複数キーで関連付けられたデータの素集合生成における分散・並列処理の利用

中里克久†

MapReduce は，元々インターネット上のパブリック・サービスなどで蓄積される巨大なデータを対象とした処理を，分散・並列処理を用いて高速化するための技術である．しかし，企業内業務データなど，数 GB～数百 MB 程度のデータを対象にした場合でも，MapReduce による処理が適する場合がある．本稿では，処理対象のデータの中で，何らかの関連のあるデータ群を重複が無いようにグループ化する，すなわち素集合を生成する処理を，MapReduce を用いて効率的に行う方式について述べる．また，本方式を実装したプログラムと，テストデータを用いた性能検証から，同環境における従来方式での MapReduce 適用の場合と比較し，約 3.15 倍の性能改善が確認できた．

1. はじめに

Google 社内で用いられていた分散・並列処理手法である MapReduce の概念が Dean らによって公開され[1]，そのオープンソース実装である Apache Hadoop (以下，Hadoop) [2] が普及したことにより，安価な計算機クラスタを用いた分散・並列処理は以前より身近な技術となった．MapReduce は，元々インターネット上の大規模なサービスで蓄積されるログデータなどの巨大なデータ(Big data，ビッグデータ)を対象とした分析などの処理を，分散・並列処理を用いて高速化するための技術である．しかし，それほど巨大でないデータ，例えば企業内の業務データなどを対象にした場合でも，MapReduce の適用が効果的な場合があり得る．

本稿では，企業内の業務データに対する MapReduce の適用，特に，リレーショナル･データベースが適さない構造のデータを扱うケースにおいて，MapReduce を用いて効率的に処理を行う方法について述べる．

2. MapReduce の概要

MapReduce では，図 1 に示すように，Map 処理と Reduce 処理という 2 種類の処理の組み合わせで一連の処理を行う．Map 処理は，ある入力データから Map データと呼ばれるキーと値からなるデータを生成し，Reduce 処理は，生成された Map データをキーの値によって集約し，集められた Map データに対して何らかの処理を行う．利用者は，Map 処理を行う Mapper と，Reduce 処理を行う Reducer を開発して利用する．実際には，Map 処理と Reduce 処理の間に行われる Shuffle 処理を制御する部品なども開発できるが，基本的には Mapper と Reducer が主な開発対象である．

MapReduce の実行基盤，例えば Hadoop は，この Mapper や Reducer を複数のマシンに分散して多数同時に実行する仕組みを提供している．分散された Mapper や Reducer は，処理対象データのうちの一部の処理をそれぞれ担当し，同時並行で処理を行うため，単一の Mapper や Reducer で処理する場合よりも全体の処理時間が短縮される．

ただし，MapReduce による性能向上効果を得るには，Mapper や Reducer が，他のプロセスや他のマシンの状態に影響されず，独立して実行可能であることが求められる．複数の Mapper 間で共通のデータベースを参照・更新し，排他待ちを行うような実装は分散･並列処理の効果を著しく低下させるため，避けなければならない．

また，Mapper と Reducer の入出力や，Map 処理と Reduce 処理の間の Shuffle 処理において，ディスク I/O やネットワーク I/O が大量に発生する点にも注意が必要である．処理全体に対して Mapper や Reducer の粒度が小さすぎる場合，I/O の影響が相対的に大きくなり，Mapper や Reducer の分散･並列処理の効果が小さくなる．

これらの点に留意する必要はあるものの，比較的容易に，計算機クラスタを活用した高速な処理を実現できる点が MapReduce の強みと言える．

3. 非ビッグデータへの MapReduce の適用

一般に，企業内で扱われる業務データは，インターネット上で広く利用されるサービスで収集されるデータのような，時に数ペタバイトにも及ぶ Web スケールのビッグデータと比較すると規模が小さく，数百メガバイト～数ギガバイト程度であることが多い．MapReduce は元々ビッグデータ向きの技術であり，前節に述べた理由などから，非ビッグデータを対象とした場合の処理効率は他の分散処理手法と比較して必ずしも最高とは言えない．しかし，以下の(1)～(5)に示す理由から，非ビッグデータを対象にする場合にも，MapReduce の適用によってメリットが生じるケースがあると考えられる．

† 株式会社富士通研究所

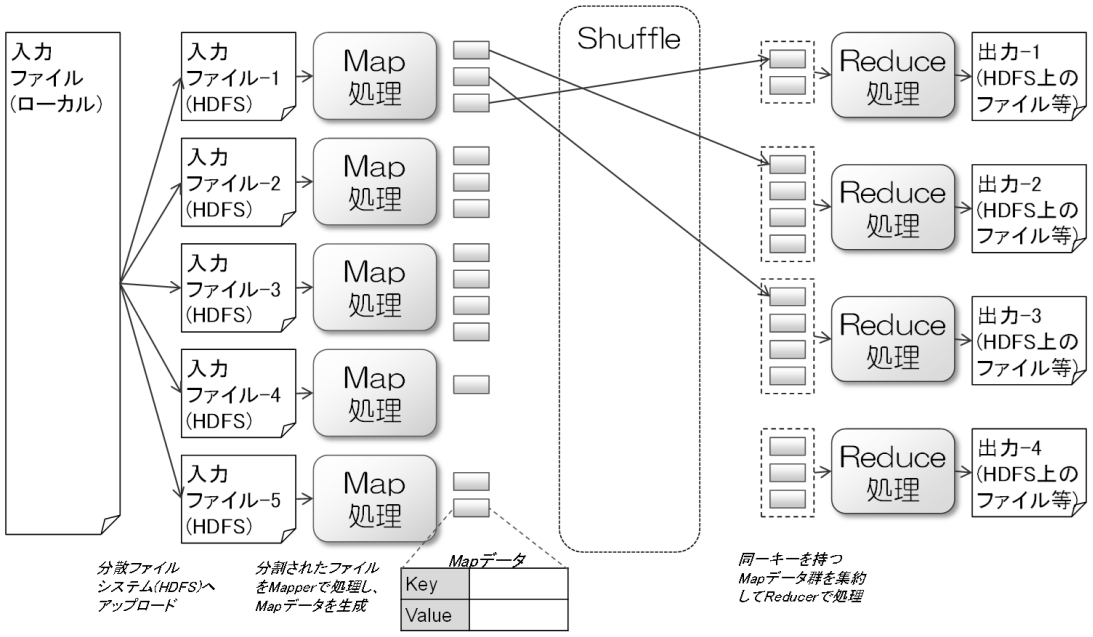

図 1　MapReduce 処理の概要

(1)　マルチコア CPU･マルチサーバの利用の容易性
(2)　I/O ボトルネック，特にディスクアクセスによるボトルネックの解消
(3)　ファイルの分散配置による冗長性の確保
(4)　アドホックな処理への対応
(5)　リレーショナル･データベース(以下，RDB)が適さない非正規化データの分析

(1)から(5)について，Hadoop を MapReduce の実行基盤として利用した場合を例にして説明する．

(1)については，通常，ひとつのまとまった処理を複数の CPU コアを利用することで高速化する場合，マルチスレッドやマルチプロセスを自ら制御するプログラムを開発する必要がある．これには，スレッド間の排他制御など，シングルスレッドのプログラムでは考慮の必要がない様々な問題に対処する設計が必要となり，開発の難易度，開発量が増大する. 更に, 複数サーバを用いた処理分散を行う場合，サーバ間通信や負荷分散の制御など，より多くの考慮が必要となる．

一方，Hadoop を用いる場合，利用者が開発するのは Mapper や Reducer など，データの処理ロジックを実装する部分のみであり，排他制御やサーバ間通信といった機能については開発の必要がない．Mapper で行う Map タスクや Reducer で行う Reduce タスクの，実サーバやプロセスへの割り当ては，Hadoop が自動的に行う．利用者は，各サーバの同時実行タスク数上限などわずかな設定を行うだけで，複数サーバ, および複数の CPU コアを利用することができる．

(2)については，Hadoop では Hadoop Distributed File System(HDFS)と呼ばれる分散ファイルシステムを利用しており，処理対象のデータや中間生成データ，出力データは基本的に HDFS 上に出力される．HDFS は，各処理サーバのディスクを，仮想的にひとつのファイルシステムとして扱う．MapReduce の入力データとして HDFS にアップロードされたファイルは，細かなブロックに分割され，それらのブロックが処理サーバのディスクに分散されて格納される．よって，あるひとつのファイルを MapReduce の処理対象とする場合，実際にはブロックが格納された複数の処理サーバのディスクから並列に読込むことになり，読込みのスループットが高まる．また，Hadoop では，ある処理は可能な限り処理対象のデータのブロックが格納されたサーバ上で行われるように制御されており，分散配置に起因するサーバ間通信による処理遅延を防止している．

(3)は，HDFS 上では，ファイルを構成する個々のブロックは，コピーされて複数の処理サーバのディスクに格納されるため，一部の処理サーバが故障してもファイルへのアクセスが可能な状態を維持できる．これにより，特別な装置なしに RAID1 や 5 などに近い耐障害性が得られ，かつ，ひとつのブロックを格納するディスク数は任意に設定できるなど，より柔軟な面もある．

(4)と(5)は，主に RDB を用いた処理システムと比較した場合の利点である. RDB を利用して実用的な性能を得るためには，適切なテーブル設計を行う必要がある．複雑なテーブル構造が必要となる場合，性能を確保するための設計はより高度なものとなる．ある業務データに対して，アドホックな分析処理を行う場合，例えば，年に 1 回だけ集計処理を行う場合に，それ専用の RDB テーブルを設計する

のは, 費用対効果が低くなってしまう可能性が高い. 一方, MapReduce では，処理サーバ数を増やすことによって処理時間を短縮することが可能であるため，ある程度のスケーラビリティを確保した処理設計さえできれば，従量制の仮想マシンレンタルサービスなどを利用し，短期間だけ大量の仮想マシンを使って処理を高速化する手法が適用可能である．特にアドホックな処理の場合には，設計の最適化によって処理時間短縮を図るより，計算機リソースの物量で処理時間を短縮する方が総合的なコストで有利になる場合が多いと考えられる．また，Hadoop における Hadoop Streaming や Apache Pig [11]のように，スクリプトによって MapReduce を実行させる手段も存在することから，試行を繰り返して分析の内容を改善する使い方にも適している．

(5)については，次節に詳細を示す.

これらの利点の他に，大規模データ(テラバイト，ペタバイト級)に対応可能という MapReduce 本来の利点もあり，ある時点では小規模であった対象データが時間経過により大規模化した場合でも，処理サーバの追加のみで対応可能である.

一方，考えられるデメリットとして，MapReduce 実行基盤は汎用であるため，あるシステム専用に開発されて高度にチューニングされた分散・並列処理制御プログラムと比べた場合，性能が劣る可能性が高い点や，処理を Mapper と Reducer の組み合わせという形で実行しなければならないために，通常のプログラム開発より制約が多くなる点などが考えられる.

前者に関しては，元々，MapReduce が Web インデックス生成のバッチ処理の高速化を目的として開発されたものであり，バッチ処理向きの設計になっていることに留意する必要がある．高速なレスポンスを要求されるアプリケーションへの適用に関しては，[10]などで検討されているが，これは MapReduce の実行基盤を改造する内容であり, 汎用的な MapReduce 実行基盤は現状ではバッチ処理向きだと言える.

後者に関しては，Apache Pig [11]や，Hadoop を SQL に似た文法を用いて利用可能にする Apache Hive [5]など，従来のプログラム開発と近い感覚で MapReduce を利用するための手法が検討されている. しかし, これらを用いた場合, MapReduce の実行が利用者から隠蔽されることになり, MapReduce の使い方の改善により全体の処理時間を短縮する手法が適用困難になるという欠点もある.

これらのメリット，デメリットを考慮し，メリットが上回る場合には, 非ビッグデータに対しても MapReduce を適用する意味があると言える.

4. リレーショナル・データベースが適さないデータと分析の例

ある処理対象のデータの中で，少しでも関連のあるデータ群をグループ化し, グループ間で要素の重複は許さない, すなわち素集合となるように処理する場合を考える．一群に集約したいデータ群すべてが単一の関連キー値を持っている場合，MapReduce を 1 回適用することで全ての素集合を集約可能である．これは[1]で示される通り，MapReduce の最も基本的な機能である.

単一の関連キーが存在しない場合，1 回の MapReduce では集約できない．この場合，図 2 に示す通常の RDB のテーブル構造のような，上位に行くに従って段階的にデータが集約されるモデルであれば, 階層数分の MapReduce を実行すれば集約可能である．このパターンは，RDB におけるテーブルの JOIN に相当し，これを MapReduce で実現するための手法は，[3]で提唱されている.

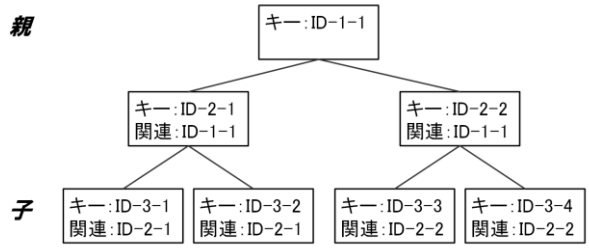

図 2　通常の RDB のテーブル構造

しかし，入力となるデータが RDB 由来のものではない場合，各データの関連はもっと複雑である場合がある．例として，図 3，図 5 のようなパターンが考えられる.

図 3 の例では, データ B の関連キーがデータ A のキーと対応しているが，データ B のキーに重複が許されている. 図 3 の各データをグループ化した場合, 図 4 に示すように,「A1, B1, B2」および,「A2, A3, B3, B4」の 2 グループになる.

データ A (関連先)

キー	値
a001	A1
a002	A2
a003	A3

データ B (関連元)

キー	関連キー	値
b001	a001	B1
b002	a001	B2
b003	a002	B3
b003	a003	B4

図 3　関連元のキー値に重複が許されており，かつ，それらが別の関連先を有しているパターン

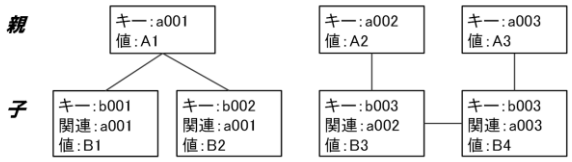

図 4　キー値重複の例

図 5 の例では, データ B の関連キーはデータ A のキーに対応しており，一方，データ A の関連キーはデータ B のキーに対応している．すなわち，双方向の関連付けができる構造になっている．図 5 の例の各データをグループ化した場合，図 6 に示すように，「A1, A2, A3, B1, B2, B3」の 1 グループになる．

このパターンの派生として，同一テーブル内のレコード間で関連が設定されているケースもある．

データ A

キー	関連キー	値
a001	b001	A1
a002	b002	A2
a003	b003	A3

データ B

キー	関連キー	値
b001	a002	B1
b002	a003	B2
b003	--	B3

図 5　相互に関連キーを有しているパターン

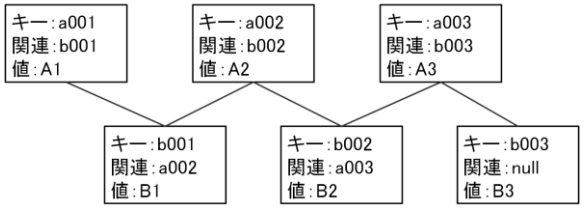

図 6　相互関連の例

図 3，図 5 の例はいずれも，RDB において正規化された適切なテーブル設計をすれば生じないケースであるが，業務分析の対象は，常に RDB のテーブル群とは限らず，別々に設計された複数の DB のデータを混ぜて分析する場合や，ログファイルを含めて分析する場合もあり得る．そのような場合，情報の関連付けは常に自然キーで行うことになるため，キー値の重複などによる複雑なグループ化の要件が発生し得る．例として，複数種の業務 DB や業務ログファイル等の，多岐にわたるソースから収集した業務実績データ群を母集団として，関連のある一連の業務データを集約し，複数の業務システムを経由する業務の流れを通貫して分析する場合などが考えられる[12].

一方，素集合の集約が現実的でない場合もある．例えば，ユーザ数が巨大な SNS におけるフレンドマップの作成を考える．各ユーザが平均数十人規模のフレンドを持っている場合, あるユーザの直接のフレンドは数十人に留まるが，フレンドのフレンドは数十×数十，以降，ある程度のフレンドの重複はあるにせよ，平均フレンド数の n 乗のオーダでグループが広がってゆく．更に，フレンド関係はスモールワールド[6]の性質を持っている可能性が高いと考えられる．平均値より遥かに多いフレンド数を持つユーザが存在し，グループの形成においてハブ的な役割を果たし，全体的には関わりの少ないユーザ群を連結してひとつのグループにまとめる．

このような条件で，最終的にグループが素集合になるまでフレンド関係を辿る処理を繰り返すと，フレンドを一切持たないユーザの他には，1〜数グループ程度の，少数の巨大な素集合が形成される可能性が高い．このような場合には, 素集合を集約するという処理自体にあまり意味が無く，むしろ，各ユーザに対し，数段階だけフレンド関係を辿ったフレンドマップを提供する方が妥当だと考えられる．

このように，業務データ集約のように意味のある素集合になりやすい対象と，フレンドマップ作成のように素集合として収束しにくい対象とが存在するため，見極めた上で適用する必要がある．

5. MapReduce を用いた関連データ群の素集合生成

前節で示したような，複雑に関連付けられた業務データ群から，MapReduce を用いて関連を持つ素集合を生成する方法を示す.

前提として，素集合内で単一となるキー値は存在しない，あるいは，存在するとは限らないものとする．[3]は，MapReduce の仕組みを変えて JOIN 相当の処理を効率的に行うことを提唱しているが，本稿では，汎用性を考慮し，MapReduce の仕組みはそのまま用いることとした.

RDB の JOIN 相当の処理と，前節で課題として提示したパターンの処理との相違は，MapReduce の実行回数が事前に確定するかしないかという点である. JOIN 相当の処理では，JOIN するテーブル数によって MapReduce の実行回数が確定する．通常，2 テーブルの JOIN は 1 回の MapReduce で実行でき，3 テーブルの JOIN は 2 回の MapReduce で実行できる．これは，複数の子がひとつの親に関連付けられ

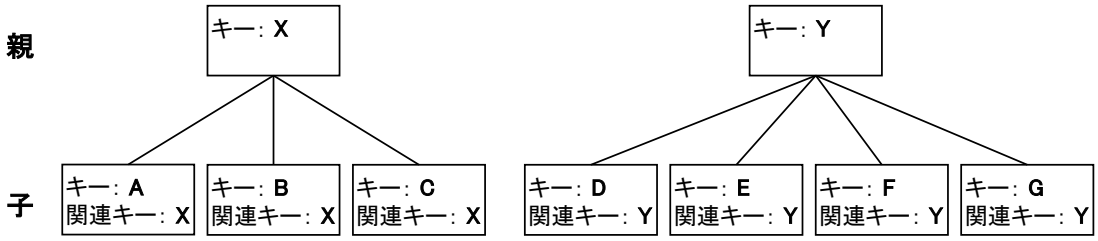

図 7　JOIN パターン

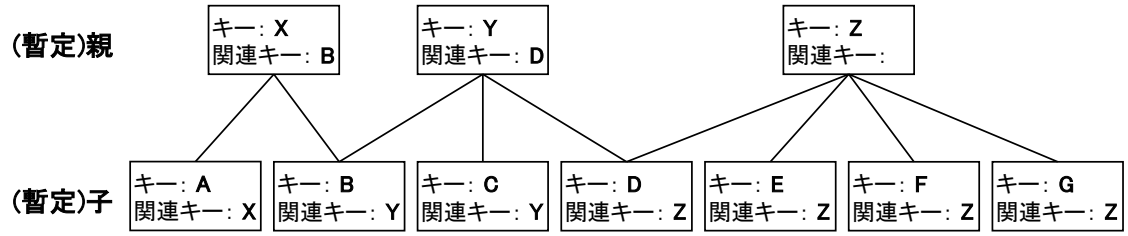

図 8　相互関連パターン

るという規約が守られているため，親のキーで MapReduce による集約を行えば親テーブルと子テーブルが JOIN でき，3 層の場合はその JOIN を 2 回繰り返せば良いからである．一方，前節のパターンでは，親子関係が明確でなく，キー値の関連の仕方により，必要な集約回数が決まり，全データを参照するまでは確定しない．よって，処理前に MapReduce の回数を見積もることもできず，素集合として収束したことが判明するまで MapReduce を繰り返し実行する処理が必要になる．

MapReduce では，複数の処理サーバで完全に並列実行される部分の他，制御サーバによる処理サーバの制御の部分など，処理サーバ数を増やしても処理時間が短縮しない部分も存在する[13]．これを直列実行部と呼ぶとすると，直列実行部に対して並列実行部の処理量が極めて大きい場合，直列実行部の影響は無視できるレベルとなり，処理サーバ数を増やせば増やすほど全体の処理時間が比例して短縮される，すなわち，スケーラビリティの高い状態になる．反対に，直列実行部の割合が増えた場合，スケーラビリティは悪化する．MapReduce の繰り返し実行は，繰り返した分だけ直列部分の処理量が増えることを意味するため，繰り返し回数はなるべく少ない方が良いことになる．

関連データ群の素集合生成処理において，1 回の MapReduce 処理は関連のあるキー値を集約する処理となる．例えば，図 7 のように関連が設定されたデータを集約する場合，親のテーブルではキー値，子のテーブルでは関連キー値を用いて集約すれば，全データが 1 回で集約される．

一方，図 8 ではデータの親子関係は不明確である．子階層の A は親階層の X に関連付けられているが, X は子階層の B に関連付けられており，A と X と B は関連のあるグループとみなせる．しかし, 最初の Mapper を実行する際，A のレコードの処理においては，主キーの A と関連キーの X はレコード内に含まれているが，B は含まれておらず，初回の Mapper において，A が B に関連することを推測するのは不可能である．

考えられる 1 つの方法として, Mapper への入力をレコード単位とせず，ある程度まとまった単位の入力とし，少なくとも関連のあるレコード群は同単位の入力に含まれるようにする方法である．こうすれば，複数レコード間で必要な関連を追跡し，グループを集約することができる．しかし，一般には，関連のあるレコードが入力ファイル上で必ずしも近接した箇所に存在しているとは限らず，どの程度の量を入力としてまとめたら良いかも不明であるため，この方法が有効であるケースは極めて限定される．

考えられるもう 1 つの方法としては, JOIN と同様に子から親への集約を行い，必要に応じて，子の階層に戻って再集約を行う方法である．

上記の例では，まず子の階層のキーで集約する．この場合はキーの重複が無いため，集約してもグループにはならない．従って，関連キーも各 1 つずつだけである．次に，親の階層のキーで集約を行うと，X, Y, Z の 3 グループに分かれる．この時，X のグループには A と X が，Y のグループには B, C, Y が，Z のグループには D, E, F, G, Z が集約される. ここで, X のグループには X の関連キーである B が，また，Y のグループには Y の関連キーである D が，それぞ

れまだ集約に利用されていない未使用関連キーとして検出される．以降，未使用関連キーを用いてグループ化を進めると，表 1 のように，6 回目の集約で図 8 の通りにすべてがひとつのグループに集約される．

前述の通り，MapReduce の繰り返し実行は直列実行部の割合を増やし，分散処理のスケーラビリティを悪化させるため，表 1 の場合よりも MapReduce の繰り返し実行回数を減らす方法を考える．表 1 において A の行と B の行を比較すると，B の行の 1 回目から 4 回目までのキーの変遷を，A の行の 3 回目から 6 回目までが同様に再現していることがわかる．MapReduce では各 Mapper，各 Reducer は独立して実行され，分散処理性能の確保のため，他の Mapper，Reducer の状態は参照すべきでない．よって，B のキーが Z に至るという情報を A の属するグループとは共有できず，このような事態が発生する．ここで，DB などを用いてグループ間で情報を共有するのは，分散処理の効果を損なうため不可であるので，あくまで Mapper や Reducer の入力情報に含められる範囲で情報を伝達すべきである．また，すべての情報をすべての集約グループに伝達すると，その通信量は膨大なものとなることが懸念されるため，ある情報は，それを必要とするグループにのみ伝達されるべきである．

表 1　集約の例

初期状態	1回目	2回目	3回目	4回目	5回目	6回目
A	A	X	B	Y	D	Z
B	B	Y	D	Z	Z	Z
C	C	Y	D	Z	Z	Z
D	D	Z	Z	Z	Z	Z
E	E	Z	Z	Z	Z	Z
F	F	Z	Z	Z	Z	Z
G	G	Z	Z	Z	Z	Z
X	X	X	B	Y	D	Z
Y	Y	Y	D	Z	Z	Z
Z	Z	Z	Z	Z	Z	Z

以上のことから，以下の方針が導かれる．

(1)　関連すると判明したキー群の情報は，そのキー群を構成する各キーを集約キーとする集約グループすべてに伝達する

(2)　あるグループでのキーの集約結果と，そのグループに入力されたキー群のうちの最大のものとが等しくなる場合，その集約でキー数が増えなかったことを意味する．すべてのグループで同様になれば，それ以上集約を繰り返してもキー数は増えないので，集約が収束したと判定できる

(1)については，例えば図 8 の例では，A のキーの初期状態では，A と X が関連することが既知である．従って，A と X からなる関連キーリストを構築し，A を集約キーとし，キーリストを値とする Map と，X を集約キーとし，同じキーリストを値とする Map と，2 つの Map データを生成する．同様に，他のキーの初期状態の関連キーリストも作成し，それぞれの構成要素を集約キーとして Map 処理する．これを受けた Reduce 処理では，それぞれの集約グループに 1～複数の関連キーリストが入力される．例えば，A を集約キーとするグループでは，A が A 以外では出現しないため，A における Map 処理結果である関連キーリスト 1 つのみが入力される．一方，B を集約キーとするグループでは，B の Map 出力の他に，B を関連キーとしている X の Map 出力も入力される．これにより，B における関連キーリストには X が追加される．同様にして各集約グループでの集約を続けると，表 2 のように，4 回目の集約ですべてのグループの関連キーリストが等しくなり，全キーが同じグループに属することが判明する．表 1 と比較すると，MapReduce が 2 回少なくて済む．

処理対象のデータにおいて関連が収束する場合，上記のアプローチによる MapReduce の繰り返し回数の削減の効果が大きく，有効である．逆に，前述のフレンドマップの例のように，事実上関連が収束しない場合，1 回の MapReduce での集約効果は上記アプローチによって高まるものの，結局現実的な回数では収束しないため MapReduce 回数の削減効果はあまり無く，キーリストをコピーして配布する分，処理量が増えるだけになるため，関連が収束しないことが予想される場合は適用しない方が良い．

また，一般に，業務データは業務に関わる重要な情報を保持している．代表的なものとしては，業務が行われた時刻などである．業務群を集約した後，時刻情報などを用いて分析を行うことが業務群集約の主目的であるので，これらの業務属性情報も含めて集約する必要がある．キー以外の属性情報は，多グループに配布する必要は無い．従って，集約によって集められた属性情報のリストは，常にキーリスト内の単一のキー，例えば辞書式に先頭のキーのみに送信する．この属性情報は，最終の集約でそれぞれの素集合毎に 1 箇所に集約する．

表 2　改良した集約の例

初期状態	1回目	2回目	3回目	4回目
A (A,X)	A,X	A,B,X	A,B,C,D,X,Y	A,B,C,D,E,F,G,X,Y,Z
B (B,Y)	B,X,Y	A,B,C,D,X,Y	A,B,C,D,E,F,G,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z
C (C,Y)	C,Y	B,C,D,Y	A,B,C,D,E,F,G,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z
D (D,Z)	D,Y,Z	B,C,D,E,F,G,Y,Z	A,B,C,D,E,F,G,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z
E (E,Z)	E,Z	D,E,F,G,Z	B,C,D,E,F,G,Y,Z	A,B,C,D,E,F,G,X,Y,Z
F (F,Z)	F,Z	D,E,F,G,Z	B,C,D,E,F,G,Y,Z	A,B,C,D,E,F,G,X,Y,Z
G(G,Z)	G,Z	D,E,F,G,Z	B,C,D,E,F,G,Y,Z	A,B,C,D,E,F,G,X,Y,Z
X (B,X)	A,B,X	A,B,X,Y	A,B,C,D,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z
Y (D,Y)	B,C,D,Y	B,C,D,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z	A,B,C,D,E,F,G,X,Y,Z
Z (Z)	D,E,F,G,Z	D,E,F,G,Y,Z	B,C,D,E,F,G,Y,Z	A,B,C,D,E,F,G,X,Y,Z

以下，各段階における処理の詳細と模擬コードを示す．処理対象は任意の形式の業務データだが，前提として，各種の業務的な ID や，業務実行のタイムスタンプ等の属性情報を含んでいるとする．

・前処理

入力の業務データにおいて，業務データ間の関係を示す業務 ID はどれで，どのような関係を持っているかを定義する．業務 ID は複数種用いても良い．

業務 ID には一意な名称を設定する．例えば，複数の入力ファイルに同名の「伝票番号」項目が複数存在しているが，それらは異なる番号を指している場合には，通番を付与したり(伝票番号-1, 伝票番号-2)，より意味が明確化する項目名に修正する(受注伝票番号，発注伝票番号)などして区別する．

RDB のテーブル間の関連付けなどでは，異なる名称のキーによって紐付けられている場合がある．例えば，「受注」テーブルには「受注番号」が存在し，「売上」テーブルには「受注確定番号」が存在するが，「受注番号」と「受注確定番号」の値は同じもので，これらによって両テーブルが関連付けられている場合などである．この場合，「受注番号」に統一するなど，共通した一意名に付け替えて解釈する．この前処理により，MapReduce へ入力するデータは，業務 ID と，業務 ID 以外の属性情報(タイムスタンプ等)から構成される業務データとなる．

・Mapper-1

業務属性情報を複数含めることができる業務属性リストと，業務 ID(業務 ID の種類と ID 値とからなる識別子)を複数含めることができる業務 ID リストとを保持することができる構造(業務オブジェクト)を定義する．

業務データの 1 単位(ログファイルの 1 行など)を入力とし，主キー・関連キー相当の役割で使われている業務 ID 群が全て含まれる業務 ID リストを作成する．また，業務データに含まれる単一の業務属性情報を含む業務属性リストを作成する．

新たに作成した業務 ID リストの要素数分のループを行い，各要素(業務 ID)をキー，業務オブジェクトを値とする Map データを出力する．この時，業務 ID リスト中で辞書式に先頭の業務 ID がキーとなる場合には，作成した業務 ID リストと業務属性リストの両方を含む業務オブジェクトを値とし，それ以外の場合には，業務 ID リストのみを含む業務オブジェクトを値とする．

```
class Mapper-1
method Map(int num, string line)
  SET S to new set of strings
  SET B to new set of objects
  //  a line of input text contains one or
  //  more items of business data
  FOR each
    business_data(
      array[business_id1, business_id2, business_id3, ...],
      object business_property
    )
  in line
    FOR each business_id
    in array[business_id1, business_id2, business_id3, ...]
      ADD business_id to S
    ADD business_property to B

  FOR each string business_id in set S
    IF business_id is first of S
      Emit(business_id, data(S, B))
    ELSE
      Emit(business_id, data(S, null))
```

図 9　Mapper-1 の擬似コード

・Reducer-1

Mapper が作成した Map データを，キーによってグループ化したものを入力とする．

各 Map データの値(業務オブジェクト)から，業務 ID リストと業務属性リストを抽出する．そして，各業務 ID リストの要素数を保持する．また，新たな業務 ID リスト，および業務属性リストを作成し，入力の業務 ID リストおよび業務属性リストから内容をコピーする．その際，業務 ID リスト内では値の重複が生じないようにする. すべての入力を処理したら，新たな業務オブジェクトを 2 種類作成する．ひとつは，作成した業務 ID リストと業務属性リストを含む業務オブジェクト(属性を含む業務オブジェクト)であり，他方は，作成した業務 ID リストのみを含み，業務属性リストは空の業務オブジェクト(ID のみの業務オブジェクト)である．また，入力の業務 ID リストの最大要素数と，新たな業務 ID リストの最大要素数とを比較し，要素数が増えたかどうかを判定する．

増えた場合，MapReduce で標準的に規定されたカウンタ機能を用い，キー数増加カウンタをインクリメントする．また，新たに作成した業務 ID リストの要素数分のループを行い，各要素(業務 ID)をキーとし，ID のみの業務オブジェクトを値としてシーケンスファイルに出力する. その際，辞書式に判定して先頭となる業務 ID がキーになる場合のみ，属性を含む業務オブジェクトを値とする．

増えていない場合, その Reduce 処理グループのキーをキーとし，属性を含む業務オブジェクトを値としてシーケンスファイルに出力する．

```
class Reducer-1
method Reduce(string id, array[data1, data2, ...])
  SET Sg to new set of strings
  SET Bg to new set of objects
  SET max to 0
  FOR each data(set S, set B) in array[data1, data2, ...]
    FOR each string business_id in set S
      ADD business_id to Sg
    FOR each object business_property in set B
      ADD business_data to Bg
    IF max < size of S
      SET max to size of S
  IF max < size of Sg
    INCREMENT MasterCounter
    FOR each string business_id in set Sg
      IF business_id is first of Sg
        Emit(business_id, data(Sg, Bg))
      ELSE
        Emit(business_id, data(Sg, null))
  ELSE
    Emit(id, data(Sg, Bg))
```

図 10　Reducer-1 の擬似コード

・MapReduce 制御部

Reducer の終了後，MapReduce を起動した部分に制御が戻る．ここを MapReduce 制御部と呼ぶこととする．制御部では，直前の Reducer においてキー数増加カウンタがインクリメントされたかどうかをチェックし，されていたら 2 回目の MapReduce を実行する．されていなければ，最終処理を行う．

関連が収束しないデータへの対策を行う場合，MapReduce の実行回数を確認し，しきい値に達していた場合に強制的に最終処理へ移行させる．

・Mapper-2

直前の Reducer が出力したシーケンスファイルを入力とする．特に処理は行わず，シーケンスファイル内のキーと値をそのまま用いた Map データを作成して出力する．

・Reducer-2

Reducer-1 と同様の処理を行う．

以降，MapReduce 制御部に戻り，処理を繰り返す.

・最終処理-Mapper

直前の Reducer が出力したシーケンスファイルを入力とする. 値の業務オブジェクトから業務 ID リストを抽出し，その内容から一意文字列を作成する．例えば，全要素を辞書順に並べて連結するなどすると一意文字列を作成できる. その文字列をキーとし，業務オブジェクトを値とする Map データを作成し，出力する．

・最終処理-Reducer

Mapper が作成した Map データを，キーでグループ化したものを入力とする．

新たな業務属性リストを作成し, 各 Map データから業務オブジェクトを抽出して，それに含まれる業務属性リストから，業務属性情報を抽出して新たな業務属性リストにコピーする. 業務 ID 群については, Mapper での処理に応じ，キーから復元しても良いし, 業務オブジェクトから業務 ID リストを抽出して新たな業務 ID リストを集約しても良い. 各 Reducer で集計したものが，関連のある業務データ群となり，Reducer 間で，業務 ID および業務属性情報の重複は生じない.

上記は基本的な実装モデルであり，例えば 2 回目の Reducer において, 1 回目の結果を配布した相手には差分のみ配布する，といった性能向上のための実装上の工夫を追加することも可能である．

概要

ファイル	行数	主キー	関連キー	関連先	主キー種別数	関連キー種別数
1st.csv	10000000	ID1	なし	なし	9516194	なし
2nd.csv	10000000	ID2	ID1	1st.csv, ID1	9515898	6124854
3rd.csv	10000000	ID3	ID2	2nd.csv, ID2	9515961	6124022

備考

各主キーはランダム生成。主キー値の重複も認める
各関連キーは、関連先の主キー値から1種類を選択して決定
主キーの値と行の順番は無関係。同じ主キー値が離れた行に存在する可能性もある

1st.csv

ID1	Date
3749282	2011/5/10 11:09:16
7183154	2011/5/12 09:59:30
1726028	2011/5/13 14:11:53
:	:

2nd.csv

ID2	ID1	Date
2071844	3749282	2011/5/12 09:31:48
9101738	3749282	2011/5/12 10:27:39
4821396	1726028	2011/5/13 11:15:26
:	:	:

3rd.csv

ID3	ID2	Date
5128177	2071844	2011/5/14 10:20:34
9123827	4821396	2011/5/17 14:30:07
9123827	7622991	2011/5/18 15:44:39
:	:	:

図 11　テストデータの概要

6. 性能測定と評価

前節の方式を実装したプログラムを開発し, 図 11 に概要を示すテストデータを利用して性能測定を行った．このテストデータは 3 種類のデータファイルからなり, 3rd と 2nd, および 2nd と 1st が異なる関連キーで関連付けられている．また，1st, 2nd, 3rd それぞれのキー値は重複が許されている．データの規模は，それぞれ 1000 万行ずつ，計 3000 万行であり，関連キーによって 8548053 の素集合に分類される構造である．データ容量は，3 種合計で約 1GB である．

実験環境は，ブレードサーバ上に構築した Hadoop クラスタであり，Hadoop のバージョンは 0.20.203 である．CPU に関しては，処理用のスレーブサーバ群は最大で 42 個の CPU コアを用いており，この他にクラスタ制御用のマスタサーバが 2 個の CPU コアを用いている．メモリは総計で 96GB，OS は Ubuntu Linux の 10.10 を用いた．

図 12 は，表 2 のようにキーリストをコピーしながら集約する本稿の新方式と，表 1 のように関連キーをひとつずつ使用して集約する従来の旧方式の性能比較結果である．新方式では 8 回の MapReduce で集約が完了し, 処理時間が 34 分だったのに対し，旧方式では 22 回の MapReduce が必要となり，107 分を要した．両者の MapReduce の回数の比は 2.75 で, 処理時間の比が約 3.15 となっていることから，新方式による MapReduce の回数削減が処理時間短縮に寄与したと考えられる．

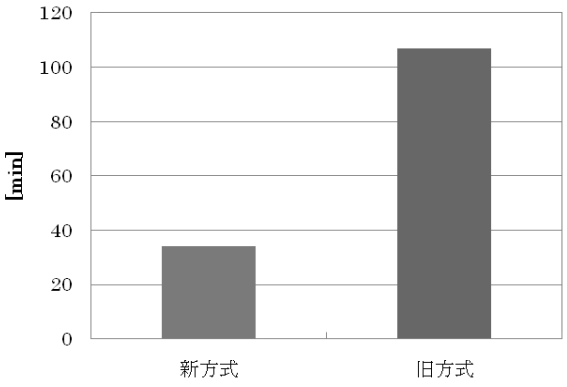

図 12　新方式と旧方式の処理時間比較

また，図 13 は，新方式で図 11 のデータを処理する場合の, 処理に利用する CPU コア数が処理時間に与える影響を示したグラフである．横軸は利用したコア数，縦軸は，1 コア利用時の処理時間を基準にした処理性能の倍率(Speedup)を示している．実験環境の最大コア数である 42 コア利用時の性能倍率の実測値は約 31 倍であった．図 13 のグラフ形状を見る限り，42 コアの段階では性能倍率の頭打ちの傾向は見られないことから，コア数を増やすことで絶対的な処理時間の更なる短縮は可能であると予測できる．

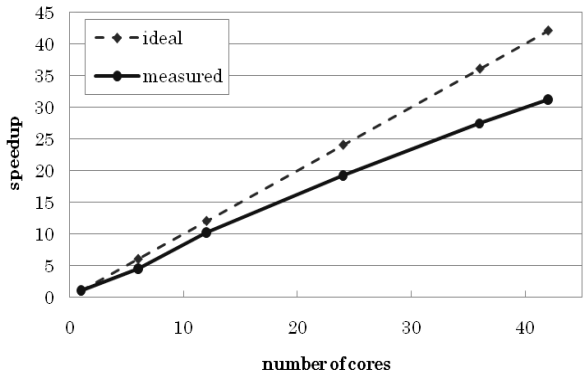

図 13　コア数と性能倍率の関係

図 14 は，処理対象のデータの規模を変えた場合の，42 コア利用時の 1 コア比性能倍率の変化を示したグラフである．理想的には，すべてのデータ規模で 42 倍となるべきだが，実際には，300 万行では約 10 倍，1500 万行では約 26 倍，3000 万行では約 31 倍となっており，データ規模に応じて性能倍率が上がっている．データの構造など，規模以外の条件は等しく，また，Mapper や Reducer のロジックは同一であることを考えると，対象データの規模に依らない Hadoop 利用時のオーバーヘッドが存在し，性能倍率を押し下げている可能性が高い．よって，データ規模が大きければ大きいほど，処理時間全体に占めるオーバーヘッドの比率が小さくなり，理想の性能倍率である 42 倍に近付くと推測できる．ただし，オーバーヘッドの絶対量は減らないため，42 倍に到達することはないと考えられ，図 14 もその傾向を示している．この図から，更にデータ規模を増やしていった場合の 42 コア時の性能倍率の上限は，35 倍程度になると予測できる．

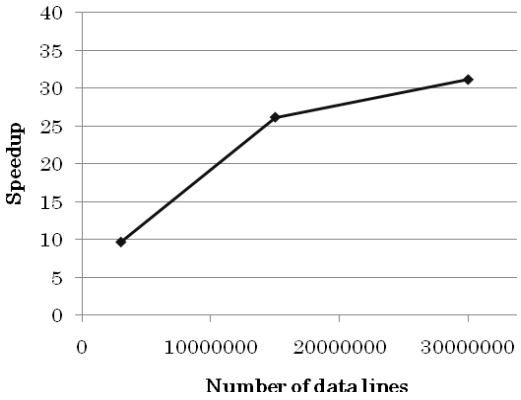

図 14　データ規模と性能倍率の関係

なお，42 コア時の理想的な性能倍率である 42 倍に達していない原因はいくつか考えられるが，主な要因のひとつとして，実験環境におけるハードディスク構成が考えられる．実験環境全体でハードディスクは計 8 本と，CPU コア数よりもかなり少ない構成であった．テストデータの容量は 1GB 程度なので処理全体に占めるディスク I/O の比重はさほど高くないと考えられるが，各コアで実行される Mapper や Reducer の入出力先であるディスクの本数が少ないことで，その部分は並列化が不十分な直列的実行部分となった可能性がある．[13]では，性能倍率(Speedup)の計算式を以下のように示している．

$$Speedup = \frac{1}{S + \frac{P}{N}}$$

Speedup: 性能倍率
P: 処理における並列実行部分の比率
S: 処理における直列実行部分の比率(=1-P)
N: 並列度(=CPU コア数)

42 コア時に 42 倍の性能倍率を得るには，S が 0 である必要があるが，S が 1%でも存在すると，42 コア時の性能倍率は約 29.8 倍まで下落してしまう．性能倍率が 31 倍の時の S は，約 0.87%である．また，図 14 において，42 コア時でもデータ規模により性能倍率が変化していることから，S は処理方式で固定の値にはならず，データの規模や内容，システムのハード構成など各種の要因に影響を受ける値だと考えられる．ハードディスクの本数不足という S の増加要因がある状況下でも S は約 0.87%に収まっており，この測定時の分散処理環境の利用効率は妥当な水準であると考えられる．

以上より，本稿で示した MapReduce 適用の新方式は，3000 万行のテストデータの素集合生成において従来の旧方式と比較して約 3.15 倍と明確に処理性能が向上した．また，スケーラビリティの良さを示す性能倍率は，実験環境のディスク構成に問題がある状態でも 42 コア利用時に 1 コア比約 31 倍を示しており，処理対象のデータ規模が約 1GB と比較的小規模な条件下でも，MapReduce と本稿の新方式により，分散処理環境を有効活用できていると評価できる．

7. 他の処理手法との比較

MapReduce と本稿の新方式の組み合わせと，他の処理方式との比較を検討する．まず，典型的な例として，分散処理を行わないシングルスレッド動作の処理プログラムと，RDB との組み合わせによる処理方式が比較対象として考えられる．図 11 の構造を持つ 3000 万行のデータを本稿の新方式を用いて 42 コアで処理する際の処理時間は

2042.031 秒である．仮に，これと同等の処理性能をシングルスレッドの処理プログラムと RDB との組み合わせで実現しようとする場合，アルゴリズムによって増減はあるものの，3000 万レコード以上の読み書きは最低限発生すると考えられるため，約 14691TPS で更新を含む処理を実行する必要が生じる．単一のクライアントに対して 14691TPS の処理性能を実現するには，1 件あたりの処理時間は約 68 μ 秒となり，一般的に用いられるディスクにデータを不揮発化するタイプの RDB における更新を含む処理のレスポンス時間としては実現が難しいレベルの性能と考えられる．

また，他の比較対象として，分散環境上で実行される RDB であるパラレル RDB も考えられる．本稿ではパラレル RDB について試行していないが，[7]では MapReduce とパラレル RDB との比較で，RDB へのデータ入力とクエリ部分を分けて考慮し，主にクエリ部分の性能比較でパラレル RDB の優位性を論じている．前述の通り，MapReduce は常に最高の分散処理効率を示すわけではないため，RDB が向く条件下ではパラレル RDB の方が性能が高い場合はあり得る．ただし，4 節で示した通り，複雑な関係を持つデータにおける素集合生成は RDB に不向きな処理であるため，[7]に示されるほどパラレル RDB が優位になることは考えにくい．また，[7]ではデータ入力部分については MapReduce の方が高速であることも示されており，形式が不揃いな各種企業内データをアドホックに分析するようなケースでは，この部分の比重が高くなると予想されるため，性能差は縮小，あるいは逆転する可能性もあると考えられる．また，パラレル RDB と比較した場合の Hadoop のセットアップの容易さも利点として挙げられており，RDB のテーブル設計の難しさと合わせて，環境構築の容易性の点から MapReduce を選択する場合もあり得るだろう．

その他，Bulk Synchronous Parallel(BSP，バルク同期並列)の実行基盤である Apache Hama[14]など，非 RDB かつ非 MapReduce の分散処理手法も比較対象として考えられるが，これらとの比較・検証は今後の検討課題である．一般に，これらは MapReduce および Hadoop の流行後に，MapReduce では性能が出ないケースに対処するために開発されたものであるため，処理内容によっては MapReduce より高速になる場合もある．ただし，MapReduce および Hadoop ほど普及しているものは現時点で存在せず，RDB とも MapReduce とも異なる処理手法の学習コストも必要になることから，当面は MapReduce の方が汎用の分散処理基盤として有力であると考える．

8. まとめ

本稿では，企業内業務ログなどの非ビッグデータに対し，MapReduce を適用する場合の利点を示し，特に RDB に不向きな構造を持つデータを対象として，関連するデータ群の素集合を生成する場合に，MapReduce を用いて効率的に実行する方法について説明した．また，その方法を実装したプログラムを用いて性能測定を行った結果，1GB 程度の小規模なデータが対象であっても，MapReduce を用いた従来方式と比較して処理性能は約 3.15 倍と，明確な優位性があることが確認できた．また，MapReduce を用いない他の方式との比較について考察し，非分散処理に対しては性能の面で，他の分散処理手法に対しては主に環境構築の容易性や汎用性の面で優位であることを示した．

参考文献

1) Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)
2) Apache Hadoop project. http://hadoop.apache.org/
3) Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In Proceedings of ACM SIGMOD, 2007.
4) Jimmy Lin and Chris Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010.
5) Apache Hive project. http://hive.apache.org/
6) Stanley Milgram. The Small-World Problem. Psychology Today, vol.1, 61-67 (1967).
7) Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In Proceedings of SIGMOD, 2009.
8) Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Maden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? CACM, 53(1), 2010.
9) Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy and Russell Sears. MapReduce Online. In NSDI, 2010.
10) Jimmy Lin, Anand Bahety, Shravya Konda and Samantha Mahindrakar. Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework. HCIL Technical Report HCIL-2009-01, Jan. 2009.
11) Apache Pig project. http://pig.apache.org/
12) Keisuke Yano, Yoshihide Nomura, and Tsuyoshi Kanai. A Practical Approach to Automated Business Process Discovery. Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2013 17th IEEE International, 2013.
13) G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, pages 483-485, April 1967.
14) Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An Effcient Matrix Computation with the MapReduce Framework. In Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, 2010.