THE IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS (JAPANESE EDITION)

電子情報通信学会 論文誌 D 情報・システム

VOL. J97-D NO. 4
APRIL 2014

情報・システムソサイエティ

一般社団法人 電子情報通信学会
THE INFORMATION AND SYSTEMS SOCIETY
THE INSTITUTE OF ELECTRONICS, INFORMATION AND COMMUNICATION ENGINEERS

Hadoopをはじめとする並列データ処理系へのアウトオブオーダ型実行方式の適用とその有効性の検証

山田　浩之[†a)]　　合田　和生[††b)]　　喜連川　優[††,†††c)]

Application of Out-of-Order Execution to Parallel Data Processing Systems and Evaluation of Its Effectiveness

Hiroyuki YAMADA[†a)], Kazuo GODA[††b)], and Masaru KITSUREGAWA[††,†††c)]

あらまし　並列データ処理系であるHadoopにおいては，近年Hiveをはじめとする上位層ソフトウェアの充実が見られ，当該処理系は大規模データの解析基盤として広く用いられるようになりつつある．同時に，元来のMapReduceなるデータ処理に特化し対象データの全走査を前提とするという設計を見直し，データ処理の効率性を高めるべく，索引やパーティショニング等の各種のデータベース技術を取り込む方向性が見られる．本論文では，Hadoopをはじめとする並列データ処理系において，関係データベースエンジンで試みられているアウトオブオーダ型実行方式を拡張して適用することにより，データ処理の一層の高速化を目指す．アウトオブオーダ型実行方式を適用することにより，並列データ処理系の各々の計算機は，並列データ処理の実行時にタスク分解を行い，分解されたタスクにおいて自らの二次記憶並びにネットワークを介した他の計算機の二次記憶への入出力を行い，入出力の完了に伴い関連する演算を実行する．すなわち，並列データ処理系全体の入出力を非同期化する．データインテンシブな並列データ処理においては，入出力に性能が律速されることが多く，当該入出力を非同期化することにより，従来型の処理系に比して，特にデータセット空間の一部のデータを対象とするデータ処理において，飛躍的な高速化が期待される．本論文では，著者らが試作を行ったHadoopをベースとするアウトオブオーダ型並列データ処理系Hadooodeの構成法を明らかにするとともに，20台の計算機からなるクラスタマシンにおいて当該試作を用いて行った性能評価実験を示し，その有効性を明らかにする．

キーワード　アウトオブオーダ型実行，非同期入出力，並列データ処理，並列問合せ処理，大規模データ解析，Hadoop

1. まえがき

情報技術の発展とその普及によって，人々の経済活動や社会活動が膨大なデジタル情報として蓄積されるようになりつつあり，所謂ビッグデータ[1]と称される当該情報は，企業における経営の効率化や新たな社会的サービスの創出などに活用され始めている．インターネット通信販売やインターネットオークションサービスを手がける米eBayは，利用者の行動履歴等からなる40PB以上のデータベースを構築し，当該データベースの解析により，利用者の関心に合った商品ページの提供を試みている[2]．また，風力タービンの製造・販売を行うデンマークのVestas Wind Systemsは，気象，潮汐，地理空間データ，衛星写真などからなるPB規模のデータベースを構築し，当該データベースの解析結果を発電所の設置戦略の立案に活かす試みを行っている[3]．今後，更に多くの応用が生み出されていくものと考えられる．

ビッグデータ解析のためのシステム基盤としては，一つの技術潮流として，Hadoop[5]に代表される並列

† 東京大学大学院情報理工学系研究科，東京都
Graduate School of Information Science and Technology, The University of Tokyo, 7–3–1 Hongo, Bunkyo-ku, Tokyo, 113–8656 Japan
†† 東京大学生産技術研究所，東京都
Institute of Industrial Science, The University of Tokyo, 4–6–1 Komaba, Meguro-ku, Tokyo, 153–8505 Japan
††† 国立情報学研究所，東京都
National Institute of Informatics, 2–1–2 Hitotsubashi, Chiyoda-ku, Tokyo, 101–8430 Japan
a) E-mail: hiroyuki@tkl.iis.u-tokyo.ac.jp
b) E-mail: kgoda@tkl.iis.u-tokyo.ac.jp
c) E-mail: kitsure@tkl.iis.u-tokyo.ac.jp

データ処理系の広範な利用が見られる．近年，Hadoop においては，Hive を始めとする上位層ソフトウェア [6], [7] の拡充が見られ，また，Cloudera 等によるソフトウェアディストリビューションの整備が進展しており，Hadoop は企業等における本格的なビッグデータ解析に用いられるようになってきている．

他方で，Hadoop は，当初の並列データ処理を MapReduce [4] なる単純なプログラミングモデルに限定することにより，プログラマによる並列処理の実装労力を軽減できる点が注目され，広く利用されるに至ったが，MapReduce 型のデータ処理は基本的に対象となるデータセット空間の全走査を前提としている．ETL 用途等においては妥当な選択と言えるが，データセット空間の一部のデータを対象とするデータ処理も少なからず存在し，ビッグデータ時代においては，毎回全体空間を走査することはより困難になると思われることから，その比重は今後増していく可能性がある．

Hadoop においても，データセット空間の全体を毎回走査するという元来の設計を見直す試みが行われ始めている．HadoopDB [8], [9] や Hadoop++ [10] 等においては，二次索引，索引構成表やパーティショニング等の構造化データへのアクセス技法をはじめとするデータベースシステムで培われた技法の Hadoop への取り込みが行われている．当該潮流は急速に進展しつつあり，Hadoop をベースとしつつ多様なデータベース技術を組み込んだエンタープライズ向けの並列データ処理系が次々と発表されるに至っている [11]〜[13]．

本論文では，Hadoop をはじめとする並列データ処理系を対象として，喜連川らが関係データベースシステムにおいて考案した**アウトオブオーダ型実行方式** [14] を拡張して適用することにより，データ処理の一層の高速化を目指す．アウトオブオーダ型実行方式の下では，並列データ処理系の各々の計算機は，データ処理の実行時にデータの入出力を要する都度に，動的にタスク分解を行い，分解されたタスクにおいて入出力を発行し，また，関連する演算を実行する．この際，並列データ処理系においては，入出力の対象となる二次記憶が計算機自身が管理するものであれば計算機において当該二次記憶に発行し，他の計算機が管理するものであればネットワークを介して他の計算機を経由にして当該二次記憶に発行する．データインテンシブな並列データ処理においては，入出力に性能が律速されることが多く，当該入出力を非同期化することにより，従来型の処理系に比して，特にデータセット空間の一部のデータを対象とするデータ処理において，飛躍的な高速化が期待される．すなわち，これまでの研究 [14], [15] では，1 台の共有メモリ型の計算機を対象としてアウトオブオーダ型実行方式が明らかにしてきたが，本論文では，当該実行方式を複数台の計算機から構成される無共有型（Shared nothing architecture）のクラスタマシンへと適用すべく拡張する．

本論文では，著者らが試作を行った Hadoop をベースとする**アウトオブオーダ型並列データ処理系 Hadooode** の構成法を明らかにするとともに，ビッグデータ解析のための高密度実装サーバとして広く利用されつつある 24 台の磁気ディスクドライブを備える計算機 20 台から構成したクラスタマシンにおける性能評価実験を示し，当該試作の有効性を明らかにする．著者らは文献 [16] において，Hadooode の潜在的な有効性を少数の問合せを用いて明らかにしてきたが，本論文では上位層の Hive を含めた Hadooode の構成法を明らかにするとともに，精緻な性能評価実験を示すことにより，その有効性を明らかにする．著者らの知る限り，アウトオブオーダ型実行方式による並列データ処理系を提案し，その有効性を明らかにする研究はこれまで他に行われていない．

本論文の構成は次のとおりである．**2.** では，並列データ処理系におけるアウトオブオーダ型実行方式を示し，その潜在的な有効性を議論する．**3.** では，著者らが Hadooode の試作において実現した当該実行方式による MapReduce 型データ処理並びに Hive 問合せ処理を示す．**4.** では，解析タスクデータセット並びに TPC-H データセットを用いた Hadooode の性能評価実験を示し，その有効性を論じる．**5.** では関連研究を述べ，**6.** で本論文をまとめる．

2. 並列データ処理のアウトオブオーダ型実行

並列データ処理は，高速なネットワークによって接続された複数の計算機を用いて，与えられたデータ処理を並列化して実行することにより，その高速化を目指すものである．これまで多様なシステムアーキテクチャが検討されてきたが，多くの並列データ処理系では，事前にデータ処理の対象となるデータセットを分割して計算機の備える二次記憶に格納しておき(注1)，

(注1)：提案方式はストレージネットワーク等による共有型の二次記憶へも適用可能である．

データ処理のジョブが与えられると，当該ジョブを分割若しくは複製して計算機に割り当て，各計算機において割り当てられたジョブを実行する．各々の計算機では，割り当てられた各ジョブにおいて，データの入出力命令を発行して，当該入出力の完了を待って演算を実行することを，対象の全てのデータを処理し終えるまで繰り返す．入出力と演算は事前にプログラムされた順序に基づき行われることから，当該方式を**インオーダ型の実行方式**と称する．

これに対して，本論文では，並列データ処理系における**アウトオブオーダ型実行方式**を提案する．当該実行方式では，各計算機においてジョブの実行時に，新たな入出力を発行する必要が生じると，都度にタスク分解を行い，分解された並行実行可能なタスク上で当該入出力とそれにかかる演算を実行する．すなわち，ジョブにおいては，タスク分解によって，その入出力が非同期的に発行されることとなり，入出力の完了とともに演算が駆動されることとなる．並列データ処理系のストレージアーキテクチャに依存するが，入出力は発行元の計算機の管理する二次記憶に対してのみならず，他の計算機が管理する二次記憶に対して行う必要がある場合があり，この場合，計算機間のネットワークを介した通信も同様にタスク上で実行される．入出力と演算が多段で行われるような場合，このような手順に従って，タスク分解が再帰的に行われることとなり，実行時に，データセットとジョブの実行論理が許す限りにおいて，多数の入出力が並行して発行され，また，多数の演算が並行して実行されることとなる．計算機が有する資源は有限であり，実際には，同時に実行可能なタスクの数や同時に実行可能な入出力や通信の数はこれによっても制約されるものの，例えば，後述する実験環境における 2U 程度のきょう体に収まるサーバ型計算機においては，少なくとも 1,000 個程度のタスクの同時実行が可能である．マイクロプロセッサ技術の潮流としては，プロセッサあたりのコア数は着実に増加する傾向にあり，ビッグデータブームに牽引され，サーバ型計算機が備える二次記憶装置の集積密度は従来に比して高まっている．アウトオブオーダ型のソフトウェア実行方式は，これらの資源を効率的に活用することにより，インオーダ型の実行方式に比して，データ処理のスループットの大幅な向上を目指すものである．

並列データ処理系においては，1980 年代におけるハッシュクラスタリングアルゴリズム [17] の開発以降，アルゴリズムの工夫によって基本的なファイルアクセスをシーケンシャルアクセスとすることにより，多数の磁気ディスクスピンドルを並列駆動してスループットを高めるアプローチが広く採用されてきた．当時から磁気ディスクドライブのレイテンシンの削減は年率 5%程度に留まっている一方で，記録密度の向上は指数関数的に増大しており [18]，今日においても同様のアイデアは広くみられる．Hadoop もその一端をなしており，MapReduce なる単純なデータ処理モデルに基づき，ファイルアクセスとしては全体走査を基本としている．当然のことながら，このような全体走査は必ずしも常に効率的であるわけではない．一般に，アクセスパスという観点では，ファイルの多くの部分のデータに対してデータ処理を行う場合，すなわち，データ処理の選択性が低い場合は，ファイル全体を読み出す全体走査が有効である．反対に，ファイルのごく一部の部分のデータに対してデータ処理を行う場合，すなわち，データ処理の選択性が高い場合においては，索引を用いたアクセスが有効である．後者の索引アクセスについては，商用の並列データ処理系等 [19] において実装されているものの，著者の知る限り，その有効性に関しては文献 [20]〜[22] 等で議論がされるに留まってきており，特に解析的なデータ処理については，十分に活用されてはきていなかったと見られる．しかしながら，先述のビッグデータブームにより，並列データ処理系が抱えるデータセットの量は格段に増加する傾向にあり，当該傾向においては，ファイル全体を読み出す全体走査が可能な機会は限定的とならざるを得ず，また，エビデンスドリブンな意思決定支援を実現するためには，現状の所謂ビジネスインテリジェンスツールと比べて遙かに深度の高い解析が求められるようになると見られ，インタラクティブに対象空間をドリルダウンしながら発行されるデータ処理を機動的に実現することが求められると見られる．これまで並列データ処理系において索引アクセスが有効な領域は必ずしも顕著ではなかったが，当該領域は今後拡大する方向にあると言える．本論文で提案する並列データ処理システムのアウトオブオーダ型実行方式は，当該索引アクセスの飛躍的な高速化を目指すものである．

なお，並列データ処理系においては，与えられたデータ処理を細かいジョブに分割しておくことにより，各計算機で複数のジョブを並行して実行する技法が用いられることがある．各計算機において複数の入出力が同時に発行され，また，複数の演算が同時に実行さ

れるようになる点が似通っているものの，依然として，各ジョブにおいてはインオーダ型の実行方式により入出力の発行と演算の実行が逐次的に行われるに過ぎない．著者らの提案するアウトオブオーダ型実行方式は，動的なタスク分解によって，実行論理の許す限りにおいて，多数のタスクの並行実行を実現する点が大きく異なる．

3. アウトオブオーダ型並列データ処理系 Hadooode

著者らは MapReduce 型並列データ処理系のオープンソース実装である Hadoop をベースとし，提案するアウトオブオーダ型実行方式に基づく並列データ処理機能を備えた新たな並列データ処理系 Hadooode を開発している．本論文では，当該処理系におけるアウトオブオーダ型実行方式による MapReduce 型並列データ処理の実現方法，並びに Hive をはじめとする上位系を含めたソフトウェア構成法を述べる．

3.1 MapReduce 型並列データ処理のアウトオブオーダ型実行の実現

MapReduce 型の並列データ処理は，ユーザの規定する Map() と Reduce() なる二つの手続きから構成される．Map() は対象データを読み込んでキーとバリューのリストを生成する手続きであり，Reduce() はキーとバリューのリストから別のバリューのリストを生成する手続きである [4]．並列データ処理においては，前者は対象データの選択演算を行うのに用いられ，後者は集約演算を行うのに用いられることが多い．処理系である Hadoop の実装では，Mapper と Reducer なるソフトウェアモジュールを各計算機に配置し，それぞれにおいて Map() と Reduce() を実行する．この際，例えば，Mapper においては，図 1 (a) に示すように，入出力を管理する RecordReader なる機構が備える Next() なる手続きを呼び出すことにより，計算機自身の二次記憶若しくはネットワークを介して他の計算機の二次記憶からデータを読出し，読み出したデータを用いて Map() 手続きを実行し，このような手順を対象データの全体を読み出すまで繰り返す．Next() の実行と Map() の実行は，逐次的に繰り返され，すなわち，データ処理はインオーダ型の実行方式によって行われる．

これに対して，Hadooode は同様のデータ処理をアウトオブオーダ型の実行方式に基づいて行う．図 1 (b) に当該方法を示す．Mapper において，Next() 手続き

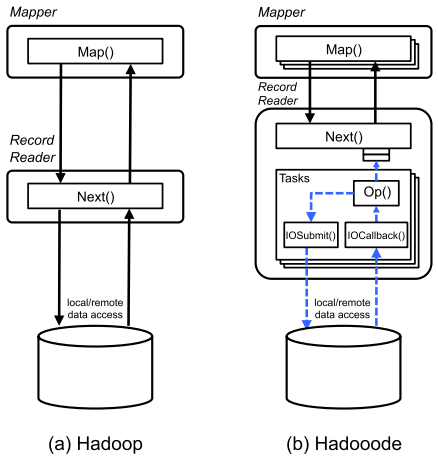


図 1　MapReduce 型データ処理におけるアウトオブオーダ型実行方式の適用
Fig. 1　Application of out-of-order execution to MapReduce data processing.

を呼び出す都度にタスクを生成し，当該タスク上において計算機自身の二次記憶若しくはネットワークを介して他の計算機の二次記憶からデータを読出し，読み出したデータを用いて Map() 手続きを実行する．このようにアウトオブオーダ型実行方式を適用することにより，ユーザの規定する Map() 手続きを改変することなく，多数の入出力とネットワーク通信を同時に発行し，また，多数の演算を同時に実行することが可能となる．この際，生成されたタスクを並行実行する手段としては，最近の OS が備えるスレッドを用いる手段と，非同期入出力を用いる手段が考えられるが，著者らは，両手段を組み合わせて用いることとした(注2)．すなわち，Mapper において，生成されたタスクを管理する機構を設け，空きスレッドが生じるとタスクを割り当ててタスクを実行し，当該タスクにおいて入出力を発行する際には非同期入出力を用いて発行して当該タスクをスレッドから分離し，非同期入出力の完了を契機として再びタスクをスレッドに割り当てて実行することとした．

なお，現実にはアウトオブオーダ型の実行方式が飛躍的な高速化を発揮するのは，索引アクセスを行う場合である．Hadoop のネイティブ実装は，対象ファイルの全体をシーケンシャルに読み出すことを基本

(注2)：後述の実装においては，pthread 並びに libaio を用いている．本論文で提案する並列データ処理のアウトオブオーダ型実行方式は当該ライブラリに依存したものではなく，多様な実装方法により実現可能なものと考えている．

としており，索引アクセスを実現していない．著者らは，Hadooode においてアウトオブオーダ型実行方式を実現する際に，加えて，索引アクセスを実現することとした(注3)．Hadoop においては，多様なジョブ実行インターフェースが用意されているが，一般には，ユーザは Hadoop に MapReduce データ処理を命令する際に，当該データ処理の対象ファイルや当該ファイルに対するアクセス方法等のメタ情報を与える．Hadooode においては，当該メタ情報を拡張し，対象ファイルをアクセスする際に利用可能な索引の構成や，対象ファイルに対するアクセス方法，並びに対象ファイルに対する選択条件等を指定することにより，MapReduce データ処理の実行時に，索引アクセスを行ってデータを読み出すことを実現する．図 2 に拡張されたメタ情報の記載例を示す．図 2 (a) は高水準のインターフェースを用いた場合の拡張メタ情報の例であり，ここでは，lineitem は Long 値である第 1 フィールドを分割キーとした分割表であり，並びに，lineitem.l_shipdate.index は lineitem 表の l_shipdate 属性に対するローカル索引であることが規定されている．すなわち，高水準インターフェースにおいては索引と実表との関係が暗黙的に定義されることとなる．一方，図 2 (b) (c) (d) は低水準のインターフェースを用いた場合の拡張メタ情報の例であり，データの読み出しに関しては図 2 (a) と同等の情報が定義されている．図 2 (a) にあるように，低水準のインターフェースにおいては，表の構成，索引の構成，索引から表へのアクセス方法を Java コードで明示的に指定する必要があるが，高水準インターフェースでは表現できない構成を柔軟に記述可能となっている．

当該メタ情報を用いたファイルアクセスは Map() 手続きには透過的であり，RecordReader 内において実現される．例えば，対象ファイルに対して B+木による索引が定義されており，当該索引を用いて指定された選択条件を満たすレコードを取得して Map() 手続きを行う場合，RecordReader 内の Next() 手続きにおいては，まず，メタ情報に従って B+木の探索を行って索引エントリを取得するための非同期入出力を発行し（図 1 (b) における IOSubmit() 手続き），その応答を契機として取得した索引エントリを解釈して選択条件を評価する演算を実行し（図 1 (b) における IOCallback() 手続きと，それによって駆動される Op() 手続き），その結果に基づいて，更にファイルからレコードを取得するための非同期入出力を発行し，その応答を契機として取得したレコードを用いて Map() 手続きを実行する．すなわち，Next() 手続きのアウトオブオーダ型実行に加えて，Next() 手続きの

```
<configuration>

   <replica name="lineitem" group=”tpch">
      <type>TextPartition</type>
      <partitionkey>0:Long</partitionkey>
   </replica>

   <replica name="lineitem.l_shipdate.index"
            group=”tpch">
      <type>LocalUnclusteredIndex</type>
      <key>10:String</key>
      <base>lineitem</base>
   </replica>

</configuration>
```

(a) High-level I/F

```
public class
LineitemPartitionRecordFormat extends
DefaultRecordFormat {

   public void initialize() {
      addPartitionKeyClass(LongWritable.class);
      addValueClass(Text.class);
   }
}
```

(b) Primitive I/F (defining table format)

```
public class
LineitemShipdateLocalIndexRecordFormat
extends DefaultRecordFormat {

   public void initialize() {
      addKeyClass(Text.class);
      addPartitionKeyClass(LongWritable.class);
      addValueClass(LongWritable.class);
      addValueClass(IntWritable.class);
   }
}
```

(c) Primitive I/F (defining index format)

```
public class
LineitemShipdateLocalIndexDereferencer
extends PhysicalDereferencer {

   public void set(List<Writable> values) {
      setOffset(values.get(0));
      setLength(values.get(1));
   }
}
```

(d) Primitive I/F (defining file access procedure)

図 2　Hadooode における拡張メタ情報
Fig. 2　Extended meta-information in Hadooode.

(注3)：後述の実装においては，B+木による索引機構を実現した．本論文で提案する Hadooode は B+木のみならず，多様な索引機構に対しても有効に機能するものと考えている．

内部的な実行手続きにおいてもアウトオブオーダ型実行を行う．

一般に，入出力を多重的に発行する場合，入出力パスのスループットはその多重度によって影響される場合がある．RecordReader においては，非同期入出力を用いて，計算機自身の二次記憶若しくはネットワークを介して他の計算機の二次記憶からデータを読出す．当然のことながら，計算機自身の二次記憶からデータを読み出す場合と，ネットワークを介して他の計算機の二次記憶からデータを読出す場合では，レイテンシやバッファキューに蓄積可能な入出力要求の数等，入出力パスの特性が異なる．例えば，二次記憶を束ねるホストバスアダプタの同時入出力発行数と比べて，計算機同士を接続するネットワークインターフェースカードの同時通信数が大幅に低い場合，アウトオブオーダ型の実行方式による入出力の発行多重度は，ネットワークインターフェースカードの同時通信数に律速されることとなる．このような場合，一つの通信パケットに対して，他の計算機の二次記憶に対する複数の入出力要求をカプセル化するブロッキング技法により，当該律速を軽減し，性能の向上に資することがある．Hadooode においては，RecordReader の発行する他の計算機に対する非同期入出力要求をいったん保留し，また，他の計算機に返すべき二次記憶からの非同期入出力応答をいったん保留し，これら複数を束ねて通信パケットに集約して他の計算機に発行する機構を備えることにより，ブロッキングを実現する(注4)．

上記では主に Mapper を例に，アウトオブオーダ型実行の実現方式を述べてきた．メタ情報を拡張することにより，ユーザの規定する Map() 手続きそのものは変更せずにアウトオブオーダ型実行を実現する点に特徴を有しており，同様のアプローチによって Reducer やその他の類似のジョブにおいてもアウトオブオーダ型実行を実現することが可能である．なお，図 2 に記載の拡張メタ情報は，索引アクセスのみならず，索引を構築するためにも用いることが可能である．Hadooode は，当該拡張メタ情報を入力として，索引の構築を行うための MapReduce 型データ処理を生成するツールを備える．当該ツールが生成する索引構築ジョブを Hadoop において実行することにより，並列処理による索引構築を実現することが可能である．

3.2 Hive 問合せ処理のアウトオブオーダ型実行の実現

Hadoop をはじめとする並列データ処理系においては，MapReduce 型のデータ処理をユーザがプログラムして実行することに加えて，近年では Hive をはじめとする高水準の問合せ応用層が充実してきている．著者らの開発する Hadooode においては，Hive の問合せ実行計画生成器を拡張することにより，高水準の問合せ処理のアウトオブオーダ型実行を実現する．Hive は HiveQL なる SQL 派生の問合せ言語によって記述された問合せを受理し，これから MapReduce 型データ処理のためのジョブを生成し，Hadoop を用いて当該ジョブを実行することにより，当該問合せの並列処理を実現する．Hadoop のネイティブ実装は，MapReduce 型のデータ処理を対象ファイルの全走査によって行うことから，Hive は，問合せ処理において，関係表の全走査を基本とし，また，結合演算としては整列併合結合アルゴリズム若しくはハッシュ結合アルゴリズムを用いる．これに対して，Hadooode においては，必ずしもファイル全体のデータを要するわけではないデータ処理に対して，索引アクセスを飛躍的に高速化する点に特徴を有しており，これを活用するべく，索引走査やネステッドループ結合等の索引アクセスを用いた問合せ実行計画によるジョブを生成可能となるように Hive の拡張を行った．この際，System-R 最適化方式 [23] に基づき，ファイルの全走査が有利か索引走査が有利かについては，問合せの選択率に従ったコストベースの最適化を行い，また，ネステッドループ結合の生成においては，現時点では left-deep 結合計画のみを探索対象とし，動的計画法を用いた結合順序の最適化を行う(注5)．

Hive 問合せ処理におけるアウトオブオーダ型実行による対象ファイルに対する索引アクセスは，MapReduce 型並列データ処理の場合と同様に，RecordReader 内で実現する．前節では，索引エントリの取得とそれから参照される対象ファイルのレコードの取得という 2 段階の例で説明したが，当該手順を複数のファイル間でのレコードの結合に拡張することにより，所謂 Map サイドにおける並列ネステッドループ結合を実現することが可能であり，当該結合をアウトオブオーダ型の実行方式により行う．

(注4)：Hadooode においては，一つの実装方法として，TCP/IP における IP のペイロードに複数の入出力要求を格納する方法を採用した．

(注5)：より高度な問合せ最適化方式 [24] については，今後の検討課題としたい．

4. Hadooode を用いたアウトオブオーダ型並列データ処理方式の評価

著者らは，Hadooode の試作実装を行い，当該実装を用いてアウトオブオーダ型並列データ処理方式の評価実験を行った．本章では，評価実験の結果を示し，提案方式の有効性を明らかにする．

著者らが構築した実験システムを図 3 に示す．20 台のデータ処理用計算機と，1 台の管理用計算機をギガビットイーサネットスイッチを介して接続し，クラスタシステムとして構成した．各計算機は 16 プロセッサコア，64GB の主記憶，2 台の OS 動作用の磁気ディスクドライブ並びに 24 台のデータ格納用の磁気ディスクドライブから構成され，オペレーティングシステムとしては CentOS Linux 5.8 が動作する．このうち，24 台のデータ格納用の磁気ディスクドライブから，SAS ホストバスアダプタが備える RAID 機構によって，セグメントサイズ 512KB の RAID-6 編成（22D+2P）の論理ユニットを構成し，当該論理ユニット上に ext4 ファイルシステムボリュームを構築した．以下の実験では，当該ファイルボリュームに対象データセットを格納して，実験を行った．

評価実験においては，データセットとして，Hadoop の性能試験に広く用いられている解析タスクデータセットと，データベース分野における標準的なベンチマークである TPC-H のデータセットを用い，それぞれのデータセットに対して解析的なクエリを実行し，その実行に要する時間を計測した．

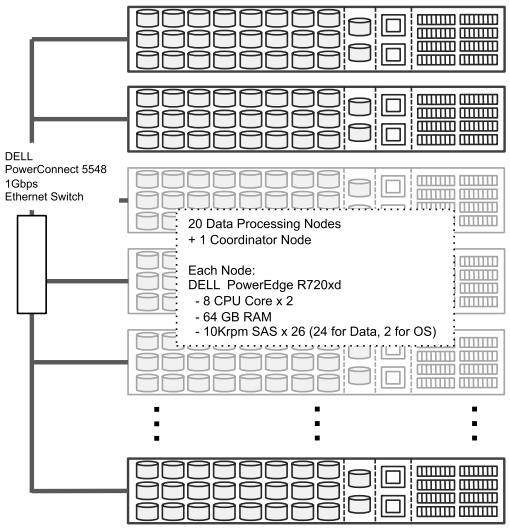


図 3　実験システム
Fig. 3　Experimental system.

この際，著者らの提案する Hadooode の実装アプローチ，すなわち，並列データ処理系のコアエンジンにアウトオブオーダ型実行方式を適用するアプローチの有効性を検証するために，ネイティブの Hadoop 実装と比較することに加えて，HadoopDB の実装アプローチ，すなわち，Hadoop のストレージエンジンに関係データベースエンジンを組込むアプローチとの比較を行う．すなわち，以下に詳細を述べる五つの Hadoop ベースの並列データ処理系のそれぞれにおいて同様の計測を行い，性能を比較する．それぞれの処理系の詳細は以下のとおりである．

Hadoop： CDH3 Update5 に含まれる標準的な Hadoop (Hadoop 0.20.2)．図 4 (a) に当該システムを模式的に示す．

Hadoop+DE： HadoopDB に相当する，ストレージにデータベースエンジンを用いる並列データ処理系．HadoopDB は，Hadoop のストレージに PostgreSQL データベースエンジンを用いることにより，索引アクセス等のデータベース技法を Hadoop において活用することを目指すものである．論文 [8] や過去の HadoopDB の公開ソースコード(注6)を参考に，著者らにおいて CDH3 Updat5 をベースとした実装を行い，これを用い計測を行った．図 4 (b) に当該システムを模式的に示す．

Hadoop+OoODE： Hadoop+DE においてデータベースエンジンに代えてアウトオブオーダ型データベースエンジン（OoODE）を用いる並列データ処理系．Hadoop+OoODE においては，各計算機が管理する二次記憶に対する入出力のみが OoODE により非同期化される．図 4 (c) に当該システムを模式的に示す．

Hadooode： 著者らが CDH3 Updat5 をベースに試作実装を行った Hadooode．Hadooode においては，並列データ処理系全体の入出力が非同期化されることとなり，例えば，他の計算機が管理する二次記憶へのネットワークを介した入出力も非同期的に行われる．図 4 (d) に当該システムを模式的に示す．

Hadooode（In-Order）： 動的タスク分解を行わない Hadooode

(注6)：https://hadoopdb.svn.sourceforge.net/svnroot/hadoopdb/

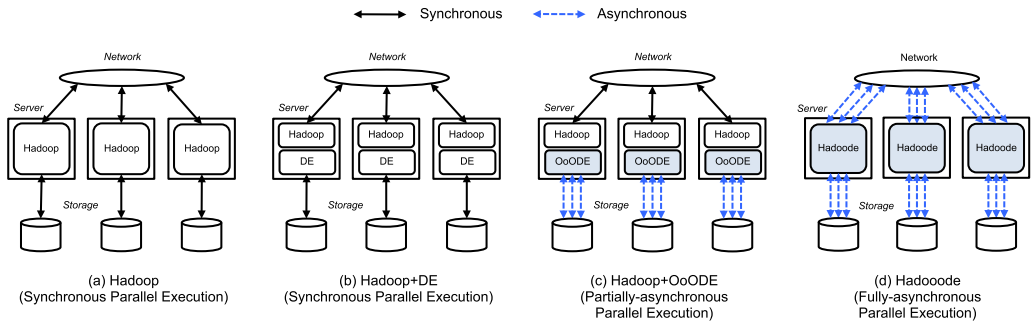


図 4　実験に用いた Hadoop ベースの並列データ処理系
Fig. 4　Hadoop-based parallel data processing system used in experiments.

尚，並列データ処理系にアウトオブオーダ型実行方式を適用するアプローチとしては，

（1） 並列データ処理系全体にアウトオブオーダ型実行方式を適用

（2） 並列データ処理系における各計算機のストレージエンジンにアウトオブオーダ型実行方式を適用

の二つに大別できると考えられる．本論文は，上記アプローチ（1）を提案し，その有効性を明らかにするものである．当該アプローチを適用した並列データ処理系（Hadooode）は，実現には既存システムの大幅な改変または拡張を要する可能性があるが，並列データ処理全体の入出力が非同期化されるという特徴を有する．一方，上記アプローチ（2）は，既存研究（HadoopDB [8]）のアイデアを応用し，Hadoop 等の並列データ処理系における各計算機のストレージとして OoODE 等のアウトオブオーダ型のデータベースエンジンを用いることにより，アウトオブオーダ型並列データ処理系を実現しようとするものである．当該アプローチを適用した Hadoop+OoODE を始めとする並列データ処理系は，既存システムを大幅に変更することなく既存システムの組合せにより実現可能であると考えられるが，各計算機が管理する二次記憶に対する入出力のみがアウトオブオーダ型データベースエンジンにより非同期化されるものである．すなわち，当該並列データ処理系は，アウトオブオーダ型実行方式が部分的に適用されたシステムであり，例えば，他の計算機が管理する二次記憶に対するネットワークを介した入出力は Hadoop により同期的に行われることとなる．並列データ処理系においては多様な問合せが想定され，アプローチ（2）によりアウトオブオーダ型実行の効果が充分に発揮される問合せも多く存在すると考えられる一方で，アプローチ（1）により初めて，若しくは，更にその効果が発揮される問合せも少なくない．本論文では，上記のアプローチの効果の違いを明らかにするべく，比較実験を行う．

表 1　Hadoop の設定
Table 1　Hadoop setting.

dfs.block.size	134217728
dfs.replication	1
mapred.tasktracker.map.tasks.maximum	16
mapred.tasktracker.reduce.tasks.maximum	16
mapred.child.java.opts	-Xmx1024m -Xms512m
io.sort.mb	256
io.sort.factor	256
mapred.reduce.parallel.copies	20
tasktracker.http.threads	80
mapred.job.reuse.jvm.num.tasks	-1 (reuse JVM)

Hadoop の主な設定は表 1 に纏める．

4.1　解析タスクデータセット

解析タスクデータセットは，Web サーバのアクセスログを模擬したデータセットであり，HTTP サーバのアクセスログを保持する UserVisits 表，HTML 文書を保持する Documents 表，Documents 表に対して付与されたメタデータを保持する Rankings 表からなる．当該データセットのスキーマ情報の詳細は文献 [25] を参照されたい．当該データセットに対するクエリは，UserVisits 表の選択を行う Selection クエリと，Rankings 表と UserVisits 表の結合を行う Join クエリからなる．図 5 にそれぞれのクエリを示す．

実験に際しては，約 1600 億件の UserVisits 表レコード（約 20TB）並びに約 20 億件の Rankings 表レコード（約 100GB）を作成して，これを用いた(注7)．いずれのケースにおいても，データセットは各計算

(注7)：Documents 表はクエリでは使用しないため，作成しなかった．

```
SELECT sourceIPAddr, destinationURL
FROM UserVisits
WHERE adRevenue BETWEEN X AND Y
```

(a) Selection Query

```
SELECT sourceIPAddr, destinationURL
FROM Rankings r JOIN UserVisits u
        ON r.pageURL = u.destinationURL
WHERE r.pageRank BETWEEN X AND Y
          u.visitDate BETWEEN '2000-01-15'
          AND '2000-01-22'
```

(b) Join Query

図 5　解析タスクデータセットに対するクエリ（X，Y は変数）
Fig. 5　Queries for analytical task data set (X and Y are variables).

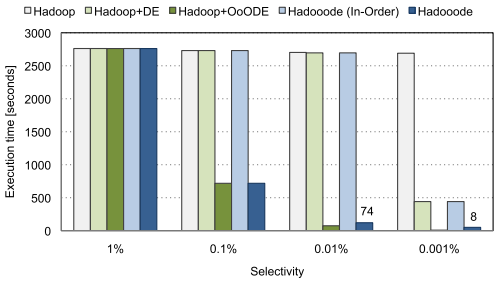


図 6　Selection クエリにおける実行時間の比較
Fig. 6　Execution time in Selection query.

機の二次記憶上に構成した ext4 ファイルシステムボリューム上に格納した．この際，Hadoop においては，当該データセットを HDFS 機構によるラウンドロビン分割によって各計算機に分配してファイルに格納した．対して，残りの四つのケースにおいては，論文 [25] のケースを参考に，当該データセットをハッシュ分割によって各計算機に分配してファイルに格納するとともに，索引を構成した．この際のハッシュ分割としては，UserVisits 表に関しては destURL をキーとする場合と，sourceIP をキーとする場合の 2 通りのハッシュ分割を比較のために設けることとし，Rankings 表は pageURL をキーとしてハッシュ分割を行った．また，索引としては UserVisits 表の destURL 並びに adRevenue，Rankings 表の pageRank に対して構成した．

4.1.1　Selection クエリ

Selection クエリは，ウェブアクセスログにおいて広告収入が指定した範囲内にあるウェブページに対するアクセスを抽出するための選択クエリである．図 5 (a) にはそのクエリを SQL を以って示すが，実験においては相当する MapReduce 型のデータ処理プログラムを構成して，実行した．尚，選択率ごとの性能特性を見るため，クエリの選択率を 1%，0.1%，0.01%，0.001%と変化させて実験を実施した

図 6 に Selection クエリの実験結果を示す．横軸はクエリの選択率を表し，縦軸は実行時間を表す．Hadoop では，表の全走査を行うため，クエリの選択率に大きく依らず長い実行時間を要していることがわかる．また，Hadoop+DE においては，選択率 1%から 0.01%の場合，アクセスパスとして全走査が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.001%においては，索引を用いた選択処理が選択され，Hadoop に対して性能向上が見られた．Hadoop+OoODE においては，選択率 1%の場合，アクセスパスとして全走査が選択されたことから，Hadoop と同程度の実行時間を要したが，Hadoop+DE と比べて，アウトオブオーダ型実行によって索引を用いた選択処理が高速化されていることがわかる．これらに対して，Hadooode（In-Order）並びに Hadooode の場合は，Hadoop+DE 並びに Hadoop+OoODE の場合とそれぞれ同様の実行時間を要した．Hadooode は，選択率 0.01%の評価実験において，Hadoop，Hadoop+DE，Hadooode（In-Order）に対してそれぞれ約 36 倍の性能向上を達成し，Hadooode の備える索引アクセス機構並びにアウトオブオーダ型実行方式のもつ高い有効性が確認された．一方で，この高速化率は Hadoop+OoODE のそれと同程度であったことから，Selection クエリに対しては，Hadoop+OoODE の実装アプローチに対する Hadooode の実装アプローチがもつ優位性は確認されなかった．

4.1.2　Join クエリ

Join クエリは，ページランクが指定した範囲内にあるウェブページに対するアクセスのうち，特定の期間内のものを抽出するクエリである．図 5 (b) の SQL で記載された Hive クエリを実行し，この際，Selection クエリと同様に，Join クエリにおける Ranking 表の選択率を 1%，0.1%，0.01%，0.001%と変化させて実行時間を計測した．この際，UserVisits 表については destURL をキーとしてハッシュ分割した場合と，sourceIP をキーとしてハッシュ分割した場合の二つのケースについて，測定を行った．

図 7 に，UserVisits 表が destURL 属性によってハッシュ分割された構成における Join クエリ（Join クエリ

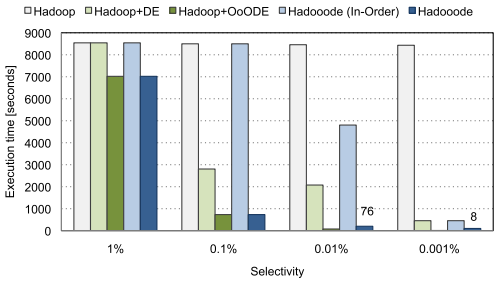


図 7　Join クエリにおける実行時間の比較（UserVisits 表が destURL 属性でハッシュ分割されている構成）
Fig. 7　Execution time in Join query (UserVisits table is partitioned by destURL).

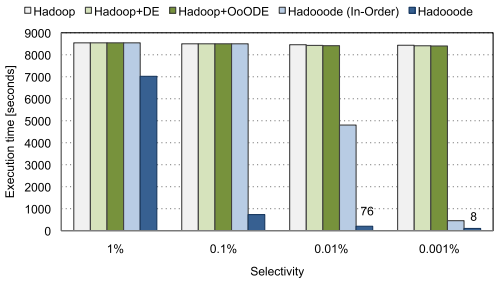


図 8　Join クエリにおける実行時間の比較（UserVisits 表が sourceIP 属性でハッシュ分割されている構成）
Fig. 8　Execution time in Join query (UserVisits table is partitioned by sourceIP).

1）の実験結果を示す．横軸はクエリにおける最外表の選択率を表し，縦軸は実行時間を表す．Hadoop においては，両表の全走査を伴う Reduce 側結合（並列ソートマージ結合）が実行され，クエリの選択率に大きく依らず長い実行時間を要していることがわかる．また，Hadoop+DE においては，選択率 1%では結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.1%，0.01%でノード内に閉じたハッシュ結合が，選択率 0.001%でノード内に閉じたネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．Hadoop+OoODE においては，全ての選択率においてノード内に閉じたネステッドループ結合が実行され，アウトオブオーダ型実行の効果により，Hadoop+DE に対して大幅な性能向上が見られた．Hadooode（In-Order）は選択率 1%，0.1%においては，結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.01%，選択率 0.001%においては，ネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．これに対して，Hadooode は全ての選択率においてネステッドループ結合が実行され，他の Hadoop 処理系と同等若しくはそれ以上の性能を達成し，Hadooode の備える索引アクセス機構並びにアウトオブオーダ型実行方式のもつ高い有効性が確認された．一方で，この高速化率は Hadoop+OoODE のそれと同程度であったことから，結合表がともに結合キーでハッシュ分割されている当該クエリにおいては，計算機間でネットワークを介した入出力は行われないため，Hadoop+OoODE の実装アプローチに対する Hadooode の実装アプローチがもつ優位性は確認されなかった．Hadooode は，選択率 0.01%の評価実験において，Hadoop に対して約 111 倍，Hadoop+DE に対して約 27 倍，Hadooode（In-Order）に対して約 63 倍の性能向上を達成した．

次に，UserVisits 表が sourceIP 属性によってハッシュ分割された構成における Join クエリ（Join クエリ 2）の実験結果を図 8 に示す．Hadoop においては，表の分割構成に変化はないため，Join クエリ 1 と同様の結果が見られた．Hadoop+DE においては，Join クエリ 1 と異なり，結合処理はノード内に閉じないため，全ての選択率で Reduce 側結合が実行され，Hadoop と同等の性能になっていることがわかる．Hadoop+OoODE は Hadoop+DE と同様に，全ての選択率で Reduce 側結合が実行され，Hadoop と同等の性能になっていることがわかる．Hadooode（In-Order）は選択率 1%，0.1%においては，結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.01%，選択率 0.001%においては，ネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．これに対して，Hadooode は全ての選択率でネステッドループ結合が選択され，アウトオブオーダ型実行の効果により Hadoop 並びにその他の Hadoop 処理系に対して大幅な性能向上が確認された．すなわち，結合表がともに結合キーでハッシュ分割されていない当該クエリにおいては，計算機間でネットワークを介した入出力が行われるため，Hadoop+OoODE の実装アプローチに対して Hadooode の実装アプローチが高い優位性をもつことが確認された．Hadooode は選択率 0.01%の評価実験において，Hadoop に対して約 111 倍，Hadoop+DE 及び Hadoop+OoODE に対して約 110 倍，Hadooode（In-Order）に対して約 63 倍の性

```
SELECT l.l_orderkey, l.l_linenumber,
       o.o_totalprice
FROM orders o JOIN lineitem l
     ON o.o_orderkey = l.l_orderkey
WHERE o.o_totalprice
      BETWEEN X AND Y
```

(a) Orders-Lineitem Join

```
SELECT l.l_orderkey, p.p_retailprice
FROM part p JOIN lineitem l
     ON p.p_partkey = l.l_partkey
WHERE p.p_retailprice
      BETWEEN X AND Y
```

(b) Part-Lineitem Join

```
SELECT l.l_orderkey, s.s_acctbal
FROM supplier s JOIN lineitem l
     ON s.s_suppkey = l.l_suppkey
WHERE s.s_acctbal
      BETWEEN X AND Y
```

(c) Supplier-Lineitem Join

図 9　TPC-H データセットにおけるシンプルな結合クエリ（X，Y は変数）
Fig. 9　Simple join queries for TPC-H data set (X and Y are variables).

能向上を達成した．

この結果から，クエリが表の分割構成に適合する場合に限り，Hadoop+OoODE はアウトオブオーダ型実行による高速化の恩恵を十分に得ることができるが，それ以外の場合では，Hadoop+OoODE が得られるアウトオブオーダ型実行の恩恵は非常に小さく，Hadoop と同等の性能になってしまうことがわかる．一方，Hadooode においてはネットワークを介した他の計算機の二次記憶への入出力を非同期化することにより，多くのクエリにおいてアウトオブオーダ型実行による高速化の効果が得られることがわかる．

4.2　TPC-H データセット

次に，データベースの業界標準ベンチマークである TPC-H [26] のデータセットを用いた性能評価を行う．計測用クエリとして，二つの表の結合を行う三つのシンプルな結合クエリ（O-L 結合，P-L 結合，S-L 結合）と，TPC-H 規定のクエリ Q3，Q8 をベースとした Q3’ 及び Q8’ からなる．シンプルな結合クエリは図 9 に，Q3’，Q8’ は図 10，図 11 にそれぞれ示された Hive クエリを実行した．これまでの実験と同様に，クエリの選択率を 1%，0.1%，0.01%，0.001%と変化させて実験を実施した．

実験に際しては，SF=20K（合計約 20TB）のデータセットを作成して，これを用いた．いずれのケースにおいても，データセットは各計算機の二次記憶上に構成した ext4 ファイルシステムボリューム上に格納した．Hadoop においては，当該データセットをラウンドロビン分割によって各計算機に分配してファイルに格納し，残りの四つのケースにおいては，ハッシュ分割によって各計算機に分配してファイルに格納するとともに，索引を構成した．この際のハッシュ分割は，各表の主キーを基に行った．索引としては，各表の主キーと外部キーに対する二次索引に加えて，Orders 表の o_orderdate と o_totalprice，Part 表の p_retailprice，Customer 表の c_acctbal，Supplier 表の s_acctbal に対して二次索引を構成した．

```
SELECT l_orderkey, l_extendedprice, l_discount,
       o_orderdate, o_shippriority
FROM customer c JOIN orders o
     ON c.c_custkey = o.o_custkey JOIN lineitem l
     ON l.l_orderkey = o.o_orderkey
WHERE c.c_acctbal BETWEEN X AND Y AND
      o_orderdate < '1995-03-15' AND
      l_shipdate > '1995-03-15'
```

図 10　TPC-H Q3’
Fig. 10　TPC-H Q3’.

```
SELECT year(o_orderdate),
       l_extendedprice * (1-l_discount) as volume, n2.n_name as nation
FROM
 (SELECT r1.l_discount, r1.l_extendedprice, r1.o_orderdate, s.s_nationkey
  FROM
  (SELECT n1.l_suppkey, n1.l_discount, n1.l_extendedprice, n1.o_orderdate
   FROM
   (SELECT c1.l_suppkey, c1.l_discount,
           c1.l_extendedprice, c1.o_orderdate, n.n_regionkey
    FROM
    (SELECT o1.l_suppkey, o1.l_discount, o1.l_extendedprice,
            o1.o_orderdate, c.c_nationkey
     FROM
     (SELECT p1.l_suppkey, p1.l_discount, p1.l_extendedprice,
             o.o_orderdate, o.o_custkey
      FROM
      (SELECT l.l_orderkey, l.l_suppkey, l.l_discount, l.l_extendedprice
       FROM part p join lineitem l ON p.p_partkey = l.l_partkey AND
            p.p_retailprice BETWEEN X AND Y
      ) p1 JOIN orders o ON p1.l_orderkey = o.o_orderkey AND
        o.o_orderdate >= '1995-01-01' AND o.o_orderdate < '1996-12-31'
     ) o1 JOIN customer c ON o1.o_custkey = c.c_custkey
    ) c1 join nation n ON c1.c_nationkey = n.n_nationkey
   ) n1 JOIN region r ON n1.n_regionkey = r.r_regionkey AND
    r.r_name = 'AMERICA'
  ) r1 JOIN supplier s ON r1.l_suppkey = s.s_suppkey
 ) s1 JOIN nation n2 ON s1.s_nationkey = n2.n_nationkey
```

図 11　TPC-H Q8’
Fig. 11　TPC-H Q8’.

4.2.1　結合クエリ

Orders 表と Lineitem 表の結合クエリ（O-L 結合）の実験結果を図 12 に示す．O-L 結合は，注文の合計金額が指定した範囲内にある注文明細を抽出するクエリである．Hadoop においては，両表の全走査を伴う Reduce 側結合（並列ソートマージ結合）が

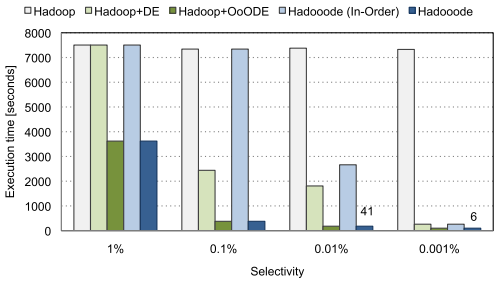


図 12　TPC-H データを用いた Orders 表-Lineitem 表結合における実行時間の比較
Fig. 12　Execution time in join of Order table and Lineitem table for TPC-H data set.

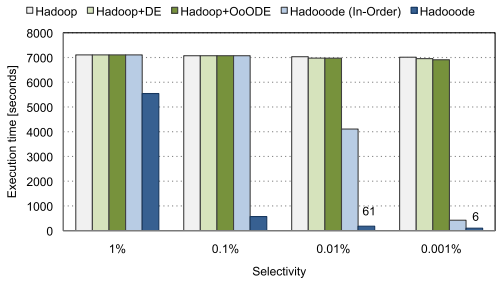


図 13　TPC-H データを用いた Part 表-Lineitem 表結合における実行時間の比較
Fig. 13　Execution time in join of Part table and Lineitem table for TPC-H data set.

実行され，クエリの選択率に大きく依らず長い実行時間を要していることがわかる．また，Hadoop+DE においては，選択率 1%では結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.1%，0.01%でノード内に閉じたハッシュ結合が，選択率 0.001%でノード内に閉じたネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．Hadoop+OoODE においては，全ての選択率においてノード内に閉じたネステッドループ結合が実行され，アウトオブオーダ型実行の効果により，Hadoop+DE に対して大幅な性能向上が見られた．Hadooode（In-Order）は選択率 1%，0.1%においては，結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.01%，選択率 0.001%においては，Hadooode（In-Order）ではネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．これに対して，Hadooode においては，全ての選択率において他の Hadoop 処理系と同等若しくはそれ以上の性能が得られた．一方で，前節における Join クエリの例と同様に，この高速化率は Hadoop+OoODE のそれと同程度であったことから，結合表がともに結合キーでパーティショニングされている当該クエリにおいては，計算機間でネットワークを介した入出力は行われないため，Hadoop+OoODE の実装アプローチに対する Hadooode の実装アプローチがもつ優位性は確認されなかった．Hadooode は，選択率 0.01%の評価実験において，Hadoop に対して約 179 倍，Hadoop+DE に対して約 44 倍，Hadooode（In-Order）に対して約 65 倍の性能向上を達成した．

次に，Part 表と Lineitem 表の結合クエリ（P-L 結合）の実験結果を図 13 に示す．P-L 結合は，希望小売価格が指定した範囲にある部品の注文明細を抽出するクエリである．Hadoop においては，表の分割構成に変化はないため，O-L 結合と同様の結果が見られた．Hadoop+DE においては，O-L 結合と異なり，結合処理はノード内に閉じないため，全ての選択率で Reduce 側結合が実行され，Hadoop と同等の性能になっていることがわかる．Hadoop+OoODE は Hadoop+DE と同様に，全ての選択率で Reduce 側結合が実行され，Hadoop と同等の性能になっていることがわかる．Hadooode（In-Order）は選択率 1%，0.1%においては，結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.01%，選択率 0.001%においては，ネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．これに対して，Hadooode は全ての選択率でネステッドループ結合が選択され，アウトオブオーダ型実行の効果により，他の Hadoop 処理系と比べて高い性能が得られた．すなわち，解析データセットでの結果と同様に，結合表がともに結合キーでハッシュ分割されていない当該クエリにおいては，計算機間でネットワークを介した入出力が行われるため，Hadoop+OoODE の実装アプローチに対して Hadooode の実装アプローチが高い優位性をもつことが確認された．選択率 0.01%の評価実験において，Hadoop に対して約 115 倍，Hadoop+DE に対して約 114 倍，Hadoop+OoODE に対して約 114 倍，Hadooode（In-Order）に対して約 67 倍の性能向上を達成した．

更に，Supplier 表と Lineitem 表の結合クエリ（S-L 結合）の実験結果を図 14 に示す．S-L 結合は，勘定残高が指定した範囲にある仕入先の注文明細を抽出するクエリである．S-L 結合においては，結合キーと

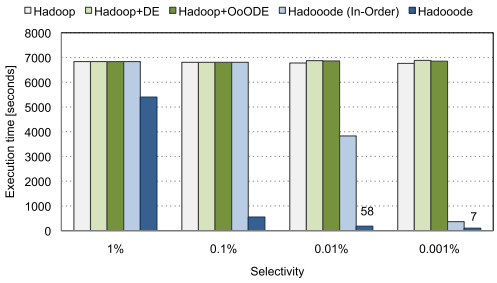


図 14　TPC-H データを用いた Supplier 表-Lineitem 表結合における実行時間の比較
Fig. 14　Execution time in join of Supplier table and Lineitem table for TPC-H data set.

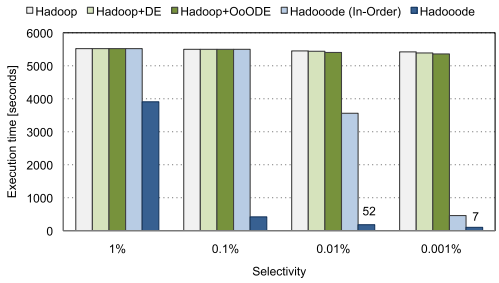


図 15　TPC-H Q3' における実行時間の比較
Fig. 15　Execution time in TPC-H Q3'.

Supplier 表の分割キーが一致しないことから，Join クエリ 2 並びに P-L 結合と同様の結果が得られていることがわかる．Hadooode は，選択率 0.01%の評価実験において，Hadoop に対して約 117 倍，Hadoop+DE に対して約 119 倍，Hadoop+OoODE に対して約 118 倍，Hadooode（In-Order）に対して約 66 倍の性能向上を達成した．

一般に，並列データベースシステム等においては，ノード間のデータ交換がなるべく生じないようにあらかじめデータを分配しておくことが多いが，意志決定支援システム等においては，多様なクエリが発行され，これを事前に見定めることは容易ではない．すなわち，結合処理において計算機の中で常に演算が閉じるように事前に計算機間でデータの分配を行うことは一般には困難であり，通常は計算機間でネットワークを介して入出力を行わざるを得ない場合がある．Hadoop+OoODE においては，各計算機が管理する二次記憶に対する入出力は非同期的に行われるものの，他の計算機が管理する二次記憶に対するネットワークを介した入出力は Hadoop により同期的に行われる．すなわち，並列データ処理においてネットワークを介する入出力が全体の入出力のうちの高い割合を占める場合においては，Hadoop+OoODE による当該並列データ処理の著しい高速化は望めない．対して，Hadooode においては，並列データ処理系全体の入出力が非同期化され，すなわち，入出力が各計算機が管理する二次記憶に対してであるか他の計算機が管理する二次記憶に対してであるかに関係なく，全ての入出力が非同期的に行われる．ゆえに，Hadooode は，当該並列データ処理においてもアウトオブオーダ型実行による著しい高速化を達成し，Hadoop+OoODE に対して高い優位性を有する．

4.2.2　TPC-H ベンチマーククエリ

次に，TPC-H ベンチマークにおけるクエリの実験結果を示す．

Q3' の実験結果を図 15 に示す．Hadoop においては，3 表の全走査を伴う Reduce 側結合（並列ソートマージ結合）が実行され，クエリの選択率に大きく依らず長い実行時間を要していることがわかる．Hadoop+DE 及び Hadoop+OoODE においては，全ての結合は Reduce 側結合により実行されたため，Hadoop と同様の結果がみられる．Hadoop+OoODE においては，Orders 表と Lineitem 表の両表は結合キーで分割されているものの，両表の選択性が低い（選択率が高い）ため，アウトオブオーダ型実行による高速化の恩恵は非常に小さくなっていることがわかる．Hadooode（In-Order）においては，選択率 1%，0.1%においては，結合方法として Reduce 側結合が選択され，Hadoop と同程度の実行時間を要したが，選択率 0.01%，選択率 0.001%においては，3 段のネステッドループ結合が実行され，Hadoop に対して性能向上が見られた．これに対して，Hadooode は全ての選択率で 3 段のネステッドループ結合が選択され，Hadooode の備える索引アクセス機構並びにアウトオブオーダ型実行方式の効果により他の Hadoop 処理系に対して大幅な性能向上が確認された．Hadooode は，選択率 0.01%の評価実験において，Hadoop 及び Hadoop+DE に対して約 105 倍，Hadoop+OoODE に対して約 104 倍，Hadooode（In-Order）に対して約 68 倍の性能向上を達成した．

次に，Q8' の実験結果を図 16 に示す．Q8' は Q3' と同様に，Hadooode においては，7 段の結合処理全てにおいてアウトオブオーダ実行により入出力が非同期化され，他の Hadoop 処理系に対して高い性能を達成していることがわかる．Hadooode は，選択率

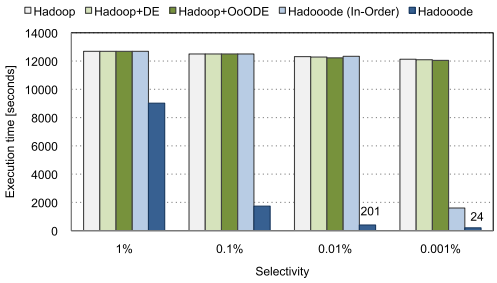


図 16　TPC-H Q8' における実行時間の比較
Fig. 16　Execution time in TPC-H Q8'.

0.01%の評価実験において，Hadoop，Hadoop+DE，Hadoop+OoODE，Hadooode（In-Order）に対してそれぞれ約 61 倍の性能向上を達成している．

この結果から，Hadooode は，多段結合のような複雑なクエリにおいても，並列データ処理のアウトオブオーダ型実行の恩恵を受けることが可能であり，他の Hadoop 処理系に対して大幅な性能向上を実現していることがわかる．

4.3　ノードスケーラビリティ

本節では，更に，ノード数を変化させた場合の性能の変化を検証する．この際，1 ノードあたりのデータサイズは一定とした．すなわち，解析タスクデータセット並びに TPC-H データセットにおいて，1 ノードあたりのデータ量を約 1TB とした．実験結果は，1 ノードでの実行時間を測定対象のノード数における実行時間で割った正規化した相対性能を示す．

図 17 に，各クエリにおける選択率 0.01%のときのノードスケーラビリティを示す．いずれのクエリにおいても，Hadooode は，他の Hadoop 処理系に対して高いスケーラビリティが得られており，20 ノードで約 97%以上のスケーラビリティを達成している．Hadoop は，Selection クエリを除くいずれのジョブにおいても，20 ノードで約 94%程度のスケーラビリティとなっていることがわかる．Hadoop+OoODE は，結合処理における入出力がノード内に閉じる場合は，図 17 (a), (b), (d) の結果にあるとおり，99%程度のスケーラビリティが得られているものの，結合処理における入出力がノードをまたがる場合は，Hadoop と同程度の 94%程度のスケーラビリティとなっていることがわかる．

このように，Hadooode における並列データ処理のアウトオブオーダ型実行は，Hadoop のスケーラビリティを阻害することなく，処理効率の向上を達成できることがわかる．

5.　関 連 研 究

5.1　Hadoop をベースとする並列データ処理系

Hadoop におけるデータ処理の効率を向上させる研究は近年盛んに行われている．データの全走査を基本とする Hadoop に対して，Hadoop++ [10]，HAIL [27]，HadoopDB [8], [9] では，索引を用いたファイル中のレコードアクセス手法を提案している．

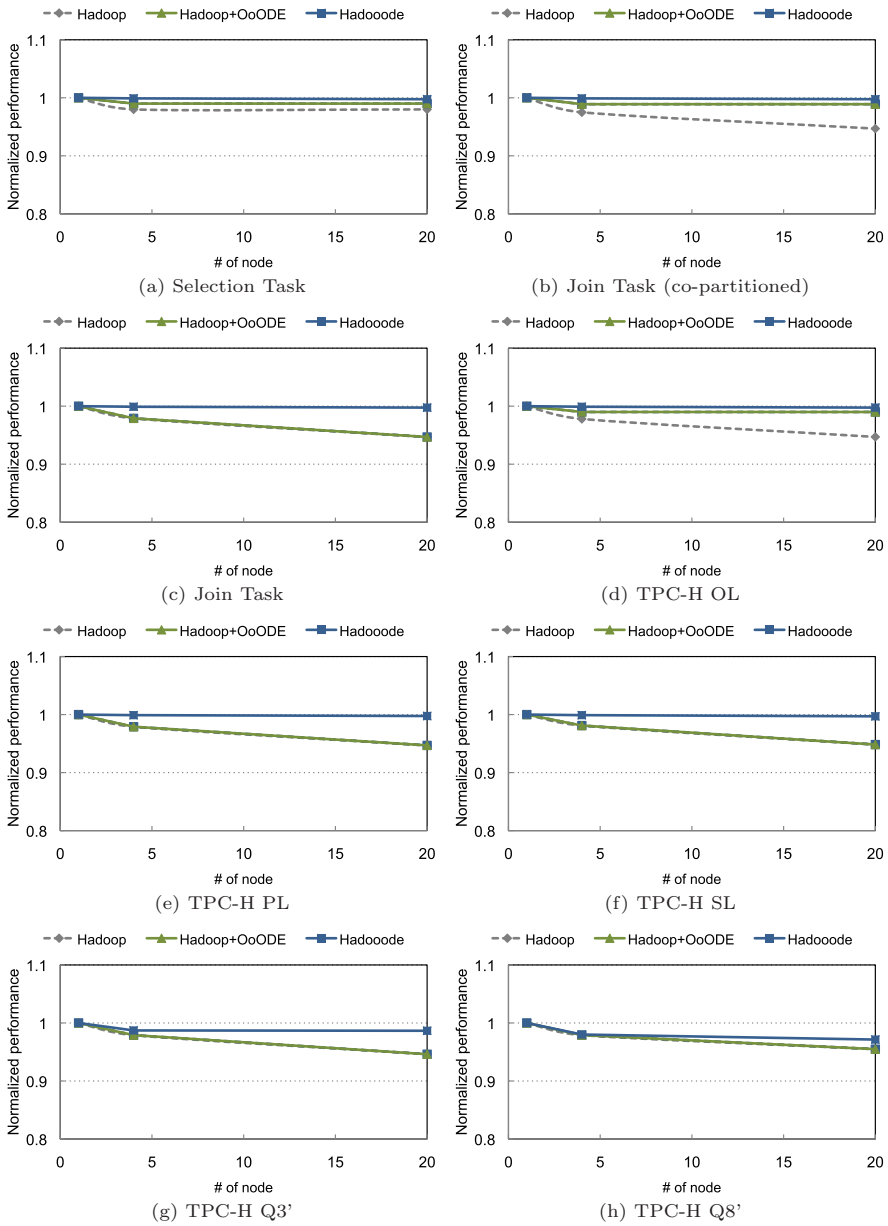
Blanas ら [28] は，MapReduce 処理モデル上における関係データベースの主要な結合処理の実装方式を議論している．Afrati ら [29] は，複数段の結合処理を一つの MapReduce ジョブで実行するための効率的なタプルの分配方法を提案している．また，HadoopDB [8], [9] や Hadoop++ [10]，Llama [30] では，複数段の結合処理を一つの Map フェーズで実行するためのデータ配置方法と構造化方法を提案している．

また，Hadoop におけるノード間のデータ共有方式の効率化に関する研究も行われてきている．Condie ら [31] は，シャッフル機構の中間データをパイプラインで転送する方法を提案している．Li ら [32] は，シャッフル機構のパイプライン化の方法として，ハッシュを用いたデータ共有機構を提案している．また，Zaharia ら [33] は大規模データの並列処理に特化した分散共有メモリ機構とそのインターフェースを提案している．

更に，Hadoop におけるデータレイアウトやデータ配置に関する研究もみられ，HDFS ブロックに対する列指向レイアウトや PAX レイアウトの適用方法 [34]〜[36] に加えて，HDFS ブロックの物理的な配置の制御方法 [37] 等が議論されてきている．

加えて，Hadoop における MapReduce ジョブの自動生成や実行環境のチューニングに関する研究も見られるようになってきた．Wu ら [38] は，Hadoop において HiveQL から MapReduce 処理を生成するコンパイラにおけるコストベースの問合せ最適化手法を提案している．Jahani ら [39] は，Map，Reduce で与えられた手続きを基に，索引等を利用したジョブ実行プランを生成する方法を提案している．また，Herodotou ら [40] は，Hadoop における種々のパラメータを最適化する方法を提案している．

これらの研究は，既存の Hadoop における並列データ処理の効率化を図るものであり，本論文とは目的を同じとするものの，本論文は Hadoop を始めとする並列データ処理に対するアウトオブオーダ型実行方式を



(a) Selection Task　(b) Join Task (co-partitioned)

(c) Join Task　(d) TPC-H OL

(e) TPC-H PL　(f) TPC-H SL

(g) TPC-H Q3'　(h) TPC-H Q8'

図 17　解析タスク，TPC-H クエリにおけるノードスケーラビリティ（選択率 0.01%）
Fig. 17　Node scalability in analytical tasks and TPC-H queries in 0.01% selectivity.

提案し，その有効性を明らかにするものであり，これらの研究とは異なる．

5.2　Hadoop に類する並列データ処理系

Hadoop の普及に伴い，Hadoop が抱える問題を克服すべく新たな並列データ処理系が提案されてきている．Dryad [41] や Hyracks [42] では，MapReduce データ処理モデルが抱える柔軟性の低さを指摘し，有向非巡回グラフ（DAG）により表現可能な柔軟性の高いデータ処理モデルやプログラミングモデルを提案している．

また，特定の用途に特化した並列データ処理系も提案されてきている．Dremel [43] は集約問合せに特化し，入れ子データに対応したカラム指向ストレージと階層的な集約処理アーキテクチャを提案している．ま

た，Pregel [44] は巨大なグラフ処理に特化し，大規模グラフ処理のための新たな計算モデルを提案している．

一方，MapReduce 処理とデータベース処理の融合を試みるシステムも近年多く見られる．多くは MapReduce エンジン並びに DFS 上に関係演算を実装するものである [45]〜[48]．一方，Osprey [49] では，並列データベースを基盤としつつ，MapReduce に類似した耐障害性機構を追加したシステムを提案している．

以上の研究は，Hadoop に類する新たな並列データ処理系を提案し，また，当該処理系におけるジョブ実行の効率化を目指すものであるが，本論文が提案するアウトオブオーダ型実行方式による並列データ処理に類するアイデアは見られない．

5.3 並列データベースシステム

並列データベースにおいても，問合せ処理における並列度の向上を目指した研究は行われてきている．Graefe [52] は，問合せ処理におけるパーティション並列性とパイプライン並列性 [50], [51] を活用し，volcano イテレータモデルに基づいた exchange operator により，オペレータ内の処理を並列に実行する方法を提案している．

また，問合せ処理における演算や入出力を実行時に部分的に並列化するという取り組みも行われている．Boncz ら [53] は，複数のタプルをパイプラインで処理し，演算をベクトル化することにより，CPU 効率の向上を図るデータベースアーキテクチャを提案している．Cieslewicz ら [54] は，コンパイラのループ展開を活用した，ハードウェアマルチスレッドによる演算の並列化方法を議論している．Roh ら [55] は，SSD のハードウェアレベルの並列性を活用する，B+木の並列探索方法を提案している．また，シーケンシャル読込み時における線形先読みは広く実装されているほか [56]，二次索引等の走査時における入出力のベクトル化技法の活用が報告されている [57], [58]．

これらの研究では，問合せ処理におけるオペレータ若しくは特定の手続きを，事前に決定した並列度で実行し，問合せ処理の効率化を狙うものである．対して，本論文は，並列データ処理（問合せ処理）全体の入出力並びに演算を実行論理が許す限りに高多重並列に実行すべく，並列データ処理の基本的な実行方式そのものを見直し，入出力の非同期化方式並びに多重度調整方式を提案するものである．

6. むすび

本論文は，並列データ処理におけるアウトオブオーダ型実行方式，並びに当該実行方式に基づくアウトオブオーダ型並列データ処理系 Hadooode を提案した．当該実行方式の下では，並列データ処理系の各々の計算機は，並列データ処理の実行時に動的にタスク分解を行い，分解されたタスクにおいて二次記憶並びにネットワークを介した他の計算機の二次記憶への入出力処理を行い，入出力の完了に伴い関連する演算を駆動する．並列データ処理系全体の入出力が非同期化される点に新たな特徴があり，特にデータセット空間の一部のデータを対象とするデータ処理において，入出力スループットが飛躍的に向上し，データ処理の大幅な高速化が実現される．20 ノードのクラスタマシン（合計 320CPU コア，480 ディスクドライブ）により Hadooode の性能評価実験を行い，実験の結果，選択率 0.01%の解析データ処理において，Hadoop に対して最大 179 倍，Hadoop において二次記憶管理にデータベースシステムを活用する処理系に対して最大約 119 倍，Hadoop において二次記憶管理にアウトオブオーダ型のデータベースシステムを活用する処理系に対して最大約 118 倍の性能向上が確認された．更に，ノードスケーラビリティの評価実験を行い，他の Hadoop 処理系と比較して高いスケーラビリティがあることが示された．

謝辞 本研究の一部は，内閣府最先端研究開発支援プログラム「超巨大データベース時代に向けた最高速データベースエンジンの開発と当該エンジンを核とする戦略的社会サービスの実証・評価」，及び日本学術振興会科学研究費補助金（特別研究員奨励費）24・7965 の助成により行われた．

文　献

[1] McKinsey Global Institute, Big data: The next frontier for innovation, competition, and productivity, 2011.
[2] O. Ratzesberger, “Agile enterprise analytics,” Keynote in USENIX FAST, 2010.
[3] Vestas Wind Systems Turns to IBM Big Data Analytics for Smarter Wind Energy, 2011.
[4] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” Proc. OSDI, pp.137–150, 2004.
[5] Hadoop, http://hadoop.apache.org/
[6] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, “Hive

— A petabyte scale data warehouse using Hadoop," Proc. ICDE, pp.996–1005, 2010.
[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," Proc. SIGMOD, pp.1099–1110, 2008.
[8] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads," Proc. VLDB, pp.922–933, 2009.
[9] K.B. Pawlikowski, D.J. Abadi, A. Silberschatz, and E. Paulson, "Efficient processing of data warehousing queries in a split execution environment," Proc. SIGMOD, pp.1165–1176, 2011.
[10] J. Dittrich, J.A. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," Proc. VLDB, pp.515–529, 2010.
[11] "Cloudera Enterprise RTQ Powered by Cloudera Impala," http://www.cloudera.com/content/cloudera/en/products/cloudera-enterprise-core/cloudera-enterprise-RTQ.html
[12] "The Stinger Initiative: Making Apache Hive 100 Times Faster," http://hortonworks.com/blog/100x-faster-hive/
[13] "Pivotal HD: The World's Most Powerful Distribution of Apache Hadoop," http://www.greenplum.com/products/pivotal-hd
[14] 喜連川優，合田和生，"アウトオブオーダ型データベースエンジン OoODE の構想と初期実験," 日本データベース学会論文誌，vol.8, no.1, pp.131–136, 2009.
[15] 合田和生，豊田正史，喜連川優，"アウトオブオーダ型データベースエンジン OoODE の試作とその実行挙動," 第5回データ工学と情報マネジメントに関するフォーラム，F3-1, 2013.
[16] 山田浩之，合田和生，喜連川優，"Hadoop におけるアウトオブオーダ型並列処理系の実装に関する一考察," 第5回データ工学と情報マネジメントに関するフォーラム，F3-3, 2013.
[17] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to data base machine and its architecture," New Generation Computing, vol.1, no.1, pp.63–74, 1983.
[18] E. Eleftheriou, R. Haas, J. Jelitto, M. Lantz, and H. Pozidis, "Trends in storage technologies," IEEE TCDE, pp.4–13, 2010.
[19] IBM DB2 10.1, "Partitioning and Clustering Guide," 2012.
[20] D.J. DeWitt, J.F. Naughton, and J. Burger, "Nested loops revisited," Proc. PDIS, pp.230–242, 1993.
[21] A. Brown and C. Kozyrakis, "Parallelizing the index-nested loops database join primitive on a shared-nothing cluster," Technical Report, no.1598, Dept. of Computer Science, Berkeley Univ., 1998.
[22] J. Liebeherr, E.R. Omiecinski, and I.F. Akyildiz, "The effect of index partitioning schemes on the performance of distributed query processing," IEEE Trans. Knowl. Data Eng., pp.510–522, 1993.
[23] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access path selection in a relational database management system," Proc. SIGMOD, pp.23–34, 1979.
[24] A. Jhingran, T. Malkemus, and S. Padmanabhan, "Query optimization in DB2 parallel edition," IEEE Data Engineering Bulletin, vol.20, pp.27–34, 1997.
[25] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," Proc. SIGMOD, pp.165–178, 2009.
[26] Transaction Processing Performance Council, TPC-H, an ad-doc, decision support benchmark, http://www.tpc.org/tpch/
[27] J. Dittrich, J.A. QuianRuiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," Proc. VLDB, pp.1591–1602, 2012.
[28] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," Proc. SIGMOD, pp.975–986, 2010.
[29] F.N. Afrati and J.D. Ullman, "Optimizing joins in a map-reduce environment," Proc. EDBT, pp.99–110, 2010.
[30] Y. Lin, D. Agrawal, C. Chen, B.C. Ooi, and S. Wu, "Llama: Leveraging columnar storage for scalable join processing in the MapReduce framework," Proc. SIGMOD, pp.961–972, 2011.
[31] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," Proc. NSDI, pp.21–35, 2010.
[32] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using MapReduce," Proc. SIGMOD, pp.985–996, 2011.
[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," Proc. NSDI, pp.15–28, 2012.
[34] A. Floratou, J.M. Patel, E.J. Shekita, and S. Tata, "Column-oriented storage techniques for MapReduce," Proc. VLDB, pp.419–429, 2011.
[35] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," Proc. ICDE, pp.1199–1208, 2011.
[36] A. Jindal, J.A. Quiane-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," Proc. SOCC, pp.1–14, 2011.
[37] M.Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla,

A. Krettek, and J. McPherson, “CoHadoop: Flexible data placement and its exploitation in Hadoop,” Proc. VLDB, pp.575–585, 2011.
[38] S. Wu, F. Li, S. Mehrotra, and B.C. Ooi, “Query optimization for massively parallel data processing,” Proc. SOCC, pp.1–13, 2011.
[39] E. Jahani, M.J. Cafarella, and C. Re, “Automatic optimization for MapReduce programs,” Proc. VLDB, pp.385–396, 2011.
[40] H. Herodotou and S. Babu, “Profiling, what-if analysis, and cost-based optimization of MapReduce programs,” Proc. VLDB, pp.1111–1122, 2011.
[41] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” Proc. EuroSys, pp.59–72, 2007.
[42] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, “Hyracks: A flexible and extensible foundation for data-intensive computing,” Proc. ICDE, pp.1151–1162, 2011.
[43] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” Proc. VLDB, pp.330–339, 2011.
[44] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” Proc. SIGMOD, pp.135–146, 2010.
[45] H. Yang, A. Dasdan, R. Hsiao, and D.S. Parker, “Map-reduce-merge: Simplified relational data processing on large clusters,” Proc. SIGMOD, pp.1029–1040, 2007.
[46] T. Kaldewey, E.J. Shekita, and S. Tata, “Clydesdale: Structured data processing on MapReduce,” Proc. EDBT, pp.15–25, 2012.
[47] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, “Tenzing A SQL implementation on the MapReduce framework,” Proc. VLDB, pp.1318–1327, 2011.
[48] J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, and D. Shakib, “SCOPE: Parallel databases meet MapReduce,” Proc. VLDB, pp.611–636, 2012.
[49] C. Yang, C. Yen, C. Tan, and S. Madden, “Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database,” Proc. ICDE, pp.657–668, 2010.
[50] D. DeWitt and J. Gray, “Parallel database systems: The future of high performance database systems,” Commun. ACM, vol.35, no.6, pp.85–98, 1992.
[51] C. Ballinger and R. Fryer, “Born to be parallel: Why parallel origins give teradata an enduring performance edge,” IEEE Data Engineering Bulletin, vol.20, pp.3–12, 1997.
[52] G. Graefe, “Volcano— An extensible and parallel query evaluation system,” IEEE Trans. Knowl. Data Eng., vol.6, pp.120–135, 1994.
[53] P. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-pipelining query execution,” Proc. CIDR, pp.225–237, 2005.
[54] J. Cieslewicz, J. Berry, B. Hendrickson, and K.A. Ross, “Realizing parallelism in database operations: Insights from a massively multithreaded architecture,” Proc. DaMoN, 2006.
[55] H. Roh, S. Park, S. Kim, M. Shin, and S. Lee, “B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives,” Proc. VLDB, pp.286–297, 2011.
[56] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, “The oracle universal server buffer,” Proc. VLDB, pp.590–594, 1997.
[57] “DSS Performance in Oracle Database 11g,” An Oracle White Paper, 2007.
[58] E. Ding, L. Dimino, G. Gopal, and T.K. Rengarajan, “Parallel processing capabilities of sybase adaptive server enterprise 11.5,” IEEE Data Engineering Bulletin, vol.20, pp.35–43, 1997.

（平成 25 年 7 月 2 日受付，10 月 27 日再受付）

山田　浩之

2003 年上智大学理工学部機械工学科卒業．同年日本アイ・ビー・エム株式会社入社，2004 年ヤフー株式会社入社．分散型全文検索エンジンの研究開発に従事．2010 年慶應義塾大学大学院理工学研究科博士前期課程修了．同年 10 月より東京大学大学院情報理工学系研究科博士後期課程，2012 年 4 月より日本学術振興会特別研究員（DC2），現在に至る．データベースシステムに興味をもつ．情報処理学会，日本データベース学会，ACM 各会員．

合田　和生

平成 12 年，東京大学工学部電気工学科卒業，平成 14 年，同大学院工学系研究科電子情報工学専攻修士課程修了，平成 17 年，同大学院情報理工学系研究科電子情報学専攻博士課程単位取得満期退学．同年，博士 (情報理工学)．日本学術振興会特別研究員，東京大学生産技術研究所産学官連携研究員等を経て，現在，東京大学生産技術研究所特任准教授．データベースシステム，ストレージシステムの研究に従事．情報処理学会，日本データベース学会，ACM，IEEE，USENIX 各会員．

喜連川　優　（正員：フェロー）

1983 年東京大学工学系研究科情報工学専攻博士課程修了，工博．東京大学生産技術研究所教授，東京大学地球観測データ統融合連携研究機構長，2013 年 4 月より国立情報学研究所所長，2013 年 6 月より情報処理学会会長を務める．データベース工学の研究に従事．内閣府最先端研究開発支援プログラムを中心研究者として推進中．電子情報通信学会業績賞，情報処理学会功績賞，ACM SIGMOD E.F Codd InnovationsAward 受賞．ACM，IEEE，電子情報通信学会並びに情報処理学会フェロー．