Vol.2013-OS-126 No.9
2013/7/31

不良レコード処理の変更による 目的に応じた MapReduce の効率化

山本 幸一[1]　櫻井 孝平[1]　清水 裕亮[1]　山根 智[1]

概要：近年増加し続けるデータを処理するために大規模データ処理フレームワーク MapReduce があるが，その内部動作において，不良な入力レコードに対応するための処理には時間がかかり，対象データや目的によってはボトルネックとなってしまう．本研究では，不良レコードが発生した際の処理を分析し，システムを目的やデータに即した処理に変更することで MapReduce の効率化を実現する．大規模分散処理基盤 Apache Hadoop をテスト対象として実験を行い，不良レコード処理を変更することで効率化が可能であることを示した．

1. はじめに

近年の爆発的なデータ量の増加には驚くべきものがあり，年間に作成および複製されるデータ量は 2．8 ゼッタバイト（2．8 兆ギガバイト）とも言われている [4]．このような大規模なデータ量に対応するために，スケールアウトによる処理能力の向上およびそれを実現する並列分散処理技術が注目されている．開発者は MapReduce に代表されるプログラミングモデルに従うことで比較的簡単に分散ソフトウェアを実装でき，Apache Hadoop（以下 Hadoop）[2] に代表されるような分散基盤を利用することでスレッドの制御，ロック，スケジューリング等基本ソフトの操作が不要になる．

分散処理基盤を利用した分散システムにおいて，非常に大規模なデータを高速に処理するためには，様々なチューニングが必要である．本研究では検証対象の分散処理基盤として，代表的な実装である Hadoop を採用する．Hadoop を効率よくスケールアウトするために必要なチューニングのひとつに，開発者による不良レコードの対処があげられる．Hadoop では，大量のレコード（分散ファイルシステムの Key と Value のペア）を読み書きするジョブを複数のタスク（分散処理の単位）に分割して実行するが，レコードの不整合による例外が発生する場合，必ずタスクは失敗する．タスクは失敗後も何度か再試行されるが一定回数を超えると，最終的にジョブ全体が失敗してしまう．タスクの失敗を阻止するには開発者が定義するクラス内で複雑な例外処理の記述が必要であるが，そのために記述が煩雑になってしまい，欠陥の原因となる．そもそも不良レコードの発生がサードパーティ製ライブラリでソースコードが公開されていないものに起因するような場合必ずしもソースコードを修正できるとは限らない．この問題の対処を分散処理基盤で行う必要がある．

不良レコードへの対策として，Hadoop ではスキップモードと呼ばれるものが実装されているが，データの形式や処理の内容に関わらず，タスクが何度も再実行される仕様であり，さらにレコードを 1 つずつ確かめながら実行するため時間がかかり，入力データによってはボトルネックとなってしまう．対象データや目的に応じた不良レコードへの対処法が必要である．

本研究では，大規模データに対する MapReduce 処理において，開発者が定義するクラスを変更することなく目的に則した不良レコードの処理を行う手法を提案し，ジョブの実行を効率化する．非常に膨大なデータを対象に統計的分析をおこなう場合などでレコードの不整合が発見された場合，動作後の調査のためにレコード情報のみを出力し，タスクを途中停止させることなく処理を継続する．実際に 6 台のマシンを用いて Hadoop の環境を構築し，提案手法のコードを加えて実験を行った．実行中に故意に例外を発生させ，その時の実行時間，出力を通常の処理と比較する．

以下，第 2 章に検証対象とする分散基盤である Hadoop と内部の機構および関連研究について述べる．第 3 章提案手法について述べる．第 4 章で，提案手法を実現するコードを実装し，疑似的にエラーを発生させた場合の結果について述べる．第 5 章で結果について考察し，本論文をまとめる

[1] 金沢大学
Kanazawa University

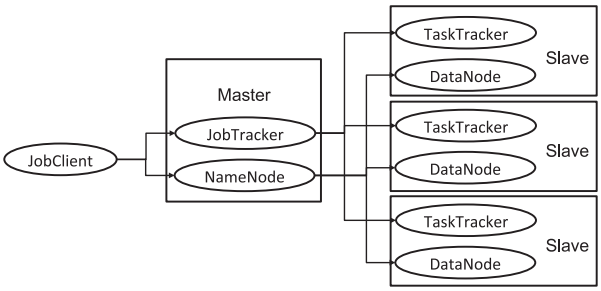

図 1 Hadoop クラスタのアーキテクチャ

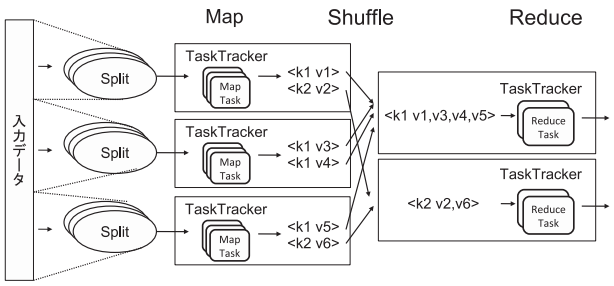

図 2 MapReduce のアーキテクチャ

2. 背景技術と関連研究

2.1 MapReduce

MapReduce は Google が 2004 年に [1] で発表した分散処理モデルで, その概念はクラスタコンピューティングの代表的プログラミングモデルとなっている．MapReduce タスクは大きく分けて以下の 3 つのフェイズから成り立っている．Map フェイズでは入力データであるレコード（Key-Value の組）に対して並列計算を行い，中間レコードを生成する．これを受け取った Shuffle フェイズでは中間データごとに Key によるソートを行い，同一 Key でまとめたレコードを出力する．Reduce フェイズでは同一 Key を持つデータを受け取り，これに対し集約処理をし，出力が最終的なタスクとしての出力を行う．

各フェイズは互いに独立であるため分散環境で同期することなく並列に処理が可能である．

2.2 Apache Hadoop

ApacheHadoop（以下 Hadoop）は，Google が 2003 年に発表した「Google File System（GFS)」[3] や先に述べた MapReduce などをもとにつくられた，大規模データを扱うための代表的なオープンソース実装の並列分散処理基盤である．大きく分けると分散ファイルシステムである「HadoopDistributedFileSystem（以下 HDFS)」とデータ処理フレームワーク「HadoopMapReduce（以下 MapReduce)」から成り立っている．データの分散を HDFS が，処理の分散を MapReduce がそれぞれ担うことで冗長性を向上し，性能のスケールアウトを実現している．開発者は Map フェイズと Reduce フェイズでの処理のみを定義することでジョブが実行可能となるため，分散並列処理を比較的簡単に行うことができる．

Hadoop はマスタスレーブ型のアーキテクチャで, 複数のコンピュータの制御や管理を行うマスターノードと, 制御されるコンピュータであるスレイブノードから成り立っており，NameNode，DataNode，JobTracker，TaskTracker という JavaVM のプロセスが各ノードでデーモンとして動作する（**図 1**）．例として HDFS のマスタである NameNode は，ファイルの位置情報を保持・管理しており，実際の大きなファイルは分割され, HDFS のスレイブである DataNode に格納される．

Hadoop では，MapReduce におけるマスタである JobTracker が一つのジョブを複数の小さなタスク に分割し，複数のスレイブである TaskTracker に割当て, 各々の TaskTracker が実際の処理を担う．まず入力データを Split と呼ばれる単位に分割し MapReduce タスクへの入力とする．Split は Map フェイズで Split 内のレコードごとに開発者が定義する map メソッドの入力となり中間レコードが出力される．partition メソッド，combiner メソッドによってソート，集約された中間レコードは Reduc e フェイズにて読み込み，処理され出力される（**図 2**）．この際 Hadoop では入力レコードの不整合対策として SkippingBadRecord クラスによるスキップモードと呼ばれる処理が用意されている．

3. 大規模データにおける MapReduce の効率化手法

3.1 不良レコード

Hadoop を効率よく動作させるためには様々なチューニングが必要となるが，開発者が対応しなければならない問題の一つに不良な入力レコードへの対応が挙げられる．主に開発者の記述ミスによって発生し，あるレコードの処理中にデータの不整合による例外が発生した場合，そのレコードは不良レコードとされ必ずタスクは失敗する．タスクは失敗後,．失敗した原因がハードウェアの問題か，その他の原因なのか判断できないため JobTracker によって再試行される．この際処理を行っていた TaskTracker 及び分割され直前までに処理されていたレコードすべてが破棄され新たにタスクを起動する仕様のため再実行の分時間がかかってしまう．タスクはその後何度か再試行されるが，一定回数を超えると 最終的にジョブ全体が失敗してしまう．不良レコードによる速度低下とジョブの失敗を防ぐためには開発者が定義するクラス内で複雑な例外処理の記述が必要となる．そのような記述は煩雑になり更なるエラーの原因になりやすい．ソースコードはエラーを回避するためにもシンプルな形式が望ましい，また Hadoop の内部動作は複雑であるため，障害によって影響を受け速度低下の原因

となる箇所の特定は簡単ではなく，デバッグも手間がかかる．そもそもサードパーティ製ライブラリでソースコードが公開されていないものに起因するのであれば訂正は困難である．そこで，開発者の関与しない段階で不良レコードへの対策を講じる必要性がある．

3.2 スキップモード

Hadoop には，Map フェイズにて入力データを処理する際，一定数の不良な入力レコードをスキップするオプションが用意されている．このオプションが有効になっている場合，一定回数 タスク が失敗した後に「スキップモード」に入り，アルゴリズム（**図 3**）にしたがって Map フェイズが行われる（**図 4**）．record$_i$ は次に処理される,Split 中の i 番目のレコードである. スキップモード中は map メソッドのたびに *ReportAsNormalRecord*(record$_i$) において処理したレコードが TaskTracker に報告される（9，10 行目）．タスクが失敗した際，TaskTracker には例外が発生する直前までのレコードが報告されているため（図 4（a)），その次のレコードが不良レコード（badrecords）であるとわかる（13 行目）．再試行の際に不良レコードとしたレコードにおける map メソッドをスキップすることで不良レコード（図 4（b)）を回避できる（4，5，6 行目）．その後別の不良レコード（図 4（c)）が出現した場合そこでもタスクが失敗するため再び最初から処理をし直し該当するすべてのレコードをスキップする．これを繰り返すことで，すべての不良レコードがスキップできる．

確実に処理を実行させることを第一とした実装であり，また一定数の再試行後にスキップモードとなるため，明白に不良な入力レコードが原因の例外発生である時のみスキップできる上 *WriteBadRecordInfo*(record$_i$) において，不良レコードの情報を HDFS に書き込むことで (5 行目) 動作後の確認や問題の原因究明のために不良レコードのデータを利用することができるのが特徴である．正常なレコード群を含む Split 全体が再試行されてしまうため HTML のような不安定なデータ形式を対象とするような処理の場合，何度も再試行されてしまい性能が低下する．MapReduce タスクへの入力である Split は負荷分散とタスク生成のオーバーヘッドとの兼ね合いでおおよそ HDFS のブロックサイズと等しくなるように設定するものであるが，このブロックサイズは通常 128MB かそれ以上の大きな値にするのが一般的である．1 つの不良レコードのたびに Split 全体を何度も再試行するのは，全体的に見て効率的とは言えない．

4. 不良レコード処理の放棄

非常に大規模なデータに対する統計的処理を行う場合を考える．大量のデータを処理するにあたって全てのデータが期待通りの値であるという仮定は，例外が発生しジョブが失敗した場合の時間的損失を考えてもできず，不良レコードへ対処が必要となる．通常，スキップモードでは例外が発生するとタスクは途中停止し再試行されるが，不良レコードのたびに何度もタスクを再試行するのは，１つのレコードに費やす労力に見合わず，目的に即した処理とは言えない．タスクを途中停止，再試行させることなく処理を継続させ，高速化をはかるほうが効率的であると言える．

そこで，レコードの不整合が発生した場合，動作後の調査のために不良レコードに関する情報だけを残し，再試行処理行うことなく次のレコードの処理に移る手法を提案する．これによりタスクの再試行におけるオーバーヘッド及び処理するデータ量を抑え，ジョブ全体としての効率化を実現する．この手法により具体的に変更を加えるクラスは，org.apache.hadoop.mapreduce.Mapper クラス及び継承・周辺クラスである．開発者定義である map 処理を呼び出しているクラスであり，開発者が意識する必要はない．

提案手法ではアルゴリズム（**図 5**）にしたがって Map フェイズが行われる（図 4）．通常は入力データを分割したものである Split ファイルから key，value の組であるレ

```
1  i ← 0
2  badrecords ← ∅
3  while i < |Split| do
4      if i ∈ badrecords then
5          WriteBadRecordInfo(record_i)
6          i ← i + 1
7      else
8          try
9              map(record_i)
10             ReportAsNormalRecord(record_i)
11             i ← i + 1
12         except
13             badrecords ← badrecords & {i}
14             i ← 0
15         end
16     end if
17 end
```

図 3 アルゴリズム：スキップモード

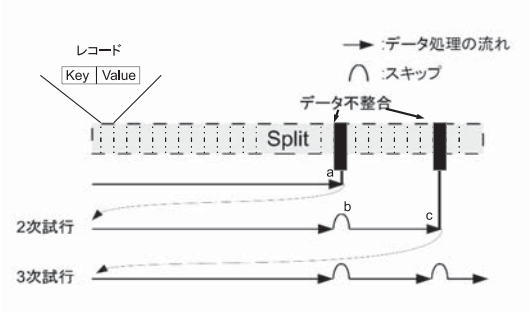

図 4 スキップモード概要図

```
1  i ← 0
2  while i < |Split| do
3      try
4          map(record_i)
5          i ← i + 1
6      except
7          WriteBadRecordInfo(record_i)
8          i ← i + 1
9      end
10 end
```

図 5 アルゴリズム：提案手法

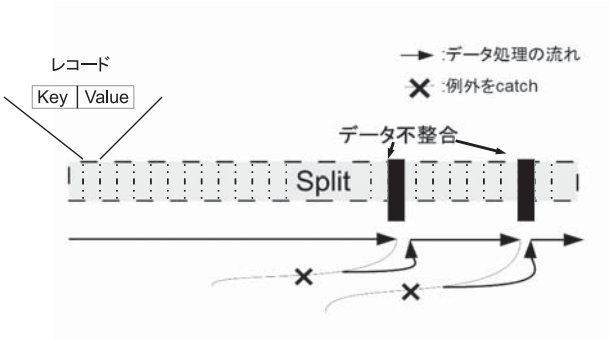

図 6 提案手法概要図

コードをとりだし，map 処理 (4 行目)．という工程を Split の終端まで繰り返している．map 処理中に例外が発生した場合，この処理は途中で停止してしまい，例外を throw しタスクが失敗してしまう．ここで TaskTracker への例外を catch してしまい，後の分析のためにその時のレコード情報のみを出力 (6，7 行目)，そのまま次のレコードの処理を継続するよう変更する (**図 6**)．またレコード情報を出力するのに必要な情報を得るためにメソッドの引数等も変更する．この手法ではレコードの不整合によらない例外発生の場合でも処理されないため，データが一部失われる可能性があるが，今回目的とするような非常に膨大なデータを対象に統計的分析を行うアプリケーションなどでは，効率化が優先され許容される場合が多いと考える．なお，HDFS の容量不足等書き込み時のエラー，レプリカ作成時のデータ欠損などによるエラーは HDFS 側で対応されるためこれには含まれない．

5. 検証実験

検証対象の大規模な分散処理システムとして Hadoop クラスタを構築，不良レコードへの処理を変更した状態でジョブを実行し，その有効性を検証する．

5.1 検証システム構築

検証対象とする Hadoop クラスタを 6 台の同仕様マシンを用いて構築する．

- Hadoop1.03
- Java（TM） SE Runtime Environment1.7.0.09

表 1 Hadoop クラスタ構成

ホスト名	IP	デーモン
master	192.168.1.10	JobTracker
		NameNode
slave1	192.168.1.11	TaskTracker
		DataNode
slave2	192.168.1.12	TaskTracker
		DataNode
slave3	192.168.1.13	TaskTracker
		DataNode
slave4	192.168.1.14	TaskTracker
		DataNode
slave5	192.168.1.15	TaskTracker
		DataNode

プロセッサ	動作周波数	コア数	メモリ
Intel(R) Core(TM) i5-3470	3.20GHz	4	8GB

図 7 構成マシン性能

表 2 例外を発生させない場合の出力結果および実行時間

手法	起動したMapTask数	入力レコード数	実行時間(ms)
スキップモード	2	1000	32834
	2	10000	31862
	2	100000	32772
	2	1000000	38867
	16	10000000	114177
	150	100000000	992252
提案手法	2	1000	31873
	2	10000	31971
	2	100000	32829
	2	1000000	38801
	16	10000000	116201
	150	100000000	947436
デフォルト	2	1000	32807
	2	10000	32900
	2	100000	32727
	2	1000000	38833
	16	10000000	116142
	150	100000000	986703

- Linux CentOS 6.3

Hadoop クラスタの構成を**表 1** に，マシン 1 台の性能を**図 7** に示す．

5.2 検証結果および考察

5.2.1 例外が発生しない場合

まず，本検証では MapReduce における処理手順を一部変更しているため，不良レコードが発生しない環境でジョブを動作させた場合の結果を取得しデフォルト設定およびスキップモードでの結果と比較することで本手法による影響を検証する．入力ファイルは 100kB，1MB，10MB，100MB，1GB，10GB のランダムデータで，1 レコードあたりが 100B であるものを使用する．また MapReduce ジョブは，Hadoop でベンチマークに使用される TeraSort を使用する．これは，大量データをソートするサンプルである．以上の条件で Hadoop を動作させた際の，各手法に対する起動タスク数・入力レコード数・実行時間を**表 2** に，入力レコード数に対する実行時間のグラフを**図 8** に示す．

図 8 から，不良レコードを発生させない場合は，各手法において実行時間に大きな差は見られないことがわかる．また出力結果も全ての手法で等しかった．したがって不良

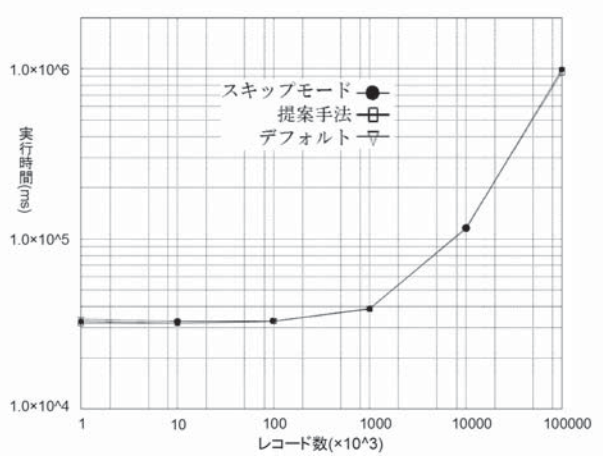

図 8 例外を発生させない場合のレコード数に対する実行時間

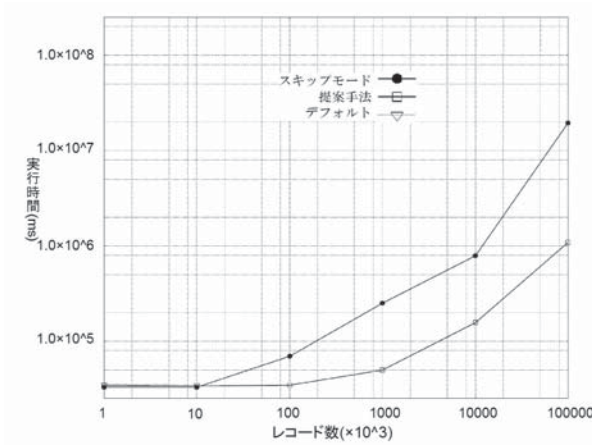

図 9 例外を発生させた場合のレコード数に対する実行時間

レコードを変更することによる影響はほとんどないと言える．全体的に実行時間が入力レコード数と比例していないのは，Reduce 処理を行うために，対象となるノードに全レコードの情報を送信する必要があるため，トラフィックが集中し速度低下につながったからだと考えられる．

5.2.2 例外が発生する場合

MapReduce ジョブの実行中，故意に例外を thorw することで疑似的にレコードの不整合を発生させた時の実行時間及び出力結果を取得し，デフォルト設定およびスキップモードでの結果と比較することで本手法の有効性を検証する．検証環境は前項の検証と同様のものを使用する．また例外の発生確率は各レコードあたり 10^{-5} とする．またこの時

以上の条件で動作させた際の，各手法に対する起動タスク数および不良レコード数・出力されたレコード数・実行時間を**表 3** に，入力レコード数に対する実行時間のグラフを**図 9** に示す．

デフォルトの設定ではレコード数が 10^5 以上の場合 Job 全体が失敗し出力結果が得られていない．実行ログを確認したところジョブの途中特定のタスクが複数回失敗・再試行を繰り返した後にジョブ全体が失敗していた．レコード数が 10^5 以上の場合ジョブ中に例外が発生するようになり，タスクが失敗したためジョブ全体も失敗し出力結果を得られなくなったと考えられる．

スキップモードではデフォルト設定と同様に実行途中に例外が発生したにもかかわらず，出力データは全体のレコード数から不良レコード数を引いたレコード数で処理が行われており，不良レコード以外のレコードの出力には成功しているといえる．実行ログを確認したところ 規定回数の失敗後スキップモードが起動し失敗の原因となっていたレコードのスキップに成功したたその後タスクは失敗せず結果出力に成功している．しかし，対象となるデータ量が増えるに従い加速度的に実行時間が増加してしまっている．これは，例外の発生により再試行されるタスク数の増加に伴うオーバヘッドおよび再試行される Split のデータ量増加のために，全体として処理されるデータ量が増加しているためと考えられる．特に同一の Split 中に複数の不良レコードがある場合，TaskTracker へレコード情報を報告するためにただでさえ動作の遅いスキップモードで何度も実行することになるため実行時間が大幅に増加する．

提案手法による不良レコードへの処理変更後での実行では，スキップモードと同様に例外が発生したレコード以外の出力に成功した，また図 9 からもわかるように，スキップモードに比べ実行時間が短く．レコード数が 10^8 の場合ではスキップモードと比較し 900%以上の高速化に成功している．これは再試行およびスキップモードによるレコード情報の送受信が発生しない分処理するデータ量が減少したためと考えられる．また，この実行時間の差は入力データ形式が不安定でレコードの不整合が発生しやすく，またデータ量が増加するほど再試行の回数が増加し，大きくなると考えられる．

6. 評価とまとめ

大規模なデータを Hadoop などの分散システムでを用いて高速に処理するためには様々なチューニングが必要である．本研究ではそのひとつである不良レコードの対処を例に挙げ，不良レコードへの内部処理を変更することで，開発者定義クラスの記述を変更することなく，大規模データに対する統計的処理などを目的とした場合のジョブ実行の効率化を実現した．

本提案手法により，開発者が関与しない段階で問題に対処することで，開発者が複雑な例外処理を記述する必要がなくなり，それによるプログラムエラーの発生抑制が期待できる．また，レコードの不整合により例外が発生した場合，再試行の処理を放棄し次のレコードへの処理に移ることにより，処理するデータ量を抑え，従来の手法よりも高速化に成功した．この際一部データが損失する可能性があるが，Hadoop の基本的な活用法であるような非常に大規

表 3　例外を発生させた場合の出力結果および実行時間

手法	起動したMapTask数	入力レコード数	不良レコード数	出力レコード数	実行時間(ms)
スキップモード	2	1000	0	1000	32858
	2	10000	0	10000	32892
	2	100000	1	99999	69921
	2	1000000	8	999992	250322
	16	10000000	95	9999905	788025
	150	100000000	1015	99998985	19565784
提案手法	2	1000	0	1000	34241
	2	10000	0	10000	34091
	2	100000	1	99999	34567
	2	1000000	8	999992	49716
	16	10000000	95	9999905	157022
	150	100000000	1015	99998985	1091311
デフォルト	2	1000	0	1000	35368
	2	10000	0	10000	34677
	2	100000	不明、ただし 1以上	Job失敗	Job失敗
	2	1000000	不明、ただし 1以上	Job失敗	Job失敗
	16	10000000	不明、ただし 1以上	Job失敗	Job失敗
	150	100000000	不明、ただし 1以上	Job失敗	Job失敗

模な，データを対象に統計的分析を行うアプリケーションなどでは，効率化が優先され許容される場合が多いと考える．今後の課題として，より一般的なプロパティに対しての開発者の負担軽減およびジョブの効率化，例として不良レコードのみを抽出してデータクレンジングを行った後に再試行するなどがある．またそれらの自動判別による処理の切り替えなどが考えられる．

参考文献

[1] Jffrey Dean and S anjay. Ghemawat : MapReduce : Simplified Data Proceesing on Large Clusters OSDI '04, pp137-150 : Sixth Symposium on Operation System Design and Implementation（2004）
[2] Apache Hadoop : http://hadoop. apache. org/
[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung:The Google File System, Google（2003）
[4] EMC:Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East :IDC Digital Universe Study（2012）
[5] Andrew. S. Tanenbaum, Maarten. van. Steen : DistributedSystems: Principles and Paradigms, Prentice-Hall（2002）（アンドリュー．S．タネンバウム，マールティン．ファン．スティーン，水野忠則・宮西洋太郎・鈴木健二・西山智・佐藤文明・東野輝夫（訳）：分散システム原理とパラダイム，O'REILLY（2003））
[6] 太田一樹・金田有二 : Hadoop 調査報告書 : Preferred Infrastructure, NTT レゾナント（2008）
[7] Tom White : Hadoop:The Difinitive Guide, O'Rilly Media,（2009）（Tom White, 玉川竜司・兼田聖士（訳）（2010）: Hadoop, O'REILLY）
[8] 太田一樹・下垣徹・山下真一・猿田浩輔・藤井達朗（著），濱野賢一朗（監修）: Hadoop 徹底入門 オープンソース分散処理環境の構築，翔泳社（2011）