Vol.2012-OS-123 No.2
Vol.2012-EMB-27 No.2
2012/12/5

KVS の動的性能伸縮に関する一考察

堀内浩基† 山口実靖†

クラウドコンピューティングの普及により，大量の計算資源を容易に入手できるようになった．これにより，スケーラビリティが高いデータベース管理システムである KVS (Key Value Store)が注目を集めている．クラウドコンピューティングでは計算資源を動的に確保，解放することが可能であり，KVS にも動的にノードを追加，削除してシステム規模を実行時に伸縮させる機能が用意されている．よって，クラウドコンピューティングと KVS を用いることにより，サービスにかかる負荷が高いときには多くの計算資源を確保してサービスの品質を保ち，負荷が低いときには確保計算資源量を減らして計算資源コストを抑制するという制御が可能となる．本稿では，KVS の動的性能伸縮性について考察を行う．まず，動的拡張性能(ノードの追加に要する時間)の評価結果を示し，性能拡張には長い時間を要することがあること，KVS に読み込み負荷がかかっている状況では特に性能拡張に多くの時間を要することを示す．次に，I/O 解析結果を示し，読み込み負荷がかかっている状況では性能拡張に多くの時間を要する理由を示す．最後に，KVS の伸縮性の向上手法についての考察を示す．

A Study on Performance Elasticity of Key Value Store

KOHKI HORIUCHI† SANEYASU YAMAGUCHI†

KVS (Key Value Store) is simple and scalable database management system. It is expected to be suitable for a cloud computing platform which has significant scalability. KVS has also large elasticity, with which system performance can be dynamically changed by adding or removing nodes without stopping the system. In this paper, we discuss elasticity of KVS. First, we evaluate elasticity of KVS using Cassandra which is one of the most famous KVS implementation and show inserting nodes into running system takes very long times. Second, we present analyses of system behavior. Third, we show the reason of the long joining time.

1. はじめに

近年，クラウドコンピューティングの普及に伴いデータベース管理システムのスケーラビリティの確保が重要視され，この解決策として Key-Value Store (KVS) が注目されている．KVS は Key と Value のみで構成されるシンプルなデータ構造のデータベースであり，高いスケーラビリティがある．また，KVS は実行時にノード数の増減を行うことが可能であり，クラウド環境の様に使用計算資源量を容易に増減できるシステム上で KVS を動作させれば, 性能を動的に伸縮可能であるデータベース管理システムを構築できると期待されている．

本研究では, 実行中の KVS システムにノードを追加して動的に性能を拡張させることに着目し，動的性能拡張性の評価と，動的性能拡張性の向上手法について考察を行う．

既存クラウドシステムの多くで Linux が使用されており, 同 OS は代表的な KVS のプラットフォーム OS と考えられる．よって本研究では，KVS を Linux 上で使用する環境に焦点を当て, 同環境における KVS の性能拡張性の向上手法について考える．

†工学院大学大学院工学研究科電気・電子工学専攻
Graduate School of Electrical and Electronics Engineering,Kogakuin Universit

2. Key-Value Store

KVS とは，Key と Value のみで構成される単純なデータベース管理システムである. データベース内に Key と Value の組を保存することができ，Key を指定してそれに対応する Value を得ることができる．構造が単純であるため分散環境への適用が容易であり，高い規模拡張性が得られることが多い．逆に，多くの場合提供される機能が豊富ではなく，また保証される一貫性が高くないため，利用者がこれらを考慮した運用をすることが求められる．

代表的な KVS の実装の 1 つに Cassandra [1]がある．Cassandra は Dynamo[2] の分散ハッシュテーブルと BigTable[3] のデータモデルを併せ持った Eventually Consistent な分散システム構造の KVS である．図 1 の様に, Cassandra を構成する各ノードはトークンと呼ばれるハッシュ値を持ち，リング状のハッシュ空間にトークンをもとに配置される．各ノードは，ハッシュ値が自身のトークン値以下でかつ直前ノードのトークン値より大きい範囲を担当する．データを保存または検索する際は，Key をハッシュ関数にかけそのハッシュ値から当該データの担当ノードを特定する．

稼働中の KVS システムに新規ノードを追加した場合, 新規ノードの担当トークン範囲にあたるデータが既存ノードより新規ノードに転送される．同様に, 稼働中の KVS システムからノードを削除した場合は，脱退ノードの担当トークン範囲のデータが新しく担当することになるノードに転

送される．本稿では特に，新規ノード追加の際に生じる「既存ノードから新規ノードへのデータ転送」に着目して考察を行う．

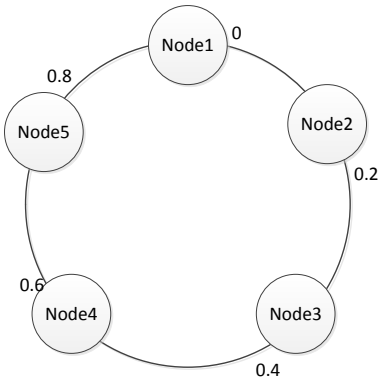

図 1 ノード配置

3. 性能伸張性の評価

KVS では，実行時にノードを追加しシステム規模や性能を動的に拡張することができる．本章では，Cassandra システムにおける規模拡張(ノード追加)性能の評価を行う．

Cassandra では，ユーザがノード追加要求(join 要求)を発行すると，ノードはまず *joining* 状態でシステムに加わり，join 処理終了後にノードは *normal* 状態となる．本稿では，「join 要求を発行した時刻」から「(*joining* 状態を終えて) *normal* 状態になった時刻」までの時間を「join 時間」と定義する．

以下の節にて，各条件下における join 時間の測定結果を示す．全ての実験において，使用した KVS は Cassandra 1.0.7，レプリカ数は 1，使用した OS は Linux 2.6.18.8，Value サイズは 16[KB]，実験環境は図 2 通り，使用計算機の仕様は表 1 の通りである．である．また，次節以降で示される読込処理とは一様分布乱数で key 指定しその value を get 要求により取得する処理であり，書込処理とは一様分布乱数で key を指定しその value を update 要求により書き込む処理である．

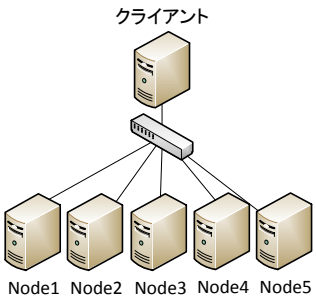

図 2 測定環境

表 1 使用計算機の仕様

クライアント,Node1,Node2	
CPU	Intel Core i3
Mem	2[GB]
HDD	150[GB]
NIC	Gigabit Ethernet

Node3,Node5	
CPU	Intel Celeron
Mem	2[GB]
HDD	150[GB]
NIC	Gigabit Ethernet

Node4	
CPU	AMD Athlon
Mem	3[GB]
HDD	150[GB]
NIC	Gigabit Ethernet

3.1 動的ノード追加による性能変化

最初に，稼働中の KVS にノードを動的に追加したときの性能変化について述べる．図 3 に，1 ノードで稼働中のシステムに 1 ノードずつ合計 4 ノード追加した時の get 性能の変化を示す．2 ノード並列に join させた時の性能の変化を図 4 に示す．図の縦軸は，16 スレッドのクライアントから発行された get 要求に対する応答時間である．データベースサイズは 16[GB]であり，初期ノード以外の各ノードの担当トークン範囲は全体の 20%，初期ノードの担当トークン範囲は 100%よりはじまりノードが追加されるごとに 20%ずつ減少させていった．図中の S は join 開始時刻を表し，図中の E は join 終了時刻を表している．

両図より，実行中にノードを追加することにより get 性能を向上させることが可能であること，join 処理中は get 性能が大幅に劣化することが確認できる．両図にて get 性能の振動を観察することができるが，本振動は join 処理を行わない状態でも発生し，join 処理と関連が大きくないため本稿では考察の対象としない．同振動は Linux OS 特有のものであり，Windows OS を用いて行った測定には存在しない．Windows OS における測定結果を付録に示す．

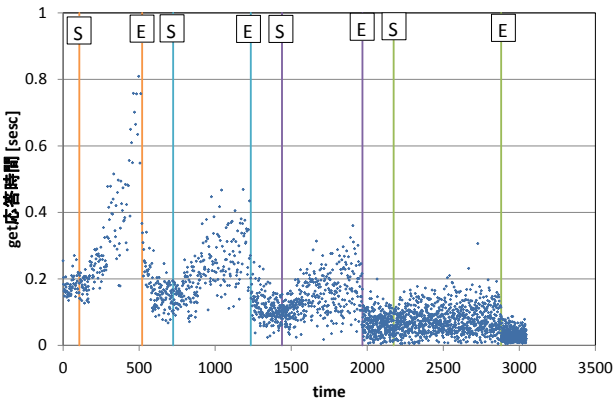

図 3 ノード join による get 応答時間の変化(1 ノードずつ)

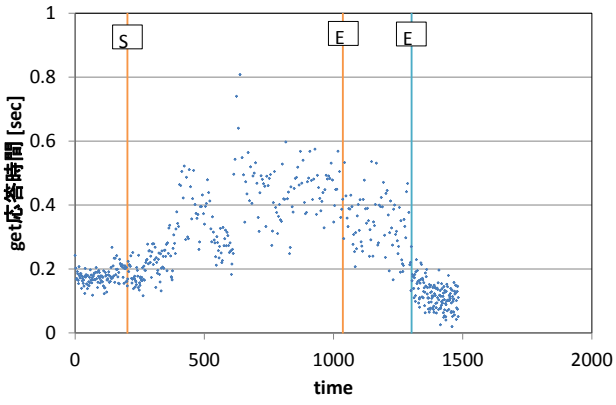

図 4 ノード同時 join による get 応答時間の変化(2 ノード同時)

3.2 データサイズと join 時間

初期ノード数が 1，データベースの総サイズが 0[GB]から 16[GB]，join 処理中の KVS の負荷状態が負荷なし，読

込負荷あり，書込負荷ありにおける 1 ノードずつ順に 4 ノード join させたときの各ノードの join 時間を図 5，図 6，図 7 に示す．ただし，初期状態では，既存の 1 ノードがトークン範囲 100%を担当しており，新規追加ノードはそれぞれ 20%を担当範囲として join を行う．

実験結果より，join 時間は 60 秒を下回らないこと，データサイズが増加するのに従い join 時間も増加する傾向があること，読み込み負荷(KVS get 処理)が存在すると join 時間が大幅に増加してしまうことがわかる．join 時間が 60 秒を下回らない理由は，Cassandra の join 実装内に 30 秒の sleep 処理が 2 個含まれているためである．

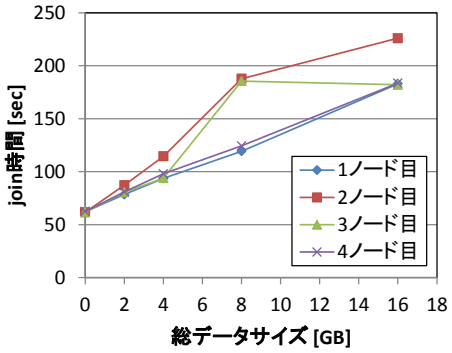

図 5 負荷なし時における総データサイズと各ノードの join 時間の関係

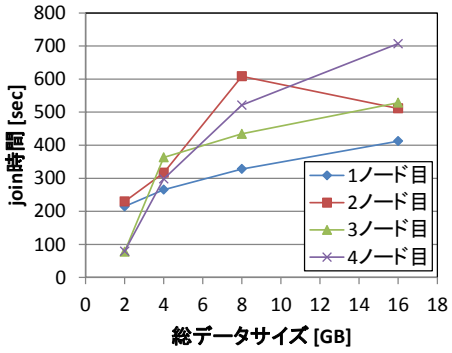

図 6 読込負荷時における総データサイズと各ノードの join 時間の関係

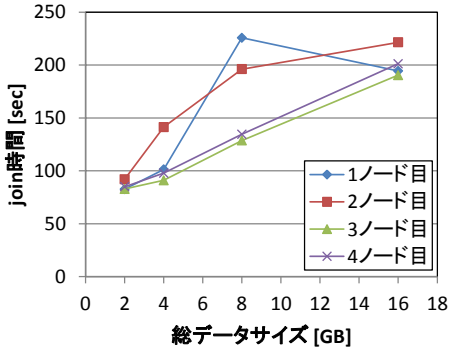

図 7 書込負荷時における総データサイズと各ノードの join 時間の関係

3.3 トークン範囲と join 時間

既存ノード数が 1，データベースの総サイズが 16[GB]，追加ノード数が 1 台，追加ノードのトークン範囲が 1[%]から 32[%]，join 処理中の KVS の負荷状態が負荷なし，読込負荷あり，書込負荷ありにおける join 時間を図 8 に示す．実験結果より，60 秒の sleep 時間を除くと join 時間はトークン範囲とほぼ比例することが分かる．また，前節同様に読み込み負荷中は，join 時間が非常に長くなることが分かる．

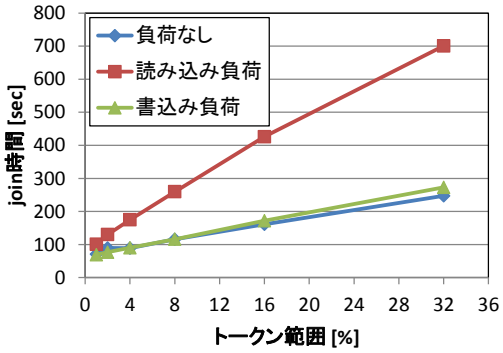

図 8 トークン範囲と join 時間の関係

3.4 並列 join ノード数と join 時間

既存ノード数が 1，データベースの総サイズが 16[GB]，並列 join ノード数が 1 から 4，join 処理中の KVS の負荷状態が負荷なし，読込負荷あり，書込負荷ありにおける join 時間を図 9 に示す．ただし，複数ノード並列 join の場合は，join 時間が最も長いノードの join 時間を「並列 join の join 時間」とした．

図より，並列 join ノード数を増加させることにより join 時間が比例以上の速度で増加し，複数台を並列して join させるより 1 台ずつ順に join を行った方が短い時間で join が完了できることが分かる．

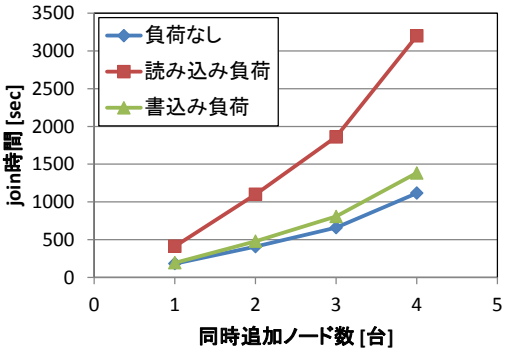

図 9 複数ノード並列

4. 考察

本章で，読込負荷が存在すると join 時間が大幅に増加してしまう原因について考察する．

4.1 読込負荷中の join 処理における発行 I/O

KVS の負荷状態が負荷なし，読込負荷あり，書込負荷ありにおける 1 台ノード join 時の disk I/O を図 10，図 11，図 12 に示す．また，これらを拡大したものを図 13，図 14，図 15 に示す．これらの図において，各プロットは HDD へのアクセス要求の発生を示しており，横軸はアクセス要求の発生時刻，縦軸はアクセス要求のアクセスアドレスである．ディスクシークについて考察するために，図 13 から図 15 においてはプロット間の線を表記してある．

負荷がない状態の図 10 に着目すると，72[GB]付近から 74[GB]付近など，join 処理のためにデータベース内のデータを連続して読み込んでいることが分かり，これがシーケンシャルリードとして効率的に処理されていることが分かる．これに対して読込負荷がある図 11, 図 14 の状況では，join 処理のためのデータベース内のデータの連続読込が get 要求の発生により細かく分断され，効率の悪いランダムアクセスとなっていることが分かる．

以上より，読込負荷があると join のためのデータベース内データの連続読み込みが効率的に行えず，結果的に join 時間が大幅にしていることが分かる．

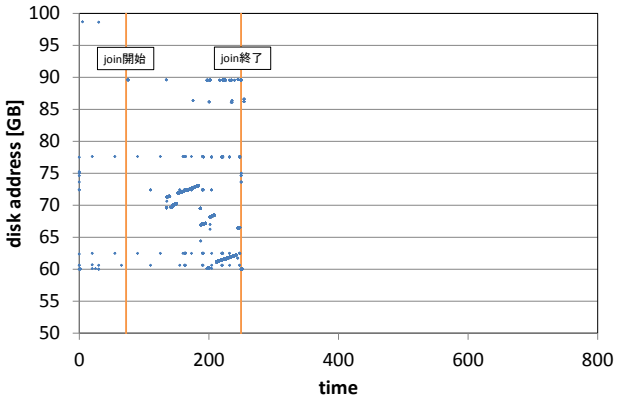

図 10 負荷なしの 1 台ノード join における disk I/O

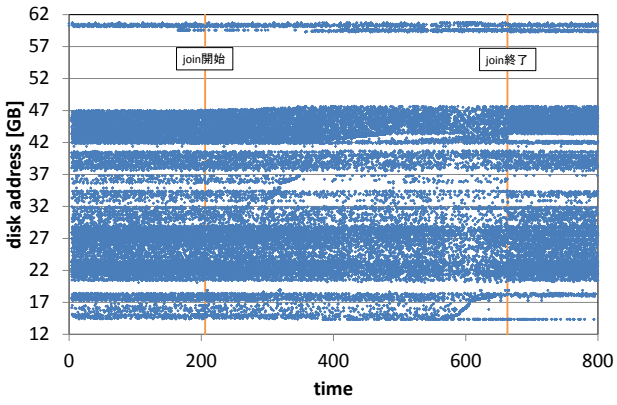

図 11 読込負荷時の 1 台ノード join における disk I/O

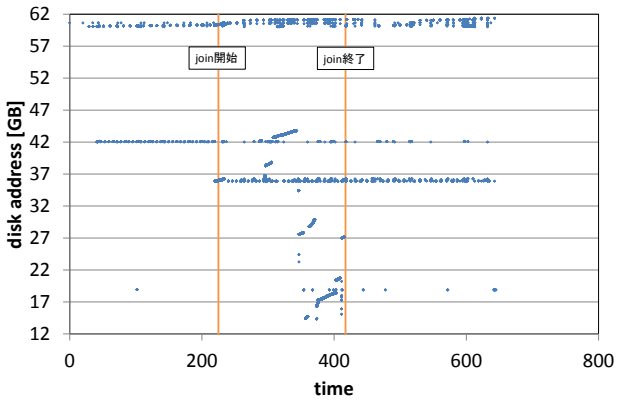

図 12 書込負荷時の 1 台ノード join における disk I/O

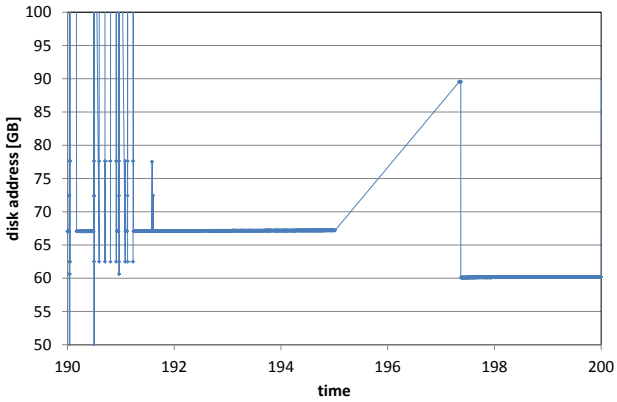

図 13 負荷なしの 1 台ノード join における disk I/O(拡大)

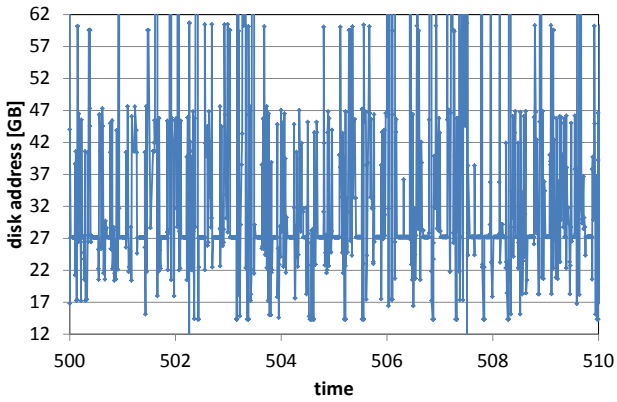

図 14 読込負荷の 1 台ノード join における disk I/O(拡大)

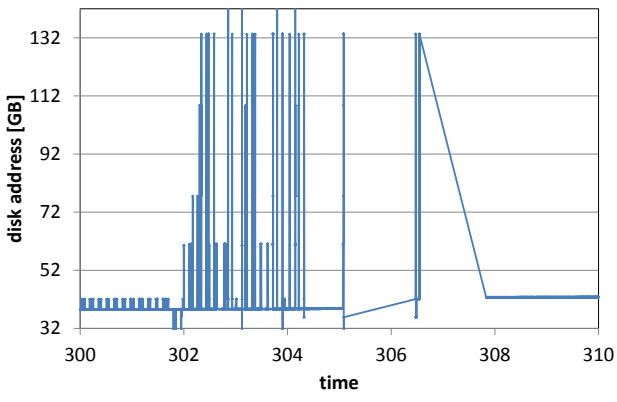

図 15 書込負荷の 1 台ノード join における disk I/O(拡大)

4.2 I/O スケジューラ

Linux OS には，NOOP，CFQ，Deadline の 3 種類の I/O スケジューラが搭載されている．I/O スケジューラごとの join 時間を図 16 に示す．図より，I/O スケジューラを変更することにより join 時間が大きく変わることがわかり，join 時間の短縮には I/O スケジューリングの最適化が効果的であると予想される．

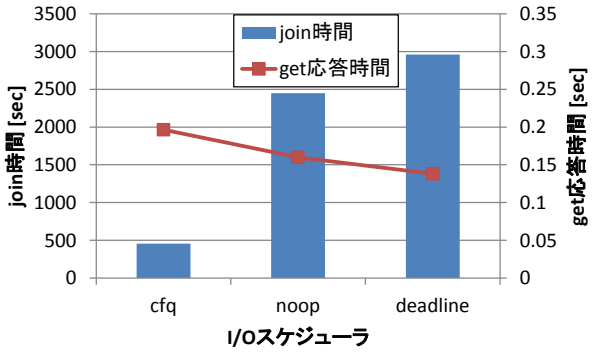

図 16 I/O スケジューラと join 時間と get 応答時間の関係

4.3 先読み処理

連続読込処理が積極的に行われるよう最大先読みサイズを変化させて join 処理を行ったときの join 性能を図 17，に示す．また，実験中の先読サイズの変化を図 18，図 19 に示す．データサイズは 4GB である．

図 17 より，最大先読サイズを初期設定の 128KB から拡大しても join 時間の短縮は実現されないことが分かる．逆に，最大先読みサイズを縮小させることにより join 時間の短縮が可能であることが確認された．また，図 18，図 19 より最大先読サイズを増加させると join 処理のための連続データ読み出しだけでなく，通常の get 処理も大きな先読みサイズで行われてしまい，結果として処理効率が低下していることが予想される．

4.4 I/O サブシステムの課題

読込負荷中に join 処理を行うと，join 時間が大幅に増加してしまう理由を OS の I/O サブシステムの側面から考察する．読込負荷中の join 処理では，粒度の細かい get 処理と，join 処理のための連続データ読込が混在している．しかしそれらの I/O 要求は両者とも Cassandra プロセスより発行されるため，I/O サブシステムは両者を区別することができない．よって，get 処理の I/O 要求に対して先読みサイズを縮小し，join 処理のための連続データ読込要求に対して先読サイズを拡大することができない．また同じ理由により，CFQ における I/O 優先度(ionice)を連続データアクセスに対してのみ行うこともできない．

これを解決するには，I/O サブシステム内においてプロセス番号以外の情報により I/O 要求を分類し最適化することが重要であると考えられる．

5. まとめ

本稿では，Linux 上で動作する Cassandra システムに着目し，その動的性能拡張性の評価，join 時間が長くなる理由の考察を行った．

今後は，join 時間の短縮手法の実装と評価を行っていく予定である．

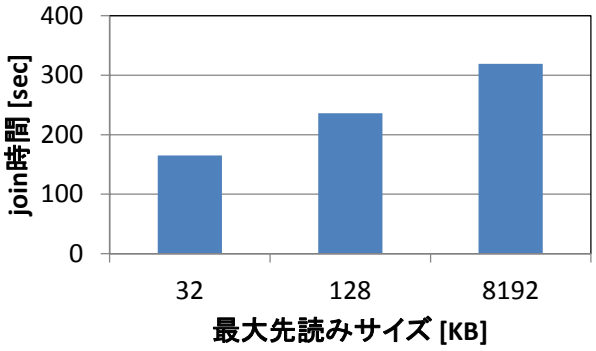

図 17 最大先読みサイズと join 時間の関係(4GB)

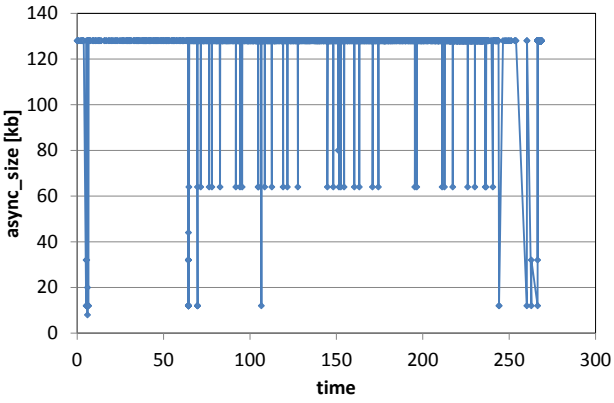

図 18 先読みサイズの変化(max=128[KB])

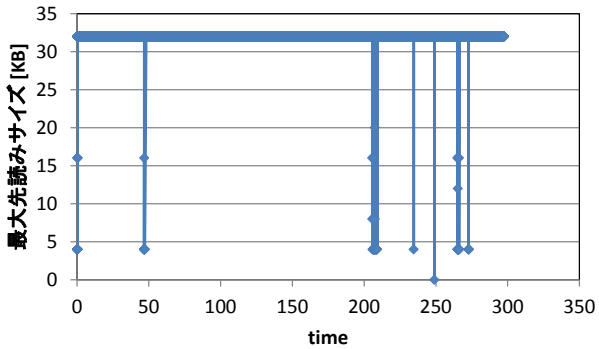

図 19 先読みサイズの変化(max=32[KB])

謝辞 本研究は JSPS 科研費 22700039, 24300034 の助成を受けたものである。

Vol.2012-OS-123 No.2
Vol.2012-EMB-27 No.2
2012/12/5

参考文献

[1] Avinash Lakshman and Prashant Malik, “Cassandra-A Decentralized Structured Storage System”, LADIS 09, 2009

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, “Dynamo: Amazon’s Highly Available Key-value Store”, SOSP ’07, 2007

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes and Robert E. Gruber, “Bigtable: A Distributed Storage System for Structured Data”, IOSDI ’06 pages 205--218, 2006

付録

Windows OS 上で Cassandra を稼働させた場合の， get 処理応答時間の推移を図 20 に示す.

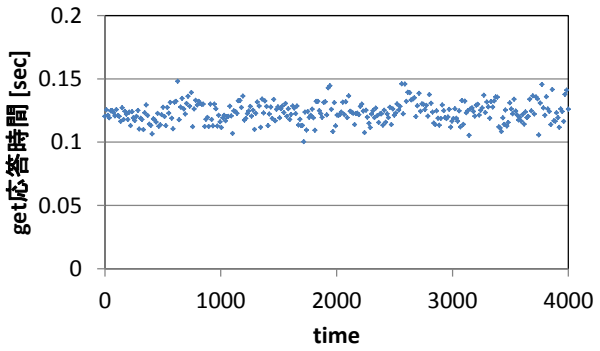

図 21 Window OS 上での get 処理応答時間の推移