情報処理学会論文誌　コンピューティングシステム　Vol.6　No.3　71–81 (Sep. 2013)

集約処理を用いたMapReduce最適化手法の提案と実装

小沢 健史[1,a)]　鬼塚 真[2,b)]　福本 佳史[1,c)]　盛合 敏[1,d)]

受付日 2012年12月21日, 採録日 2013年4月18日

概要：本稿では，MapReduce で行う処理のうち，部分集約が可能な処理を高速化する手法を示す．部分集約ができる処理に対して，既存研究では集約処理に特化した処理系を新たに作成することにより高速化を行っていた．しかし，これらの手法は MapReduce の仕組みを大幅に変更する必要があることから，Hadoop に組み込むのは困難であった．そこで本研究では，Hadoop への実装コストを低く抑え，耐故障性を担保しつつ高速化を行う Map Multi-Reduce の提案を行う．Map Multi-Reduce は，MapReduce に計算機ごとの集約処理を行う機能を追加した，MapReduce の拡張版である．提案手法の実装を行うにあたり行った Hadoop への変更量は約 800 行と小さい．このように少ない変更量にもかかわらず，実験により，300 GB の WordCount を行う際に Map 処理と Reduce 処理間のデータの受け渡しを削減し，処理速度が 1.5 倍になることを確認した．

キーワード：MapReduce，分散処理，分散システム

MapReduce Optimization Using Mapper-side Aggregation

Tsuyoshi Ozawa[1,a)]　Makoto Onizuka[2,b)]　Yoshifumi Fukumoto[1,c)]　Satoshi Moriai[1,d)]

Received: December 21, 2012, Accepted: April 18, 2013

Abstract: In this paper, we propose a MapReduce optimization by using mapper-side aggregation designed for aggregation queries. The mapper-side aggregation has been applied in different platforms, however, it is difficult for related work to be embedded within existing MapReduce framework like Hadoop, because its mechanism of task scheduling or monitoring is different and MapReduce framework does not provide inter-process communication facilitiy. To solve this problem, we prototype Map Multi-Reduce, while preserving MapReduce semantics with small modification against Hadoop. Map Multi-Reduce is an extension of MapReduce to support node-level aggregation feature with fault tolerance. Map Multi-Reduce aggregates the the outputs of multiple MapTasks in same machines and is implemented in only 800 LOC. Map Multi-Reduce improves 1.5 times faster in WordCount processing against 300 GB dataset by cutting down shuffle cost.

Keywords: MapReduce, distributed computing, distributed system

1. はじめに

記憶装置・計算機・インターネットの発展により，産業界において取り扱うデータの量が爆発的に増加しつつある．蓄積された大量のデータを安く，実用的な速度で処理するために，安価な計算機クラスタを用いて効率的に分散計算を行うフレームワークに注目が集まっている．

特に，MapReduce は最も普及している分散処理フレームワークの 1 つであり，Google，Facebook，Yahoo! をはじめとする多くの企業に利用されている [3], [6], [13], [15]．MapReduce では，並列処理を行う Map 関数と，Map 関数の結果を集約する Reduce 関数を記述するだけで，計算機の故障や，計算機間の同期処理を意識することなく，大規模な分散処理を行うことができる [6]．Hadoop は，MapRe-

[1] NTT ソフトウェアイノベーションセンタ
NTT Software Innovation Center, Musashino, Tokyo 180–8585, Japan
[2] NTT ソフトウェアイノベーションセンタ，機械学習・データ科学センタ
NTT Software Innovation Center, Machine Learning and Data Science Center, Musashino, Tokyo 180–8585, Japan
a) ozawa.tsuyoshi@lab.ntt.co.jp
b) onizuka.makoto@lab.ntt.co.jp
c) fukumoto.yoshifumi@lab.ntt.co.jp
d) satoshi.moriai@gmail.com

duce のオープンソース実装であり，Yahoo! や Cloudera, Hortonworks が主導で開発を進めている [2]．Hadoop は，単純なオンプレミス環境だけでなく，クラウド環境上のサービスとしても提供され，多くの企業で利用されている [1]．

MapReduce には，タスクを実行するために 2 種類のプロセスがある．1 つは Mapper，もう 1 つは Reducer である．Mapper は Map 関数を実行する MapTask を処理し，Reducer は Reduce 関数を実行する ReduceTask を処理する．MapReduce で行われる処理の中には，Reducer が 1 つであることを強いる処理がある．たとえば，WordCount を行った後に，頻度の多い単語順にソートする処理などである．このような処理の場合，すべての MapTask の結果が 1 つの Reducer へ転送されるため，Reducer が行う処理が増大し，過負荷になってしまい処理に時間がかかるという問題がある．

この問題の対策は 2 つある．1 つ目は，in-mapper combining というテクニックを利用して Reducer への負荷を減らす手段である [10]．in-mapper combining は，ユーザがプログラムを書き直すことで，Map 関数の中で集約処理を行う．in-mapper combining では，ほとんどの処理をメモリ内で完結し，集約処理を行うことができるため，Mapper のディスク IO と Mapper–Reducer 間の通信量を最小限に抑えることができる．2 つ目の方法は，Combiner という機能を利用する方法である．Combiner は，Hadoop で提供されている基本機能であり，1 つの MapTask の中で出力される中間ファイルに対し，集約処理を行うための仕組みである．Combiner を利用することにより Mapper と Reducer の間の IO を減らすことができる．しかし，in-mapper combining，Combiner どちらを用いても，集約処理の適用範囲は 1 つの MapTask の結果に閉じているため効果が限定的である．特に計算機がマルチコアであるような環境では，多数の MapTask が 1 台の計算機で実行されることが多い．すると，MapTask の中間出力ファイルは同一計算機内で複数生成されるにもかかわらず，それらを計算機ごとに集約することができない．よって，これらの手法だけでは計算機の増加/入力データ量の増加にともない，Reducer に渡されるデータ量も増加してしまい，Reducer が処理のボトルネックになってしまう．つまり，スケーラビリティに限度があるという問題点が生じる．

そこで本稿では，既存の Combiner を強化した，Map Multi-Reduce を提供する．Map Multi-Reduce は，計算機ごとの集約を行う機能を追加した Hadoop の拡張版であり，以下に示す 2 つの特徴を持つ．1 つ目は，データの規模増大に対して，高いスケーラビリティを持つことである．2 つ目は，MapReduce と同様の耐故障性を持つことである．この 2 つの特徴を有することにより，ユーザは既存の Combiner を用いた MapReduce との違いを意識することなく，高いスケーラビリティを有する MapReduce プログラムを実行することが可能となる．

本手法は，Combiner を利用できる処理において有効である．Combiner を利用できる処理とは，つまり，MapReduce で処理を記述した際に，演算順序が依存しない—可換法則と結合法則を同時に満たす—処理である．処理全体が演算順序に依存しない処理の組合せである代表例として，複数の単語数を数え上げる WordCount や，単語間の共起頻度計算がある．また，処理の一部が演算順序に依存しない計算の例として，平均や標準偏差などがある．

本稿では，Map Multi-Reduce の実装と評価を行い，1.5 倍の処理速度向上が見込めることが確認した．また，計算機ごとに集約を行う機能により，Mapper–Reducer 間の通信量を削減できることを確認し，1 台で動作する Mapper 数が大きい環境，および大きなデータに対してより高速になることを示す．

以下，2 章で MapReduce の基本的な挙動と in-mapper combining/Combiner を用いた MapReduce 処理の効率化とその問題点について述べる．3 章で in-mapper combining/Combiner の問題点を解決した Map Multi-Reduce を提案する．4 章で Map Multi-Reduce の実装について述べ，5 章では，実験による高速化の評価を示す．6 章で関連研究について述べ，7 章で本稿をまとめる．

2. 前提知識

2.1 MapReduce

図 1 を用いて，本研究が改善の対象とする MapReduce について説明する．MapReduce ジョブは Map フェーズと Reduce フェーズの 2 フェーズから構成される．Map フェーズでは入力データ（D）を読み込んで Key/Value ペア (K_1, V_1) を生成し，それを入力として各ペアに対してユーザが定義した Map 関数を実行し，新たな Key/Value ペアリスト (K_2, V_2) を中間データとして出力する．Reduce フェーズでは，まず中間データを Key ごとにグルーピング

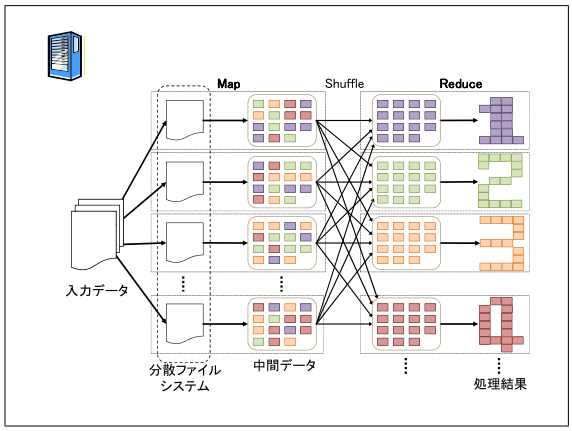

図 1　MapReduce の概要
Fig. 1　The overview of MapReduce.

(Shuffle) する．そして各グループを入力としてユーザが定義した Reduce 関数を実行し，結果として Key/Value ペアリスト (K_3, V_3) を出力する．この一連の流れを式として表現すると，以下のようになる．

$D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2\}$
$shuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\}$
$reduce(K_2, \{V_2\}) \rightarrow (K_3, V_3)$

MapReduce 処理は複数のノード (計算機) をネットワークで相互接続したクラスタ内で行われる．クラスタはリソースの割当ておよびタスクの割当てを行うマスタノードと計算を行うスレーブノードで構成される．クラスタ上には DFS (分散ファイルシステム) が構築され，入力データが格納される．MapReduce ジョブを開始すると，まずマスタが入力データをユーザの指定したサイズで分割し複数の InputSplit を生成する．次にマスタが InputSplit の数だけ MapTask を生成し，各スレーブに割り当てる．MapTask が割り当てられたスレーブでは Mapper が起動され，ユーザが定義した Map 関数が実行されて，Key/Value 形式の中間データが出力される．中間データは同じ Key 値を持つものが 1 つのスレーブに集まるようにネットワークを介して相互に移動 (shuffle) される．マスタはユーザが定義した数の ReduceTask を生成し，各スレーブに割り当てる．このとき，中間データは Key ごとに分配される．ReduceTask が割り当てられたスレーブでは Reducer が起動され，ユーザが定義した任意の Reduce 関数が実行されて，新たに Key/Value 形式の処理結果が DFS に出力される．

2.2 Mapper ごとの集約処理による MapReduce 処理の最適化

2.2.1 Combine 処理

Combine 処理は，Hadoop で提供されている基本機能であり，1 つの MapTask の中で出力される中間出力に対し，集約処理を行うための仕組みである．**図 2** に，集約処理を行わない MapReduce 標準の処理フローを，**図 3** に，Combine 処理を利用したときの処理フローを示す．Combine 処理に利用できるクラスは，Reduce クラスを継承したクラスのみである．Combine 処理の前提として，Reduce 処理を行うことによりデータサイズが減ることが多いという前提がある．たとえば，WordCount の場合，

$(K_2, V_2) = (apple, (1, 1, 1, 1, 1))$
$(K_2, V_2) = (sweet, (1, 1))$

となっているとき，Reduce 関数を Combine 処理として適用することで

$(K_2, V_2) = (apple, (5))$
$(K_2, V_2) = (sweet, (2))$

と変換することができ，データ量を減らすことができる．この性質により，Mapper と Reducer の間の IO を減らすことが可能となる．Hadoop では，Combine は MapTask

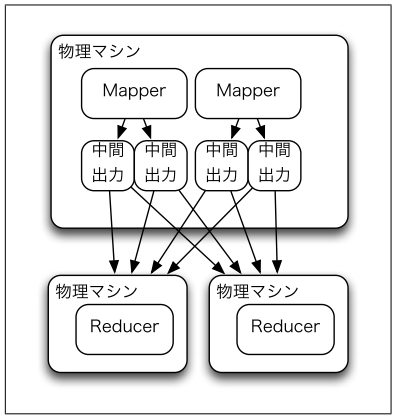

図 2　中間出力の集約を処理を行わないときの処理フロー
Fig. 2　Processing flow without aggregation of intermediate outputs.

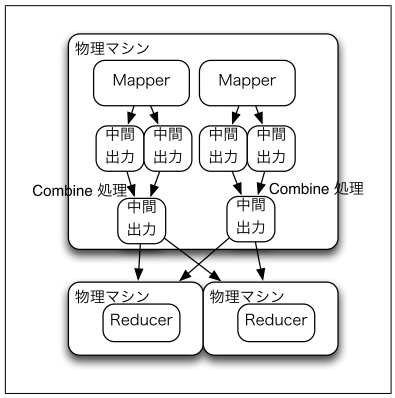

図 3　中間出力の集約を Combine 処理を用いて行ったときの処理フロー
Fig. 3　Processing flow with Combiner, mapper-level aggregation of intermediate outputs.

の一部として実現されており，

$$MapTask \begin{cases} D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2\} \\ combine(\{K_2, \{V_2\}\} \rightarrow \{K_2, V_2\} \\ localshuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\} \end{cases}$$

$$Shuffle \left\{ shuffle(\{K_2, V_2\}) \rightarrow \{K_2, \{V_2\}\} \right.$$

$$ReduceTask \left\{ reduce(K_2, \{V_2\}) \rightarrow (K_3, V_3) \right.$$

と表すことができる．Combine 処理が性能向上に寄与する場合，Map 関数の出力データ量を $|V_m|$，Combine 関数の出力データ量を $|V_c|$ とすると，$|V_m| > |V_c|$ が成立する．

ただし，現在の Hadoop 上の仕組みでは，Combine 処理の適用範囲は 1 つの MapTask の中に閉じているため，計算機内で実行された複数の MapTask の出力を集約することはできない．

2.2.2 in-mapper combining

in-mapper combining は，Map 関数の中で集約処理を行

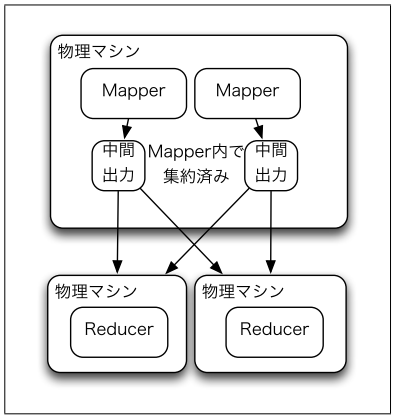

図 4　中間出力の集約を in-mapper combining を用いて行ったときの処理フロー
Fig. 4　Processing flow with in-mapper combining.

```
class Mapper
  method Setup
    H = new AssociativeArray

  method Map(docid a; doc d)
    for all term ∈ doc d do
      H{t} ← H{t} + 1

  method Cleanup
    for all term t ∈ H do
      Emit(term t; count H{t})
```

図 5　WordCount における in-mapper combining の例
Fig. 5　Pseudo code of WordCount with in-mapper combining.

うように，ユーザがプログラムを書き直す最適化手法である．in-mapper combining の処理フローを**図 4** に，WordCount における in-mapper combining の例を**図 5** に示す．in-mapper combining の一般的な実装では，Mapper のインスタンス初期化時に呼ばれる Setup メソッドの中でハッシュマップを作成する．そして，Map メソッドの中では Emit を行う代わりにハッシュマップを利用した逐次集約を行う．Mapper の終了時，Cleanup メソッドが呼ばれるため，ここでハッシュマップ内に保存されているすべての値を Emit する．

Combine 処理と in-mapper combining の違いは，Combine 処理が Hadoop の管理するバッファ上の中間出力を対象とするのに対し，in-mapper combining は Map 関数内のユーザが管理するメモリ上で集約処理を行ってしまい，集約済みの値を Reducer に渡す部分である．

メモリを利用することで，ほとんどの処理をメモリ内で完結させることができるため，Mapper のディスク IO と Mapper–Reducer 間の IO の多くを減らすことができる．

Hadoop では，in-mapper combining は map 処理の一部として実現されるので，集約済みの値を V_2' とおくと

$D \rightarrow map(K_1, V_1) \rightarrow \{K_2, V_2'\}$

$shuffle(\{K_2, V_2'\}) \rightarrow \{K_2, \{V_2'\}\}$

$reduce(K_2, \{V_2'\}) \rightarrow (K_3, V_3)$

と表すことができる．in-mapper combining が高速化に寄与するのは，Map 関数の出力データ量を $|V_m|$，in-mapper combining による出力データ量を $|V_{im}|$ とすると，$|V_m| > |V_{im}|$ が成立する場合のみである．

in-mapper combining による集約処理の適用範囲は，Combine 処理同様 1 つの MapTask に閉じているため，計算機内で実行された複数の MapTask の出力を集約することはできない．

2.3　MapReduce の耐故障性

MapReduce では，ジョブを構成するプロセスの一部および計算機の一部が故障しても故障した Task のみをやり直すことでジョブを中断することなく実行することができる．

プロセスの故障には MapTask の故障と ReduceTask の故障の 2 通りがある．MapTask が故障した場合は，分散ファイルシステムから再度データをロードし直し，MapTask をやり直す．ReduceTask が故障した場合は，ディスクに書かれている MapTask の中間出力をコピーし直して，再度 ReduceTask を実行する．

計算機の故障が発生した場合，その計算機が保持している MapTask の中間出力結果は失われてしまう．そのため，その計算機で実行されたすべての MapTask/ReduceTask をやり直すことで，ジョブを続行する．

3.　Map Multi-Reduce の設計

2.2 節で述べたように，現在の MapReduce では，集約処理の適用範囲は 1 つの MapTask に閉じられているため，集約効果が限定的である．この問題を解決するため，Map Multi-Reduce を提案する．Map Multi-Reduce は，MapReduce と同等の耐故障性を担保しつつ計算機ごとに Combine 処理を動作させることができる MapReduce の拡張である．以下では，Local Reduce と Map Multi-Reduce が提供する耐障害性の詳細について述べる．

3.1　Local Reduce

Local Reduce は，マシンごとに集約処理を行うことで，Reducer の負荷を分散するための機能である．Local Reduce のイメージ図を，**図 6** に示す．Local Reduce では，Combine 処理を複数の Mapper の中間出力に対して適用することで，Reducer に渡す中間出力ファイルのサイズを小さくする．

Local Reduce は，以下の手順で動作する．MapTask 終了後，マスタが Local Reduce の実行を Mapper に割り当てる．Local Reduce が割り当てられた Mapper は，その計算機内に存在する MapTask の中間出力ファイルを読

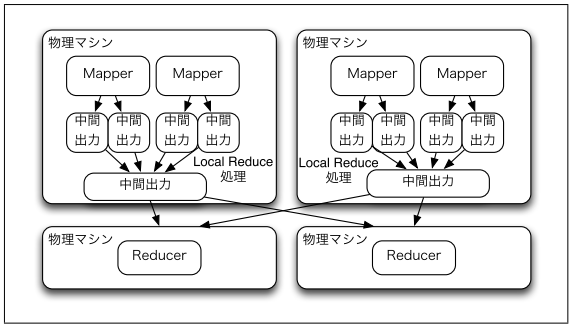

図 6　中間出力の集約を Local Reduce を用いて行ったときの処理フロー

Fig. 6　Processing flow with Local Reduce, node-level aggregation of intermediate outputs.

み込み，Combine 処理を行う．そして，Reducer は Local Reduce の入力となった中間ファイルを転送する代わりに，Combine 処理後の中間ファイルのみを転送する．

Local Reduce を用いることにより，Combiner や in-mapper combining では不可能であった，複数の Mapper 間の計算結果のまとめあげを行うことができる．集約結果の中間ファイルのデータサイズは，集約前の中間ファイルのデータサイズよりも小さくなる．よって，ReduceTask の行う Shuffle 処理のうち，中間ファイルのマージおよびソート処理による IO コストを下げることができ，ReduceTask の高速化を行うことができる．

3.2　耐故障性

Map Multi-Reduce では，上記にあげた MapReduce と同等の耐障害性を担保する．つまり，ジョブを構成するプロセスの一部および計算機の一部が故障しても故障した Task のみをやり直すことでジョブを中断することなく処理を続行することができる．

この機能を実現するためには，Local Reduce 実行時に発生したプロセス故障および計算機故障を正しくハンドリングできる必要がある．

Local Reduce 実行中にプロセスが故障した場合，マスタがプロセス故障を検出し，Local Reduce の入力となっていた中間出力ファイルが，その計算機上で動作する他の Local Reduce の入力ファイルになるようにする．

Local Reduce 実行中に計算機が故障した場合，マスタが計算機故障を検出し，Local Reduce の入力となっていた中間出力ファイルおよび Local Reduce の結果を他の計算機上でやり直すよう，スレーブに指示を出す．

Local Reduce 外での故障については，2.3 節で述べた MapTask/ReduceTask の耐故障性の仕組みと同様に対処する．

4.　Hadoop 上への Map Multi-Reduce の実装

我々は，MapReduce のオープンソース実装である Hadoop 上に提案手法である Map Multi-Reduce を実装した．今回の実装に用いた Hadoop のバージョンは，2012 年 12 月 5 日の Hadoop trunk 版である．要点は，正しい結果を出力する Local Reduce の設計・実装と，オーバヘッドを最小化するための工夫，および耐故障性を担保するためのマスタデーモンの改良である．本章では，これらの設計・実装の方法について述べる．

4.1　実装方針

Local Reduce を実装するにあたって，MapTask/ReduceTask のほかに中間集約を行うためのタスクを追加する方法が考えられる．しかしながら，既存のフレームワークでは，集約用に新たなタスクを追加した場合，オーバヘッドが大きくなるためそれほど性能が上昇しないという報告がある [16]．オーバヘッドの原因は，すべての入力が揃うまで，中間集約を行う処理が開始されず，ブロックする点にある．集約によるブロックが発生すると，以下の影響により性能が低下する可能性がある．

まず，Skew による性能低下の影響を受けやすくなる [8]．Skew とは，MapTask から出力されたキーの偏りによって負荷分散がうまくできなくなってしまうという問題のことである．通常この処理は ReduceTask を行う際に問題になるが，MapTask と ReduceTask の間に中間集約を行うためのタスクを追加することにより，計算機内であってもキーに偏りのある計算機の集約処理の性能が劣化しうる．

次に，MapTask の実行と ReduceTask の結果のコピーを並列で行うことができなくなるため，性能が低下しうる．Hadoop/MapReduce では，処理が終わった MapTask の出力は，ReduceTask が立ち上がった際に即座にコピーすることができる．しかし，中間集約用のタスクを追加して Local Reduce を行う場合，MapTask と ReduceTask の間に処理が挟まるため，すべての MapTask および集約処理が終わるまでコピーの並列化が行えず，性能が低下しうる．上記のように，中間集約用のタスクを追加すると，ブロックの機会が増えてしまうため，集約により IO が減ったとしてもオーバヘッドが大きくなってしまう．

そこで，Map Multi-Reduce では，どの MapTask の中間出力どうしで集約処理を行ってもよいという MapReduce の中間出力の特性に着目し，入力が揃うまで待つのではなく，MapTask 完了時に存在する中間出力ファイルに対して集約処理を行う方式を選択した，この方式では，新たにタスクを追加する必要がなく，MapTask の完了を待つことなく集約処理を開始することができるので，Skew による性能低下を防ぐことができる．また，MapTask から

ReduceTask へのコピーの並列度を担保するために，すべての MapTask を対象とするのではなく，計算機に溜まっている中間出力ファイルの数がしきい値を超えた時点で中間集約を行う．このしきい値を適切に設定することで，コピーの並列度を担保することができる．

4.2 実装の詳細

Hadoop/MapReduce では，MapTask が終了すると，マスタを経由して ReduceTask に MapTask 完了イベントが通知され，ReduceTask への中間出力ファイルのコピーが始まる．一方，Local Reduce を実行する提案手法の場合，MapTask が始まってから Local Reduce による集約処理が開始するまでの間，ReduceTask による中間ファイルのコピーを停止する必要がある．たとえば，中間出力ファイル f_1，f_2，f_3 を用いて Local Reduce を実行し，その中間出力が f_4 になったと仮定する．このとき，集約前の中間出力ファイル f_1，f_2，f_3 と集約処理後の中間出力ファイル f_4 の両方を ReduceTask がコピーをしてしまうと，ReduceTask は間接的に同じ中間出力ファイルを 2 度コピーしてしまうことになるため，計算結果が不正になってしまう．

そこで，正しく値を出力できるように，マスタノードの一部と，MapTask/ReduceTask の一部を改造した．今回の実装では，Local Reduce 処理が終わるまで，ReduceTask による中間出力ファイルのコピー開始を遅らせる．具体的には，終了した MapTask の完了イベントをマスタノードのハッシュマップで一時的に保存し，MapTask 完了イベントの送付を Local Reduce が完了するまで遅延する．ハッシュマップのキーは，スレーブノードの識別子，値は，TaskAttemptCompletionEvent（正常終了した MapTask の ID）のリストである．そして，Local Reduce 処理が完了したら，集約を行っていた MapTask と，集約対象となった MapTask の MapTask 完了イベントを ReduceTask に向けていっせいに送信する．MapTask 完了イベントは，どのような状態でタスクが終了したかを示す State というメンバを保持している．集約処理を行った MapTask は，State を通常成功時と同様 SUCCEEDED に，集約対象になった中間出力ファイルは，State を AGGREGATED にセットする．

ReduceTask は，State が SUCCEEDED になっている TaskAttemptCompletionEvent を受け取ると，通常の動作どおりコピーを開始する．一方で，State が AGGREGATED になっている TaskAttemptCompletionEvent を受け取った場合，コピーをスキップする．こうすることで，間接的に同じ中間出力ファイルが 2 度コピーされることを防ぐ．

4.3 集約処理中のプロセスおよび計算機故障時の動作

MapReduce と同等の耐故障性を担保するには，計算機の故障やプロセスの故障により集約処理を行っていた MapTask が失敗した際に，その入力となっていた中間出力ファイルを再び集約処理の入力として扱えるようにする必要がある．

これを実現するため，本実装ではマスタノードは集約処理待ちの MapTask の中間出力ファイルのリスト（Aggregation Wait List）と，集約処理実行中の中間出力ファイルのリスト（Aggregating List）を管理することで，集約処理が失敗した際に入力となった中間出力ファイルを次の集約処理の入力として再利用できるようにする．

集約処理前にプロセスが故障した場合，MapTask の動作は Aggregation Wait List を変更しないため，通常の MapReduce のエラーハンドリングの実行フローに従って MapTask を再実行させることができる．

集約処理中にプロセスの故障が発生した場合，集約処理の入力となっていた MapTask の中間出力ファイル一覧を Aggregation List から Aggregation Wait List に戻す必要がある．MapTask が失敗すると，Umbilical Protocol を介してマスタに通知が送信される．失敗通知を受け取ったマスタは，Aggregating List を確認し，失敗した MapTask が集約処理を実行していたのであれば，その処理対象となっていた中間出力ファイルをすべて Aggregation Wait List に戻す．その後は，通常の MapReduce のエラーハンドリングの実行フローに従って MapTask を再実行すればよい．なお，プロセス故障の場合，集約処理の入力となっていた MapTask の中間出力ファイルはディスクに永続化されているため，MapTask を再実行する必要はない．

一方で，集約処理中で計算機の故障が発生した場合，Aggretion Wait List と Aggregating List に保存されているその計算機上で実行された MapTask の中間出力ファイルを別の計算機でやり直す必要がある．集約処理中に計算機の故障を検出すると，マスタは故障に対処するためのハンドラを呼び出す．そのハンドラの中で，Aggregation Wait List と Aggregating List に保存されている TaskAttemptCompletionEvent を，タスクの失敗を意味するイベントである TaskAttemptUnsuccessfulCompletionEvent に変換する．後は，通常の MapReduce のエラーハンドリングの実行フローに従って，関連する MapTask を再実行すれば，通常の MapReduce の計算機故障と同等のフローに従って実行される．

5. 評価

提案手法による効果を確認するために実験を行った．実験には**表 1** に示した構成のマシン 8 台を用い，1 台をマスタノード，残りの 7 台を Hadoop のスレーブモードとして動作させた．

実験で用いる入力データは，いずれも Apache Hadoop に標準で含まれる RandomTextWriter を用いてランダム生

成したテキストデータである．提案手法の効果は入力データの特性およびベンチマークプログラムの処理特性に大きく依存する．このため，これらのベンチマークの処理内容およびデータセットごとに，ベンチマークプログラムのプロファイル結果を**表 2** に記述した．

5.1　ベンチマークの種類

実験には，WordCount と単語の共起頻度計算 [10] の 2 種類のベンチマークプログラムを作成し，それらを用いて評価を行った．

WordCount は，与えられた入力テキストを空白文字ごとに区切り，単語数を数え上げるプログラムである．今回実験に用いた WordCount は，提案手法による集約効果を測定するために，2.2 節で述べた in-mapper combining を用いて最適化を行っている．よって，Mapper の実装は図 5 に示したとおりである．また，Reducer は単語を Key，各 Mapper でその Key を数え上げられた結果の配列 Value として reduce 関数を呼び出し，Mapper から渡されてきた単語数を合計して，最終出力とする．

単語の共起頻度計算は，与えられたテキストに連続して存在する単語のペアの回数を数え上げるプログラムである．たとえば，“An apple is red” という内容を保持しているテキストがあった場合，“An apple”，“apple is”，“is red” がそれぞれ 1 回ずつ数え上げられる．上記の例では，探索範囲を広くして “An apple is”，“apple is red” と数え上げることも可能であるが，今回は隣り合っている単語のペアの回数を数え上げることとする．Mapper は，入力テキストを空白文字ごとに区切り，Key として隣接する単語を連結して作成した複合キー，Value として 1 を出力する．Reducer は，単語ペアを Key，各 Mapper でその単語ペアが数え上げられた結果の配列を Value として reduce 関数を呼び出し，Mapper から渡されてきた共起頻度数を足し合わせて最終出力とする．共起頻度計算については，in-mapper combining を用いると組合せ数が爆発的に増加し Out of Memory が発生してプログラムが停止してしまう事象が見られたため，in-mapper combining による最適化は行っていない．

なお，高速化のため，いずれの実験においても Combiner は Reducer と同じクラスを利用し，MapTask ごとに中間集約処理を行うように設定した．

表 1　実験に利用した計算機構成
Table 1　Environment specifications for experiments.

CPU	2.13 GHz x 8 コア
メモリ	8 GB
ネットワーク	1 GbE
OS	Linux カーネル 3.0.11
Apache Hadoop	12 月 5 日の trunk 版

5.2　実験結果

5.2.1　しきい値 V_{th} を変化させたときの効果

4.1 節で述べたように，本稿で述べている実装では，並列度を上げるために計算機に溜まっている中間出力ファイルの数がしきい値に達した際に中間集約を行う．本項では，このしきい値を変化させたときの効果を示す．

まず，1 台あたりで処理する map の数 N_{mpm}（maps-per-machine）は，1 ジョブあたりの map の数 N_{mpj}（maps-per-job）と計算機の台数 $N_{machines}$ を用いて，

$$N_{mpm} \approx \frac{N_{mpj}}{N_{machines}}$$

と表すことができる．Hadoop/MapReduce では，1 つの MapTask あたり，中間出力ファイルを 1 つ作成するため，N_{mpm} は，1 台あたりに生成される中間出力ファイルの数に等しい．そこで，しきい値 V_{th} を

$$V_{th} = ratio * N_{mpm} (0 \leq ratio \leq 1)$$

とし，$ratio$ を変化させることで，どのような変化が得られるかを見る．ただし，$ratio$ が 0 のときは，オリジナルの Hadoop/MapReduce と同等の動きとなる．なお，入力データは WordCount については 100 GB，共起頻度計算については 30 GB のデータを用い，計算機 1 台あたり並列に動作する MapTask は CPU コア数分に相当する 8 つになるようにチューニングを行った．

WordCount の実行結果を**図 7** に，共起頻度計算の実験結果を**図 8** に示す．WordCount の場合に最大で 1.5 倍程度，単語共起頻度計算の場合に最大 1.2 倍程度高速になっている．また，$ratio$ の値を 0.6 以上 0.9 以下にした場合，安

表 2　各実験で利用したデータに対する WordCount と共起頻度計算のプロファイル結果
Table 2　MapReduce profiling result of WordCount and Co-Occurrence.

データ名	Shuffle 量（バイト）	Map 出力のキーの件数	Reduce 出力のキーの件数	Redcue 出力量（バイト）
100 GB（WordCount）	6,606,242,915	226,369,770	64,266,369	2,633,501,250
200 GB（WordCount）	13,210,014,070	452,632,348	105,790,863	4,773,960,396
300 GB（WordCount）	19,968,354,412	678,991,070	144,640,178	6,989,485,732
30 GB（共起頻度計算）	39,571,351,953	231,825,389	74,496,886	2,835,026,176
50 GB（共起頻度計算）	63,323,752,335	370,910,879	115,385,891	4,419,994,267
100 GB（共起頻度計算）	131,930,561,216	772,734,007	229,140,073	8,839,771,183

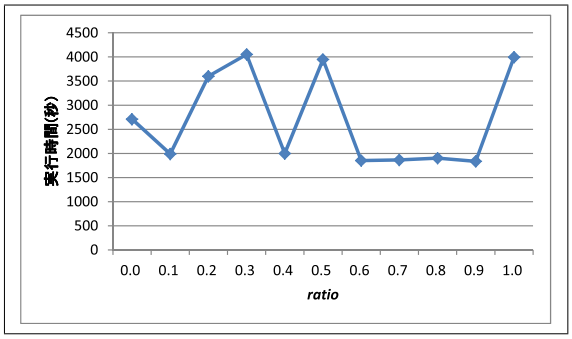

図 7　200 GB の入力データに対して，しきい値 V_{th} 中の変数 *ratio* を 0.1 ずつ変化させながら WordCount を実行したときの処理時間の比較

Fig. 7　Runtimes of WordCount for 200 GB input by changing the *ratio* in V_{th} with 0.1 step.

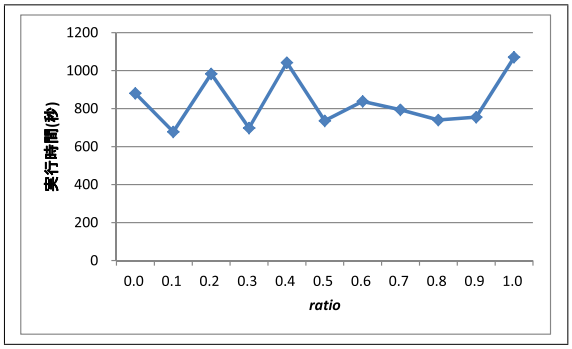

図 8　30 GB の入力データに対して，しきい値 V_{th} 中の変数 *ratio* を 0.1 ずつ変化させながら共起頻度計算を実行したときの処理時間の比較

Fig. 8　Runtimes of Co-Occurrence for 200 GB input by changing the *ratio* in V_{th} with 0.1 step.

定して速度向上している様子が見て取れる．これは，提案手法による計算機ごとの集約と中間出力のデータコピーの並列化がうまく動作しているためと考えられる．なぜなら，*ratio* が 0.6 以上のとき，計算機ごとに処理する MapTask 数に大きな偏りがない限り Local Reduce 処理はたかだか 1 回しか起動せず，Reducer の処理をブロックする可能性が低いためである．

一方で，*ratio* の値が 0.5 以下のときおよび 1.0 のとき，処理性能が安定せず，最大で 45%の性能の劣化が観測された．*ratio* の値が 0.5 以下のとき，Local Reduce 処理は複数回起動される．このため，中間集約による IO コストの削減が Local Reduce 処理による IO コストの増加を下回り，性能低下が生じうる．また，タイミング悪く Reducer が処理を始める直前に Mapper 側で計算機ごとの集約処理を開始してしまい，Reducer への中間出力ファイルのコピーを遅延させてしまうことによる性能低下が生じうる．*ratio* の値が 1.0 のときは，コピーが遅延され，かつ Reducer が処理を始める直前で中間集約を開始してしまう可能性が高い．

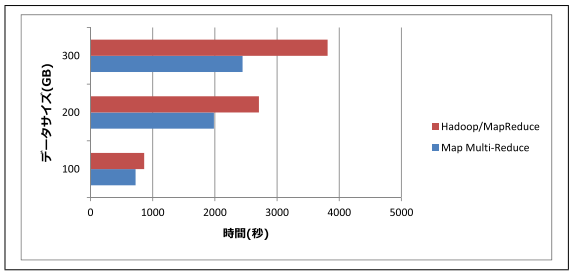

図 9　高並列度環境において，Hadoop/MapReduce と Map Multi-Reduce を用いてデータサイズを変えながら WordCount を実行したときの処理時間の比較

Fig. 9　Runtime comparison of WordCount between Hadoop/MapReduce and Map Multi-Reduce in high parallel environment.

今回観測された処理性能の低下に対する対策について考察する．IO コストの増大については，1 台の計算機上で実行する Local Reduce 処理の回数を制限することにより，IO コストの増大による性能低下を防ぐことができる．また，コピー遅延による性能低下では，Reducer がコピーする予定の中間出力ファイルが残り少なくなった時点で，実行中の Local Reduce 処理を停止し，Reducer へのコピーを開始する方法があげられる．こうすることで，Reducer が処理を始める直前に起動してしまった Local Reduce 処理によるコピーのブロックを回避することができるため，コピー遅延による性能低下を最小限に抑えることができる．これらの対策による性能低下の抑制については，今後の課題とする．

以上の結果より，提案手法は，V_{th} 中の変数 *ratio* を適切に設定することで，一定以上の性能向上が見込めることが分かる．

5.2.2　高並列度環境における性能比較

本実験では，高並列度環境における Map Multi-Reduce の性能を評価するため，計算機 1 台あたり並列に動作する MapTask が CPU コア数と同数である 8 つになるようにチューニングを行い，Hadoop/MapReduce と Map Multi-Reduce 上で WordCount と共起頻度計算を実行した．なお，しきい値 V_{th} 中の変数 *ratio* は，前項の実験結果をふまえ，安定して高速化の効果が観測できた 0.6 を設定した．

まず，WordCount の実験結果を**図 9** に示す．WordCount では，入力ファイルのデータサイズが 300 GB の際に最大で 1.5 倍の性能向上を確認できた．データサイズが増加した際に提案手法の効果が大きくなるのは，Mapper 側で中間出力ファイルを事前集約することにより，データサイズの増加にともなう Reducer の IO 負荷を軽減することができたためだと考えられる．

次に，共起頻度計算の実験結果を**図 10** に示す．共起頻度計算では，データサイズが 100 GB の場合において，約 1.2 倍の高速化が確認できた．Local Reduce が共起頻

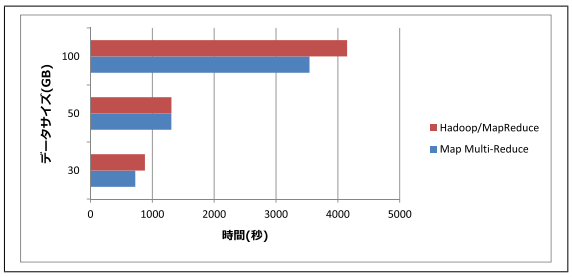

図 10　高並列度環境において，Hadoop/MapReduce と Map Multi-Reduce を用いて，データサイズを変えながら単語の共起頻度計算を実行したときの処理時間の比較

Fig. 10　Runtime comparison of Co-Occurrence between Hadoop/MapReduce and Map Multi-Reduce in high parallel environment.

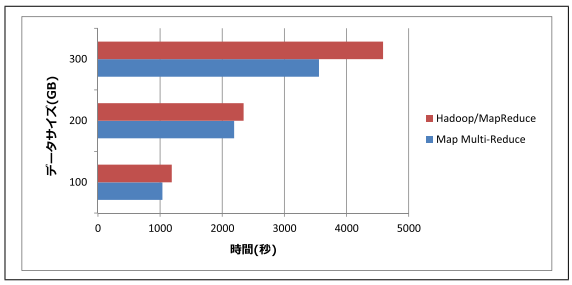

図 11　低並列度環境において，Hadoop/MapReduce と Map Multi-Reduce を用いて，データサイズを変えながら WordCount を動作させたときの処理時間の比較

Fig. 11　Runtime comparison of WordCount between Hadoop/MapReduce and Map Multi-Reduce in low parallel environment.

度計算において WordCount より効果が薄いのは，Shuffle フェイズで書き出される中間出力ファイルの量（表 2 の Shuffle 量）が多いことに起因する．特に，入力データサイズが 100 GB のとき共起頻度計算における中間出力ファイルの量は WordCount の約 7 倍大きい．その大きい中間出力ファイル群に対して Local Reduce を行うため，集約効果よりも Local Reduce により発生する IO のコストが大きくなってしまう．なお，この問題は Combiner でも同様に発生する．この問題に対する対策としては，集約率が低いデータセットにおいて，中間出力ファイルのサイズが一定を超えたら Combiner を起動しないという最適化が考えられるが，今後の課題とする．

以上の結果より，提案手法は，集約率が高くかつ入力ファイルのデータ量が大きい際に，一定以上の性能向上が見込める．

5.2.3　低並列度環境における性能比較

本実験では，低並列度環境における Map Multi-Reduce の性能を評価するため，計算機 1 台あたり並列に動作する MapTask が最大で 2 つになるようにチューニングを行ったうえで WordCount のベンチマークプログラムを Hadoop/MapReduce と Map Multi-Reduce を上で動作させ，処理時間の比較を行った．なお，しきい値 V_{th} 中の変数 *ratio* は 5.2.1 項の実験結果をふまえ，高速化の効果が現れやすい 0.6 に設定した．

実験結果を，**図 11** に示す．データサイズが 100 GB でも，Map Multi-Reduce は Hadoop と比較して同等以上の速度で動作しており，データサイズが 300 GB の場合の WordCount では，ジョブ全体で約 1.3 倍の高速化ができるということを確認できた．しかしながら，高並列度環境と比較すると，その効果は限定的である．この理由は，Local Reduce 処理は MapTask の最後に実行されるため，Local Reduce 処理中は Mapper が新たな MapTask を実行できず，処理の並列度が下がってしまうことが原因と考えられる．

Li らの研究によると，Mapper は CPU リソースを多く利用し，Reducer は IO リソースを多く利用する [9]．この特性を利用し，Local Reduce 中は，MapTask を実行する枠を 1 つ増加させることで，並列度を下げずにオーバヘッドを抑えて Local Reduce を実行する方法が考えられるが，今後の課題とする．

6.　関連研究

分散処理系において集約処理を高速化する手法は，集約を何度も行う多段集約による高速化手法と，細粒度で逐次集約を行うことによる高速化手法の 2 種類に大別される．

Dremel [11] は，カラムナストレージと分散 DB の集約技術を用いて，高速に SQL を実行できる技術である．Dremel で用いられている集約技術には，今回提案した多段集約を含む．しかしながら，Dremel には部分故障した際にリカバリする方法については言及されていない．Dremel の OSS クローンである Impala [4] は，ジョブが失敗した際にユーザが再度クエリを投げることを前提していることから，長い時間実行する処理には向かないと推測される．また，提案手法は，MapReduce 上で集約処理を行っており，Hadoop/MapReduce の API を変更せずに実現可能な点が異なる．

Camdoop [5] は，全ノードがファブリックスイッチでつながっていることを前提に，MapReduce のようなインタフェースで高速に集約処理を行うことができる技術である．計算機間を Cube と見なし，それを最速で実行できるような集約木を作成することで多段集約を高速に行える点が特徴である．これに対して，提案手法はファブリックスイッチが前提ではない点と，MapReduce の上での集約処理を行っているため，Hadoop/MapReduce API を変更せずに利用可能である点が異なる．

Dryad [7] は，MapReduce よりも汎用的なプログラミングモデルを提供することで，より柔軟，高速に処理を行うための技術である．Dryad には，自動的に部分集約するよ

うな仕組みがあるが，ノード間で処理の受け渡しを行うことが前提となっており，Mapper 側で集約処理を行うことは考慮されていない．また，Dryad の場合は，処理のフローをデータフローとして記述する必要があるが，提案手法では MapReduce 以上の処理を記述する必要がない．

Li ら [9] は，Reducer 側で逐次集約を行うことにより Hadoop の性能を高める方法を提案している．Li らは Shuffle された複数の結果を Reducer マージする処理のコストが高いことを指摘しており，この問題を Reducer 側にハッシュマップを持たせて逐次集約することで解決する．これに対して，提案手法は Mapper 側で集約する方法であるため，補完的な方法であるといえる．

分散処理系自体ではなく，分散処理系の上で動作するドメイン固有言語（DSL）において集約処理を高速化する手法も存在する．

Hive [14]/Pig [12] は，MapReduce を簡易に記述するための DSL である．Hive/Pig は DSL を MapReduce プログラムに変換するためのコンパイラを持ち，SQL の groupBy のような処理を行う際に in-mapper combining を行うプログラムを出力するなど，集約処理用の最適化を行っている．提案手法は Hive/Pig の集約処理用の最適化手法の 1 つとして利用可能であり，Hive/Pig 用のコンパイラに提案手法を利用するように改変することにより，集約演算を含む Hive/Pig スクリプトを高速化できる可能性がある．

7. まとめ

本稿では，集約処理を用いて MapReduce 処理を高速化する Map Multi-Reduce について述べた．Map Multi-Reduce では，集約の粒度を適切に設定し，計算機ごとに Combine 処理を行うことで IO コストを抑え，集約処理を高速に行うことができる．また，MapReduce と同等の耐故障性の担保を行うことでユーザが Hadoop/MapReduce との違いを意識することなく利用することが可能である．今後は，中間集約による性能劣化が現れないように MapReduce の処理系を改良する方法を検討していく．また，Hive や Pig といった MapReduce プログラム用の DSL コンパイラに本技術の対応をさせることで，既存の Hive/Pig プログラムがどの程度高速化されるかを検証していく．

参考文献

[1] Amazon Elastic MapReduce, available from ⟨http://aws.amazon.com/jp/elasticmapreduce/⟩.

[2] Apache Hadoop, available from ⟨http://hadoop.apache.org/⟩.

[3] Apache Hadoop Wiki, available from ⟨http://wiki.apache.org/hadoop/PoweredBy⟩.

[4] Cloudera Impala, available from ⟨https://github.com/cloudera/impala⟩.

[5] Costa, P., Donnelly, A., Rowstron, A. and O'Shea, G.: Camdoop: Exploiting in-network aggregation for big data applications, *Proc. 9th USENIX Conference on Networked Systems Design and Implementation*, *NSDI'12*, p.3, USENIX Association (2012) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2228298.2228302⟩.

[6] Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation*, *OSDI'04*, Vol.6, p.10, USENIX Association (2004) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251254.1251264⟩.

[7] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks, *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, *EuroSys '07*, pp.59–72, ACM (online), DOI: 10.1145/1272996.1273005 (2007).

[8] Kwon, Y., Balazinska, M., Howe, B. and Rolia, J.: SkewTune: Mitigating skew in mapreduce applications, *Proc. 2012 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '12*, pp.25–36, ACM (online), DOI: 10.1145/2213836.2213840 (2012).

[9] Li, B., Mazur, E., Diao, Y., McGregor, A. and Shenoy, P.: A platform for scalable one-pass analytics using MapReduce, *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '11*, pp.985–996, ACM (online), DOI: http://doi.acm.org/10.1145/1989323.1989426 (2011).

[10] Lin, J. and Dyer, C.: *Data-Intensive Text Processing with MapReduce*, Morgan and Claypool Publishers (2010).

[11] Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M. and Vassilakis, T.: Dremel: Interactive analysis of web-scale datasets, *Comm. ACM*, Vol.54, No.6, pp.114–123 (online), DOI: 10.1145/1953122.1953148 (2011).

[12] Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A.: Pig latin: A not-so-foreign language for data processing, *Proc. 2008 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '08*, pp.1099–1110, ACM (online), DOI: 10.1145/1376616.1376726 (2008).

[13] Silberstein, A.E., Sears, R., Zhou, W. and Cooper, B.F.: A batch of PNUTS: Experiences connecting cloud batch and serving systems, *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '11*, pp.1101–1112, ACM (online), DOI: 10.1145/1989323.1989441 (2011).

[14] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P. and Murthy, R.: Hive: A warehousing solution over a map-reduce framework, *Proc. VLDB Endow.*, Vol.2, No.2, pp.1626–1629 (2009) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1687553.1687609⟩.

[15] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R. and Liu, H.: Data warehousing and analytics infrastructure at facebook, *Proc. 2010 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '10*, pp.1013–1020, ACM (online), DOI: 10.1145/1807167.1807278 (2010).

[16] Yu, Y., Gunda, P.K. and Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, *SOSP '09*, pp.247–260, ACM (online), DOI: 10.1145/1629575.1629600 (2009).

小沢 健史　（正会員）

2010 年筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻修了．同年日本電信電話（株）に入社．現在，NTT ソフトウェアイノベーションセンタ所属．ACM 会員．

鬼塚 真　（正会員）

1991 年東京工業大学工学部情報工学科卒業．同年日本電信電話（株）入社．2000～2001 年ワシントン州立大学客員研究員．現在，日本電信電話（株）ソフトウェアイノベーションセンタおよび機械学習・データ科学センタ主幹研究員（特別研究員）．電気通信大学客員教授．博士（工学）．ACM，電子情報通信学会，日本データベース学会各会員．

福本 佳史

2009 年慶應大学環境情報学部卒業．同年日本電信電話（株）入社．現在，NTT ソフトウェアイノベーションセンタ所属．日本データベース学会会員．

盛合 敏　（正会員）

1983 年東北大学工学部電気工学科卒業．1988 年同大学院博士課程（情報工学専攻）修了．工学博士．同年日本電信電話（株）入社．2001 年（株）ぷららネットワークス．2004～2013 年日本電信電話（株）．この間，プロトコル処理，インターネットシステム運用技術，分散 OS，リアルタイム OS，セキュア OS，高信頼カーネル，ユビキタスコンピューティング基盤，仮想マシン，大規模分散システムの研究開発等に従事．現在，NTT ソフトウェア（株）．日本ソフトウエア科学会，電子情報通信学会，USENIX 各会員．