，zipfian を用いて，アクセスするキーを選択するよう設定した.

4.4 測定環境

本評価では，レプリカを 4 台の，クライアントを 1 台の物理的なサーバーで稼動させ，評価した．各レプリカは全て同じ構成であり, 4 コアの POWER6（64-bit 4.0-GHz 2core POWER6 を 2 ソケット）と 12GB のメモリ，Red Hat Enterprise Linux Server release 6.4 を OS として利用した．クライアントには, 16 コアの Xeon（64-bit 8-core Xeon E5-2680 2.7-GHz を 2 ソケット）と 32 GB のメモリ， Red Hat Enterprise Linux Server release 6.3 を OS として利用した．全てのサーバー間は，1Gbps のイーサネットで接続した.

Vol.2013-DBS-157 No.20
Vol.2013-IFAT-111 No.20
2013/7/23

4.5 評価結果

図 4 は，2 台のレプリカを利用した際の，Voldemort と提案する KVS（提案手法）のスループットを示す．スループットは 5 回測定し，その平均を測定した．両 KVS 共，R=1，W=2 と設定した．また，提案手法では，1 台のレプリカと，クライアントが照会を要求した時刻から 5 秒前の時刻をデータ鮮度と設定した．

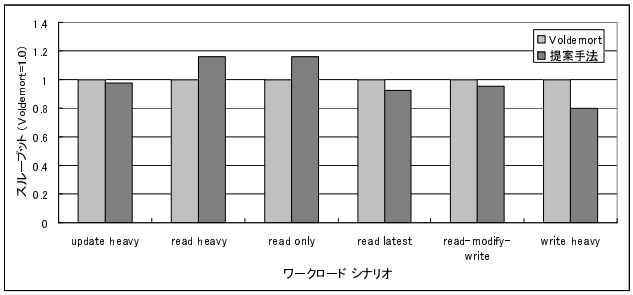

図 4　レプリカ 2 台を利用した YCSB のスループット比較
Figure 4　Throughput of YCSB with two replicas

R=1 では，Voldemort も提案手法も，照会時，クライアントは 1 台のレプリカのバリューを取得する．そのため，提案手法の優位性はない．そのため，照会の多いワークロード（read heavy，read only）においても，提案手法は 20%以下のスループット向上にとどまっている（なお，本スループットの違いは，一貫性の保障方式ではなく，照会処理の実装が原因と考えられる）．

一方，更新が多いワークロード（write heavy）の場合，提案手法は 20%程度のスループット低下が見られた．提案手法では，更新されたバージョンを定期的にレプリカ間で交換する．つまり，更新が少ない場合は，レプリカ間でバージョンを送信しあうことは少ないが，更新が多くなると，バージョンの交換はオーバヘッドになる．

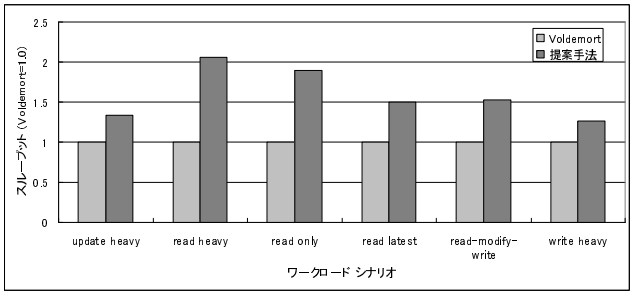

図 5　レプリカ 4 台を利用した YCSB のスループット比較
Figure 5　Throughput of YCSB with four replicas

図 5 は，4 台のレプリカを利用した際の，Voldemort と提案手法のスループットを示す．スループットは 5 回測定し，その平均を測定した．両 KVS 共，R=2，W=3 と設定した．また，提案手法では，1 台のレプリカと，クライアントが照会を要求した時刻から 5 秒前の時刻をデータ鮮度と設定した．

R=2 では，Voldemort は，照会時，クライアントは 2 台のレプリカのバリューを取得する．一方，提案手法では，データ鮮度が守られる限り，クライアントは 1 台のレプリカのバリューを取得する．本評価では，レプリカは 5 秒程度のデータ鮮度を保障するため，ほぼ全て（99%以上）の照会が，1 台のレプリカのみで処理された．つまり，照会の多いワークロード（read heavy，read only）では，提案手法が 2 倍程度のスループットを示した．また，一方，更新が多いワークロードにおいても，提案手法は 20%以上のスループット向上が見られた．これは，write only 以外は 5%の照会処理が影響していることも考えられるが，衝突が起きた際の処理の最適化が，提案手法の方が進められていることが原因と考えられる．

4.6 議論

本提案手法は，図 5 に示すとおり，照会の割合の多いワークロードに対して，2 倍以上のスループットを示すことがわかった．本提案手法は，Voldemort や Dynamo のような可用性を重視する KVS に付加的に追加できる機能であるため，5 秒程度の更新遅延を許すアプリケーションには有効である．

一方，直近のデータ鮮度を求めるアプリケーションや，システムが高負荷時の場合，1 台のレプリカのバリューだけでは，要求するデータ鮮度を保障できない場合がある．その場合，クライアントは複数のレプリカの値を照会することになり，結果的に，提案手法の優位性は薄れる．しかし，システムの負荷は，クラウド環境ではサーバーを追加することで動的に制御されるため，照会処理に数秒以上かかる負荷が常にかかることは考えづらい．また，Web アプリケーションのように，エンドユーザーとのレスポンスに数秒かかることが想定されるシステムでは，直近のデータ鮮度を求める必要はないことが考えられる．

5. 参考文献

Dynamo[3]や Voldemort[18]は，可用性を重視する KVS として，Amazon，Linkedin で利用されている．キーによってレプリカの集合が特定され，集合内の全てのレプリカがバリューの照会，更新を独立して処理する．2 章で示したように，これらの KVS は，一貫性保持のため，クライアントは，1 回の照会で複数のレプリカのバリューを取得する．しかし，多くの場合，クライアントは同じ値を取得するため，レプリカは冗長な処理を行うことになる．本提案手法で示したデータ鮮度に基づく照会処理を利用することで，レプリカの冗長な処理を避けることが可能となり，システム全体のスループットを向上させることが可能となる．

BigTable [1]は，Google が Web Index や様々な Google ア

プリケーションが利用するKVSとして，広く知られている．BigTableでは，複数のキーとそのバリューがタブレットとして構成され，クライアントはキーからタブレットを指定して更新する．タブレットは複数のレプリカで管理され，分散ロック機構 Chubby[6]を用いて，排他的にタブレットをアクセス可能なレプリカが1台選出される．選出されたサーバーに障害が発生した場合は，新しいサーバーを選出するが，障害検知には数秒かかるため，その間タブレットへのアクセスはブロックされ，可用性が低下する．

Cassandra [2]も，Facebookで利用されていたKVSとして，広く知られている．Cassandraはデータモデル BigTableに模した実装とされているが，DynamoやVoldemortと同様，各レプリカが独立して管理するバリューの照会，更新を行うことが可能である．全クライアントは時刻を同期しているものと想定されており[20]，更新された全バリューにはクライアントが時刻を付与し，同時に更新が起こった場合は最後に更新したバリューのみを，各レプリカが保存する．Cassandraは，提案手法やDynamo，Voldemortと似た構成を持つが，一貫性の自由度は低い．また，照会方法は1つのレプリカから照会する方法，Quorum を利用して照会する方法が選択可能だが，前者の場合は古すぎる値を照会する可能性があり，後者の場合はDynamo，Voldemortと同じ問題を含む．

HBase [19]は，よく利用されるKVSの1つで，Hadoopをベースに機能する．キーごとにプライマリサーバーが決定され，排他的な照会，更新処理が可能となる．しかしながら，障害検知には時間がかかるため，プライマリサーバーが障害児の可用性は低い．

データベースの複製手法[13,14,15]とし，データ鮮度と同様の一貫性の提案は存在する．しかし，これらのデータベースは，プライマリサーバーを前提とし，レプリカからの照会可能な条件をデータ鮮度で判定するために利用しており，本提案手法のように，プライマリサーバーを前提にしないシステム構成では利用できない．

6. 結論

本稿では，可用性を重視するKVSとして，全てのレプリカが同一のキーに対するバリューを，同時に照会，更新可能なKVSの処理手法を示した．さらに，クライアントがバリューを照会時に，バリューの最新度合いを表すデータ鮮度を指定する照会手法を示し，レプリカがそのデータ鮮度を保障するバリューを返答した場合，クライアントがその他レプリカのバリューを照会する必要のない照会処理手法を提案した．本提案手法を，Yahoo! Cloud Service Benchmarksで評価したところ，従来手法と比較し，最大2倍以上の性能向上が確認された．

参考文献

1) Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M. , Chandra, T., Fikes, A., Gruber, R. E.: Bigtable: a Distributed Storage System for Structured Data. in Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation, Vol. 7, Nov. 2006.
2) Lakshman, A., Malik, P., Cassandra: a Decentralized Structured Storage System, ACM SIGOPS Operating Systems Review, Vol.44 No.2, Apr 2010.
3) DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. in Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'07,) 14–17 Oct. 2007.
4) Cao, P., Liu, C.: Maintaining strong cache consistency in the World Wide Web. IEEE Transactions on Computers Vol. 47, No. 4, pp. 445-457, 1998.
5) Chandra, T. D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, Vol. 43, No. 2, pp. 225-267, Mar. 1996.
6) Burrows, M.: "The Chubby Lock Service for Loosely-Coupled Distributed Systems," In Symposium on Operating Systems Design and Implementation (OSDI), pages 335–350, Dec. 2006.
7) Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," In European Conference on Computer Systems (EuroSys), pages 59–72, Mar. 2007.
8) Ghemawat, S., Gobioff, H., and Leung, S.T.: "The Google file system," In ACM Symposium on Operating Systems Principles (SOSP 2003), pages 29–43, Oct. 2003.
9) Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding Replication in Databases and Distributed Systems. Proceedings of 20th International Conference of Distributed Computing Systems (ICDCS '00), 2000.
10) Saito, Y. and Shapiro, M.: "Optimistic Replication" in ACM Computing Surveys Vol. 37, No. 1, pp. 42–81, 2005.
11) Gifford , D. K.: "Weighted Voting for Replication" in Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 1979), 1979.
12) Lamport, L.: "Time, Clocks, the Ordering of Events in a Distributed System," ACM Communications, Vol. 21, No. 7, pp. 558-565, 1978.
13) Schneider, F. B.: Implementing fault-tolerant services using the state machine approach. A tutorial. ACM Computing Surveys, Vol. 22, No. 4, pp. 299–319, Dec. 1990.
14) Rohm, U., Bohm, K., Schek, H.-J., Schuldt, H.: FAS - a Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. in Proceedings of the 28th Very Large Data Bases, pp. 754-765, Aug. 2002.
15) Guo, H., Larson, P.-A., Ramakrishnan, R., Goldstein, J.: Relaxed Currency and Consistency: How to say "Good Enough" in SQL. in Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD2004,), pp. 815-826, Jun. 2004.
16) IBM WebSphere eXtreme Scale: http://www-01.ibm.com/software/webservers/appserv/extremescale/
17) Oracle Coherence: http://www.oracle.com/technetwork/middleware/coherence/index.html
18) Project Voldemort: http://project-voldemort.com/
19) Apache HBase: http://hbase.apache.org/
20) DataModel in Cassandra Wiki: http://wiki.apache.org/cassandra/DataModel
21) ServiceGuard Fifteenth Edition (Cluster Configuration Planning): http://docstore.mik.ua/manuals/hp-ux/en/B3936-90122/ch04s07.html#babdjifd