社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

MapReduce処理系SSSにおけるKey Value Storeアクセス手法の改良

中田　秀基†　　小川　宏高†　　工藤　知宏†

† 独立行政法人 産業技術総合研究所
〒 305–8568 茨城県つくば市梅園 1 − 1 − 1 中央第二
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp

あらまし　われわれが開発中の MapReduce 処理系 SSS は、既存の Key Value Store（KVS）である Tokyo Cabinet を要素 KVS とする分散 KVS 上に構築されている。既存の SSS 実装では Tokyo Cabinet をネットワークサーバ化するレイヤである Tokyo Tyrant を改変し、利用してきた。しかし、性能上の問題点が確認されたため、Tokyo Tyrant を廃し、独自のネットワーク・サーバレイヤを導入した。この導入により、1）ネットワーク通信時のデータコピー削減によるアクセス高速化、2）タプルグループごとにデータベースファイルを持つことによるグループ削除の高速化、3）キーに対するプレフィックス、ポストフィックスを廃することによる転送データ量の削減、を実現することができた。

キーワード　MapReduce, 分散並列計算, KVS

Improvement of Key Value Store Access Protocol of SSS, A MapReduce System

Hidemoto NAKADA†, Hirotaka OGAWA†, and Tomohiro KUDOH†

† National Institute of Advanced Industrial Science and Technology (AIST)
Tsukuba Central 2, 1-1-1 Umezono, Tsukuba, Ibaraki 305-8568, JAPAN
E-mail: †{hide-nakada,h-ogawa,t.kudoh}@aist.go.jp

Abstract　We have been developing a distributed KVS-based MapReduce system called SSS, which could be utilized for broader applications than Hadoop. We employed Tokyo Cabinet , an open sourced key value database, as unit KVS, and modified version of Tokyo Tyrant to access Tokyo Cabinet, and found that Tokyo Tyrant is one of the performance bottleneck of SSS. This paper describes newly developped network access server called 'data server' which is specially crafted for SSS. With data server, we could 1) reduce data copy during communication, 2) speed up deletion of tuple groups, 3) reduce data amount to be transfered by omitting key prefix and postfix. We confirmed that the transition from the Tokyo Tyrant to the data server significantly contributed to the performance of SSS, especially for read throughput.

Key words　MapReduce, Distributed Parallel Execution, KVS

1. はじめに

近年広く用いられつつある MapReduce は、大規模データ処理に適したプログラミングフレームワークである。大規模データ処理には、処理の分散並列実行とデータのストレージへの入出力が必須となるが、一般にはこれらはプログラマにとって大きな負担となる。MapReduce はこれらの困難な点をシステムレベルで隠蔽し、プログラマには単純な Map および Reduce 関数のみを記述させることで、プログラマへの負担を大きく軽減する。

MapReduce はキーと値（バリュー）からなるペアをデータ処理の対象とする。したがって、分散キーバリューストア（KVS）を実装し、その上に MapReduce システムを構築することが最も直感的な実装法となる。しかし、もっとも広く用いられている Hadoop [1] では、このような実装法を取っていない。Hadoop では、Map の入力データおよび Reduce の出力データを、分散ファイルシステム HDFS 上にシリアライズして置く。これらのデータアクセスはシーケンシャルリード、ライトとなる。Map と Reduce の間のデータに対しては、通信を伴う高価な処理である Shuffle を行わなければならないため、HDFS への書き出

しを行わず、各ノードのメモリとローカルファイル上に一時的に領域を確保する。このような構造をとる Hadoop は、Map への入力が特に効率的となっている。

一方、Hadoop の構造には幾つかの問題点が指摘できる。まず、Map と Reduce が 1 対 1 で強固に接続していることが挙げられる。このため、たとえば Map の結果を複数の Reduce 処理で共有することができない。また、Map だけ、Reduce だけの処理を行うことができず、何もしない空の Reduce 関数、Map 関数と組み合わあせて実行する必要がある。

もう一つの問題点は、拡張の困難さである。計算モデルとしての MapReduce に対して、さまざまな方向への拡張が提案されている。例えば連続して入力される時系列データに対して連続的に処理を行う Continuous MapReduce [2] や Map、Reduce 処理の後段に Merge 処理と呼ばれる処理を追加することで関係代数を表現することを可能にする Map Reduce Merge [3] と呼ばれるモデルがあげられる。しかし、Hadoop をこれらのモデルに拡張することは、非常に困難である。

これに対して、われわれは、直感的でナイーブな実装手法をとり、分散 KVS 上に MapReduce システム SSS [4] [5] を構築してきた。SSS はすべてのデータ保持に、分散 KVS を用いる。分散 KVS の入出力は、単純なファイル読み書きと比較すると、はるかに複雑なため、性能低下が予想される。SSS では、計算処理と I/O 処理をパイプライン的に同時処理することで、性能の低下を最低限に抑えることに成功している [6]。

SSS は分散 KVS の基盤となるユニット KVS として Tokyo Cabinet [7] を利用し、これをネットワークを経由して利用することで分散 KVS を構成している。これまでは、Tokyo Cabinet のネットワーク・サーバとして Tokyo Tyrant [8] を改変したものを利用してきたが、その他の部分の最適化がすすむにつれ、ネットワーク・サーバのプロトコルが性能的なボトルネックとなっていた。

本稿では、Tokyo Tyrant に変わるネットワーク・サーバとして新たに実装した機構、データサーバについて詳述する。データサーバを用いることで、ネットワーク・プロトコルを最適化するだけでなく、データベースファイルの構造および Tokyo Cabinet でのキーの持ち方を変更することが可能となった。このように改変したデータサーバを用いることで、速度の向上を確認することができた。

本稿の構成は以下のとおりである。2. 節で、MapReduce プログラミングモデルおよびその実装である SSS について概説する。3. 節で SSS の従来の実装について述べる。4. 節で、本稿で新たに導入した実装手法について述べる。5. 節で提案の変更による性能向上の評価を行う。6. 節はまとめである。

2. MapReduce と SSS

2.1 MapReduce

MapReduce [9] とは、入力キーバリューペアのリストを受け取り、出力キーバリューペアのリストを生成する分散計算モデルである。MapReduce の計算は、Map と Reduce という二つのユーザ定義関数からなる。これら 2 つの関数名は、Lisp の 2 つの高階関数からそれぞれ取られている。Map では大量の独立したデータに対する並列演算を、Reduce では Map の出力に対する集約演算を行う。各 Map 関数、Reduce 関数はそれぞれ独立しており、相互に依存関係がないため、同期なしに並列に実行することができ、分散環境での実行に適している。

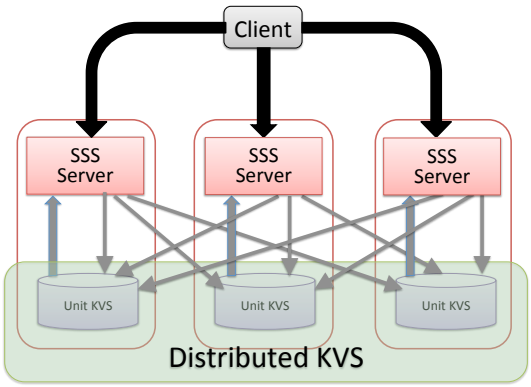

図 1　SSS の構成

Map と Reduce の間には、シャッフルと呼ばれるフェイズがありし、Map が出力するキーバリューペア全てに対してキーごとに集約を行う。ここで、分散ノード間での通信が発生する。分散計算の立場から考えると、MapReduce は通信のパターンをシャッフルに限定することで、モデルを単純化し、記述性を向上させたモデルであると考えることができる。

2.2 SSS

SSS [4] は、われわれが開発中の MapReduce 処理系である。SSS は HDFS のようなファイルシステムを基盤とせず、分散 KVS を基盤とする点に特徴がある。入力データは予めキーとバリューの形で分散 KVS にアップロードしておき、出力結果も分散 KVS からダウンロードする形となる (図 1)。

SSS ではデータをキーに対するハッシュで分散した上で、Owner Compute ルールにしたがって計算を行う。つまり、各ノード上の Mapper/Reducer は自ノード内のキーバリューペアのみを対象として処理を行う。これは、データ転送の時間を削減するとともに、ネットワークの衝突を防ぐためである。

Mapper は、生成したキーバリューペアを、そのキーでハッシングして、保持担当ノードを決定し直接書き込む。書きこまれたデータは各ノード上の KVS によって自動的にキー毎にグループ分けされる。これをそのまま利用して、Reduce を行う。つまり SSS においては、シャッフルは、キーのハッシングと KVS によるキー毎のグループ分けで実現されることになる。

SSS のもうひとつの特徴は、Map と Reduce を自由に組み合わせた繰り返し計算が容易にできることである。前述のように、SSS では Map と Reduce の間でやりとりされるデータも KVS に蓄積されるため、Map と Reduce が 1 対 1 に対応している必要がない。したがって、任意個数、段数の Map と Reduce から構成される、より柔軟なデータフロー構造を対象とすることができる。

2.2.1 タプルグループ

SSS はデータ空間を**タプルグループ**とよぶサブ空間に分割す

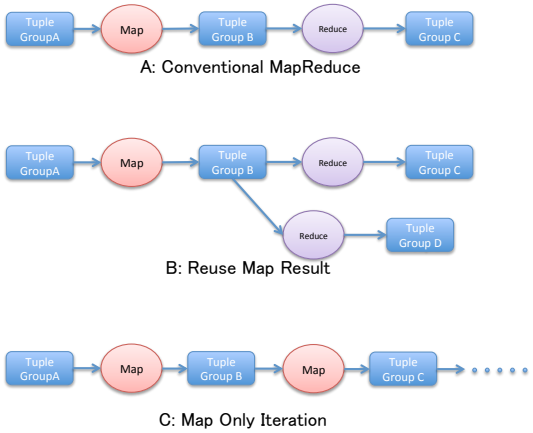

図 2　タプルグループと Map/Reduce

る。個々の Mapper/Reducer は、特定のタプルグループからタプル（キーバリューペア）を読み込み、特定の 0 個以上のタプルグループに対して出力を行う。複数の Mapper/Reducer がタプルグループを共有することも可能である。

図 2:A に、一般的なの MapReduce の様子を示す。Mapper/Reducer はそれぞれ 1 つのタプルグループから読み込み、1 つのタプルグループに対して書き出している。図 2:B では、Mapper の出力を再利用している。図 2:C は、Mapper のみによる繰り返し演算を示している。B,C のようなワークフローは Hadoop では実現できない。

2.2.2　分散 KVS の実現

SSS の分散 KVS は、各ワーカノードに設置される要素 KVS の集合から構成される。データの分散には、キーによる単純なハッシュを用いた。キーバリューペアを書き込む際、書き込みノードでキーのハッシュを取り、そのハッシュ値に基づいて書きこむノードを決定する。すべての SSS サーバがハッシュ関数を共有しているので、同じキーをもつキーバリューペアは同じ要素 KVS に書き込まれることが保証される。

3.　従来の実装

3.1　要素 KVS の実装

要素 KVS として Tokyo Cabinet を用い、Tokyo Cabinet へのリモートインターフェイスとして Tokyo Tyrant を用いていた。Tokyo Cabinet [7] は、Fal Labs の平林幹雄氏が開発したキーとバリューからなるペアを格納するデータベース、いわゆるキーバリューストア (KVS) である。Tokyo Tyrant [8] は、やはり平林氏が開発した Tokyo Cabinet に対するネットワークインターフェイス層で、リモートからの Tokyo Cabinet へのアクセス機能を提供する。Tokyo Cabinet、Tokyo Tyrant は Tokyo シリーズと呼ばれる一連の DB 関連製品群の一部である

また、SSS からの Tokyo Tyrant へのブリッジとしては Tokyo Tyrant の Java Binding [10] を用いた。これは、Tokyo Tyrant の C クライアントを Java でラップしたものである。

3.2　タプルグループの実装

Tokyo Cabinet はタプルグループのような名前空間をサポートしていないので、われわれはこの機能をキーのエンコーディングで実現した (図 3)。Tokyo Cabinet のキーに、ユーザの指定したキー（ユーザ・キー）だけでなく、タプルグループの ID をエンコードする。このようにエンコードすることで、特定のタプルグループに属するすべてのタプルを取得するには、プレフィックスを指定したスキャンが出来れば良いことになる。これは、Tokyo Cabinet で容易に実現できる。

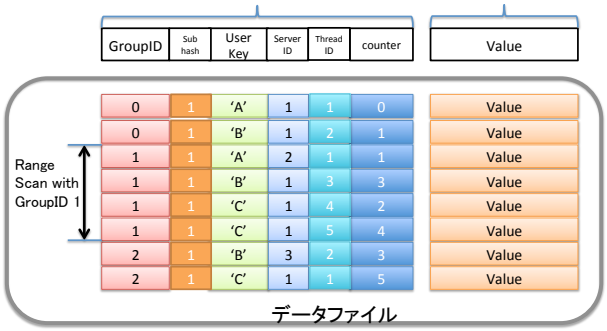

図 3　タプルグループの実装

3.3　同一キーの取り扱い

MapReduce では、同一のユーザ・キーに対して複数の値を持つ必要がある。Tokyo Cabinet は同一キーに対して複数の値を持つことができるが、Tokyo Tyrant は、範囲指定を行なってバルクで読み出す場合には、同一ユーザ・キーを持つ複数のタプルをうまく取り扱うことができない。これは、指定した範囲を何度かに分けて転送する際に、転送の切れ目となる位置を指定することが不可能なためである。

われわれは、タプルグループ ID と同様に、SSS サーバ番号、スレッド番号、カウンタに関しても、キーの一部としてエンコードすることでこの問題を回避する。図 3 に示す通り、ユーザ・キーの後ろにこれらの値を付加することで、キーはかならずユニークとなり、前述の問題は発生しない。

3.4　サブハッシュ

SSS の Map/Reduce 関数は単一のグループに対して読み出しを行う。ノード上のコアを有効に利用するためには、読み出しを複数のストリームに対してマルチスレッドで行う必要がある。このために、1 つのノードに割り当てられたデータを、更に分割して記録する。分割にはユーザ・キーのハッシュ値の下位ビットを用い、これをプリフィックスの一部として、グループ ID の後ろに付加する (図 3)。これによって、一つのタプルグループをプレフィックススキャンによって、複数のストリームとして読み出すことが可能になる。

3.5　SSS アクセスパターンと最適化

前項までに述べたプレフィックスの構造を仮定すると、SSS での KVS に対するアクセスは、範囲指定のバルク読み出し、同一プレフィックスデータのバルク書き込み、範囲指定のバルク消去の 3 種類に限定される。

このアクセスパターンに特化するよう、Tokyo Cabinet および Tokyo Tyrant に対して改変を行った ([6])。具体的には、始点キーと終点キーを指定し、その間の範囲のキーをもつキーバリューペアに対して一括して操作を行うコマンドを追加すると同時に、バルク操作に対して最適化するするために、内部の

ロックを解除せずに連続してアクセスするよう改変した。

3.6 問 題 点

前述のように実装された分散 KVS には、下に述べるようにいくつかの問題点がある。

- 不要なデータコピー

Tokyo Tyrant では、データを一度特定の形式の構造体にコピーした上で転送する。このデータコピーは本来不要である。特にデータ量が多い場合にはこのデータコピーのオーバヘッドが問題となる。

- タプルグループの削除が遅い

SSS がターゲットとする繰り返し処理に於いては、1 イタレーションごとに古いデータを削除し、新しいデータを生成する場合が多い。既存の実装では、タプルグループは、ID をプレフィックスを持つタプルの集合として表現されるが、これを一括で削除する方法は Tokyo Cabinet には用意されておらず、タプルを個別に削除するしかない。このため、削除が低速となり、イタレーション速度を制約する結果となっている。

- キーのサイズがオーバヘッドに

前述のように、SSS ではプレフィックスとポストフィックスをユーザ・キーについかすることで、タプルグループを表現し、同一キーを持つタプルの動作を実現している。しかし、タプルグループ ID として 17 バイト (うち 1 バイトはキーのフォーマットを表すフラグ)、サブハッシュ1 バイト、サーバ番号、スレッド番号がそれぞれ 4 バイトカウンタが 8 バイトで、計 34 バイトを追加する結果として、Tokyo Cabinet 上のキーのサイズはユーザキーに対して 34 バイト増加する。キーや値が十分大きい場合にはこのオーバヘッドは問題にならないが、そうでない場合にはデータのスループットに直結する問題となる。

4. データサーバによる実装

前節で述べたように、Tokyo Tyrant を用いた実装には幾つかの問題点があったため、あらたに Tokyo Tyrant に変わるレイヤとしてデータサーバを実装した。KVS としては Tokyo Cabinet を引き続き利用する。

4.1 基本的な方針

Tokyo Tyrant での実装では、Tokyo Cabinet のデータベースファイルは Tokyo Tyrant サーバあたり 1 つであり、ユーザ・キーにさまざまなプレフィックス、ポストフィックスをつけることで、さまざまな機能を実現していた。これは、Tokyo Tyrant では複数のデータベースファイルをあつかうことができないためである。

データサーバは、以下の方針にもとづいて設計した。

- 複数のデータベースファイルを用いる。具体的には、各タプルグループのサブハッシュごとにファイルを設ける。これによってプレフィックスを利用する必要がなくなる。さらに、タプルグループの削除時にはデータベースファイルを削除すればよいので、高速な削除が可能になる。
- データをチャンクとして送るのはなく、ストリームをベースとしたプロトコルとする。これによって、余分なデータコピーの必要性がなくなる。さらに、3.3 で述べた問題がなくなるため、同一キーのデータをサーバ番号等で区別する必要がなくなり、ポストフィックスも省くことができる。

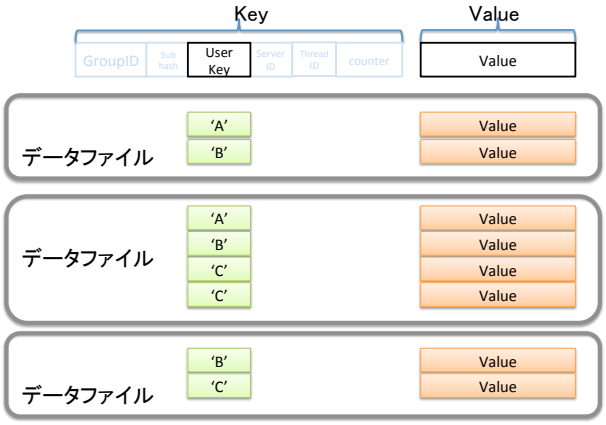

図 4 データサーバにおけるタプルグループの実装

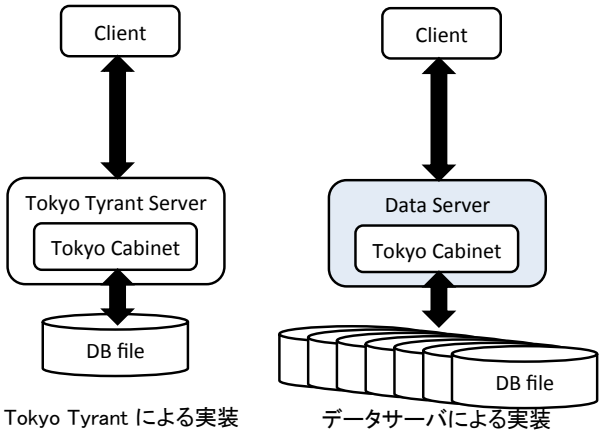

図 5 Tokyo Tyrant とデータサーバ

図 4 にデータサーバにおけるタプルグループの実装を示す。キーはユーザキーのみとなり、プレフィックス、ポストフィックスがなくなっている。グループはデータファイルとして表現される。

4.2 実 装

データサーバは C++で実装した。データサーバは Tokyo Cabinet ライブラリとリンクされ、データベースファイルの操作を行う。図 5 に、従来の実装と新たな実装の比較を示す。従来の実装の Tokyo Tyrant を、データサーバが置き換えている。また、従来の実装ではデータベースファイルはひとつだったが、新たな実装では、数多くのデータベースファイルを同時に使うものとなっている。

5. 評 価

5.1 評 価 環 境

評価には、表 1 に示すように、1 台のマスターノードと 16 台のワーカーノード（ストレージノードと MapReduce 実行ノードを兼ねる）からなる小規模クラスタを用いた。各ノードは 10Gbit Ethernet で接続され、各ワーカーノードは Fusion-io ioDrive Duo 320GB(SSD と表記) と Fujitsu の 147GB SAS HDD を備えている。

表 1　Benchmarking Environment

# nodes	17 (*)
CPU	Intel(R) Xeon(R) W5590 3.33GHz
# CPU per node	2
# Cores per CPU	4
Memory per node	48GB
Operating System	CentOS 5.5 x86_64
Storage (ioDrive)	Fusion-io ioDrive Duo 320GB
Storage (SAS HDD)	Fujitsu MBA3147RC 147GB/15000rpm
Network Interface	Mellanox ConnextX-II 10G Adapter
Network Switch	Cisco Nexus 5010

(*) うち 1 ノードはマスタサーバ

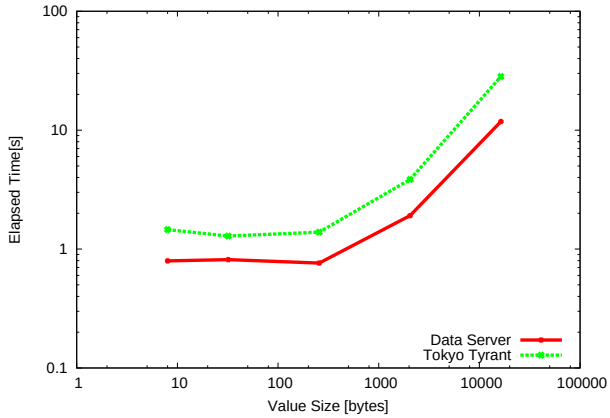

図 6　マイクロベンチマーク（読み出し）

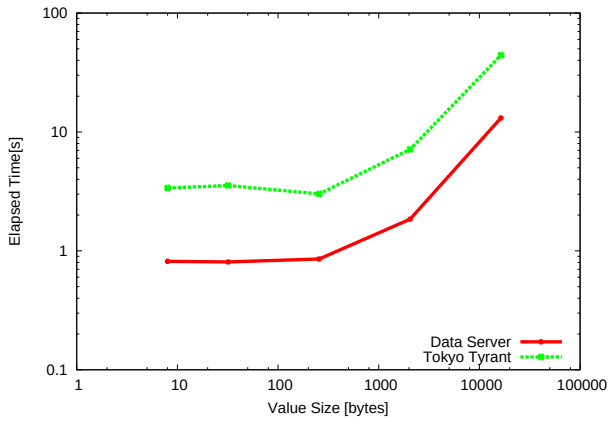

図 7　マイクロベンチマーク（書き込み）

5.2　マイクロベンチマークによる評価

基本的な性能特性を知るために、1 ノード上に SSS サーバとデータサーバもしくは TokyoTyrant を起動し、SSS サーバからデータを読み書きする実験を行った。実験では、1Mi 個のレコードを 16 スレッドで同時に読みだし、もしくは書き込みを行い、終了するまでの時間を測定した。ストレージとしては SSD を用いた。タプルのキーサイズは 32byte に固定し、値のサイズを 8byte, 32byte, 256byte, 2Kibyte, 16Kibyte と変化させた。

読み出しの結果を図 6, 書き込みの結果を図 7 に示す。X 軸、Y 軸ともに、対数となっていることに注意されたい。データサイズが小さい場合のレイテンシ、大きい場合のスループットともに大きく改善されていることがわかる。

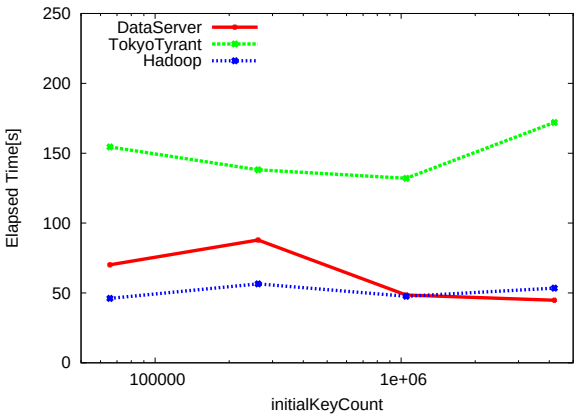

図 8　合成ベンチマーク（読み出し）

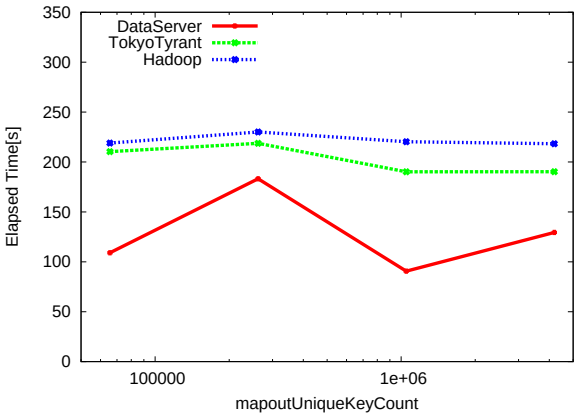

図 9　合成ベンチマーク（書き込み）

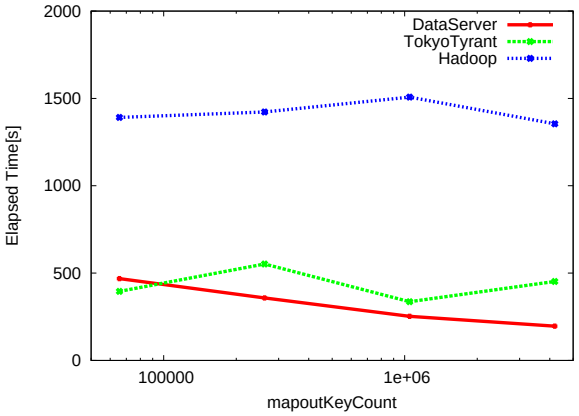

図 10　合成ベンチマーク（シャッフル）

5.3　合成ベンチマークによる評価

つぎに、MapReduce 処理系全体としての性能を見るために、[11] で提案した合成ベンチマークによって評価を行った。データの値の総量を 1Tibyte に固定したまま、値のサイズとタプルの総数を変化させている。

比較のため、Hadoop での測定値も表示している。Hadoop はこのベンチマークの設定では起動のみで 23 秒かかるため、公平な比較のために、それぞれの起動時間を全実行時間から引いてプロットしている。

図 8, 図 9, 図 10 にそれぞれ読み出し、書き込み、シャッフル

表 2　削除時間 [s]

環境 ＼ タプル数		100Ki	1000Ki	10000Ki
SSD	Tokyo Tyrant 実装	1.28	8.11	43.46
	データサーバ実装	0.68	0.63	0.60
HDD	Tokyo Tyrant 実装	3.63	43.46	527.73
	データサーバ実装	0.66	0.64	0.67

の結果を示す(注1)。

Tokyo Tyrant と比較するとほぼすべての場合において、データサーバが高速であることが確認できる。読み出し時に着目すると、Tokyo Tyrant では Hadoop と比較して大きく劣っていたスループットが改善し、タプル数が大きい場合には Hadoop よりも高速であることがわかる。書き込み時の Tokyo Tyrant に対する性能向上も大きい。

5.4　タプルグループ削除による評価

データベースファイルの管理方法の変更による削除時間への影響を知るために実験をおこなった。実験では、マイクロベンチマークと同様に、クライアントプログラムをデータサーバもしくは Tokyo Tyrant と同じノードに設置した。100Ki 個、1000Ki 個、10000Ki 個のタプルを持つタプルグループを作成し、これを削除するのにかかる時間を測定した。タプルのサイズはキーと値ともに 32byte とした。結果を表 2 に示す。

Tokyo Tyrant による実装では、タプル数の増大にともない削除にかかる時間が増大していく事がわかる。これに対して、データサーバによる実装では、タプル数にかかわらず一定時間で削除できている。

6. おわりに

MapReduce 処理系 SSS で用いる、KVS との通信レイヤを既存の Tokyo Tyrant から独自に実装したデータサーバに置き換えることにより、性能が向上を図った。データベースファイルを細分化し、ファイル単位で操作することで、タプルのユニークネスを保証するためのタグが不要になり、データ転送の際のオーバヘッドを大きく削減することができることを確認した。また、削除時間も大きく低減できることを確認した。

今後の課題としては実アプリケーションでのデータサーバによる性能への寄与の確認が挙げられる。また、性能向上による消費電力削減の確認も課題である。

謝　　辞

本研究の一部は，独立行政法人新エネルギー・産業技術総合開発機構（NEDO）の委託業務「グリーンネットワーク・システム技術研究開発プロジェクト（グリーン IT プロジェクト）」の成果を活用している．

文　　献

[1] : Hadoop, http://hadoop.apache.org/.
[2] Backman, N., Pattabiraman, K., Fonseca, R. and Cetintemel, U.: C-MR: Continuously Executing MapReduce Workflows on Multi-core Processors, *Proc. of the 3rd International Workshop on MapReduce and its Applications, MapReduce '12* (2012).
[3] Yang, H.-C., Dasdan, A., Hsiao, R.-L. and Parker, D. S.: Map-reduce-merge: simplified relational data processing on large clusters, *Proc. of SIGMOD*, pp. 1029–1040 (2007).
[4] Ogawa, H., Nakada, H., Takano, R. and Kudoh, T.: SSS: An Implementation of Key-value Store based MapReduce Framework, *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (Accepted as a paper for First International Workshop on Theory and Practice of MapReduce (MAPRED'2010))*, pp. 754–761 (2010).
[5] 中田秀基, 小川宏高, 工藤知宏: 分散 KVS に基づく MapReduce 処理系 SSS, インターネットコンファレンス 2011 (IC2011) 論文集, pp. 21–29 (2011).
[6] 中田秀基, 小川宏高, 工藤知宏: MapReduce 処理系 SSS に向けた KVS の改良, 信学技報, Vol.112, No.2 CPSY2012-1 - CPSY2012-8, pp. 19–24 (2012).
[7] FAL Labs: Tokyo Cabinet: a modern implementation of DBM, http://fallabs.com/tokyocabinet/index.html.
[8] FAL Labs: Tokyo Tyrant: network interface of Tokyo Cabinet, http://fallabs.com/tokyotyrant/.
[9] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (2004).
[10] : Java Binding of Tokyo Tyrant, http://code.google.com/p/jtokyotyrant.
[11] 小川宏高, 中田秀基, 工藤知宏: 合成ベンチマークによる MapReduce 処理系 SSS の性能評価, 情報処理学会研究報告　2011-HPC-130 (2011).

(注1)：シャッフル時にはキーが全く重ならないようにキーを調整した。これは、データサーバにキーが大量に重複した場合に動作が不安定になる不具合があるためである。