2008－HPC－116 (38)
2008／8／7

MapReduceにおけるRDF-DB処理に適したデータ分散格納方法の提案

谷 村 勇 輔[†] 的 野 晃 整[†] 小 島 功[†]
田 中 良 夫[†] 関 口 智 嗣[†]

ユビキタスコンピューティングの世界で用いられる “ucode” を管理するシステムに RDF-DB (RDF database) を利用するには，スケーラブルな RDF-DB を構築する技術の確立が必要である．そこで，我々は RDF-DB のバックエンドに分散ストレージと MapReduce フレームワークを用いた並列データ処理を利用することで，膨大なデータに対する多数の問合せに対応したシステムの構築を試みている．本稿では，まず MapReduce を実装する Hadoop において，データベースの結合演算を行うプログラムの性能を評価した．次に，Hadoop と RDF-DB のそれぞれの特徴に基づき，データベースの基本的なデータ格納手法である Vertical Partitioning, Horizontal Partitioning, Sorting をもとに，MapReduce フレームワークにおける RDF-DB に適したデータの分散格納方法を提案する．そして，約 274 万のトリプルに対して，2 または 3 組の predicate を選択条件とし，subject に対する結合演算を行う問合せを用いて評価実験を行った．これらを通じて，最終的に構築しようとしているシステムの設計を行う上での基本的な知見を得た．

Data Storing Methods for RDF Database Processing Using MapReduce

Yusuke Tanimura,[†] Akiyoshi Matono,[†] Isao Kojima,[†]
Yoshio Tanaka[†] and Satoshi Sekiguchi[†]

Research for scalable RDF-DB (RDF database) is highly expected today, in order to construct the “ucode” management system in the ubiquitous world. Our approach is to use parallel data processing technology with distributed storage and MapReduce framework, as a backend of RDF-DB. In this report, performance of the JOIN operation in the database domain was evaluated on the Hadoop cluster, in which MapReduce framework is provided by Hadoop. Then data storing/distributing methods based on conventional Vertical Partitioning, Horizontal Partitioning and Sorting, are proposed so that they take advantages of the Hadoop behaviors and the RDF-DB features. The proposed methods were evaluated by the experiment with the query which selects the RDF triples by 2 or 3 *predicates* and joins the triples on the *subject* from 2.4 millions' triples. Through the examinations, the design principle of our developing scalable RDF-DB system was confirmed.

1. はじめに

ユビキタスコンピューティングの世界では，世の中にある物や場所の全てに “ucode” と呼ばれる ID を割り当て，その ID によって，その物が誰によって作られた物であるかなどの様々な情報の問合せが可能になる[1]．これを実現するには ucode を管理する何らかのシステムが必要になるが，ucode のメタデータ体系は Resource Description Framework (RDF)[2] になっており，RDF を管理するための基盤として RDF-DB(RDF database) が必要とされている．

ucode 管理のための RDF-DB の課題としては，ucode の数に対するスケーラビリティである．既存の RDF-DB は単一サーバにしか対応しておらず，分散 RDF-DB，特に問合せ処理の増加に対応した並列化による高速化を考慮した実装は存在しない．しかし，数億またはそれ以上の ucode を扱う場合，データ量としては数百 TB の規模になり，現状では単一のストレージサーバで対応できない．さらに，それらのデータに対する問合せ処理を現実的な応答時間で完了することが求められる．このスケーラビリティの問題に対して，少数の高性能なストレージサーバやデータベースサーバを用いるアプローチも考えられるが，本研究ではコモディティなハードウェアを多数集めて分散ストレージを構成し，その上で分散処理を行うシステムを構築するアプローチを検討し，高いコストパフォーマンスを達成することを目指す．また，管理対象とするデータの規模に合わせた柔軟な ucode 管理システムを容易に構築するための基本アーキテクチャを明らかにする．

† 産業技術総合研究所 / National Institute of Advanced Industrial Science and Technology (AIST)

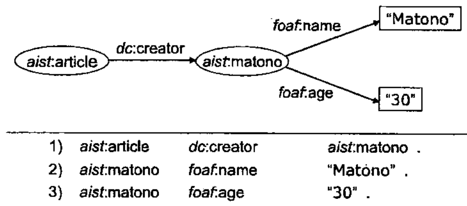

図 1　単純な RDF グラフの例

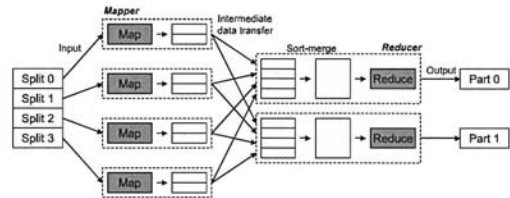

図 2　MapReduce フレームワークにおけるデータの流れ

本稿では具体的に，Google File System (GFS)[3] と MapReduce[4] のクローンである Hadoop[5] を用いて RDF-DB のバックエンドとなるシステムを構築し，GFS と MapReduce のフレームワークにおいて，RDF-DB に適したデータの分散格納方法を提案する．MapReduce は並列データ処理のためのプログラミングモデルであり，key/value のペアからなるデータにユーザ定義の map 操作を適用して中間データを生成し，続いて，同じ key に関連づけられた中間データをマージするユーザ定義の reduce 操作を適用するモデルである．MapReduce は多くのデータ処理アプリケーションに応用でき，これを用いてデータベースの問合せ処理を記述可能である．ただし，それを RDF-DB に適した形で実装する方法や得られる性能はまだ明らかになっていない．そこで，Hadoop による MapReduce の実装，および RDF-DB の特徴を考慮し，従来的なデータ格納手法を拡張した手法を提案し，予備実験を通じてそれぞれの有効性や問題点を明らかにする．これらの検討結果は，スケーラブルな RDF-DB の設計のための基本的な知見になると考える．

2. RDF とその問合せ処理

Resource Description Framework (RDF)[2] は，資源に対する構造的なメタデータを記述するための基礎的な枠組である．RDF に基づいて表現されたメタデータは，資源間の二項関係を表現できる RDF トリプル（または，単にトリプル）と呼ばれる基本単位の集合によって構成される．トリプルは subject（主語），predicate（述語），object（目的語）で構成され，subject は URI，predicate は URI，object は URI あるいはリテラルで表される．トリプルは subject が示す資源と object が示す資源，あるいはリテラルとを predicate が示す関係で成り立つことを意味する．トリプルの集合は subject と object を頂点とし，述語が有向辺に対応したラベル付き有向グラフの構造を表現する．RDF データが構成するラベル付き有向グラフを RDF グラフと呼ぶ．**図 1** は単純な RDF グラフの例である．この例では，記事（*aist*:article）の作成者（*dc*:creator）である的野（*aist*:matono）の名前（*foaf*:name）が "Matono" で，年齢（*foaf*:age）が "30" であることを示している．

RDF データの問合せは，0 個以上 3 個以下の変数を含んだトリプルの集合で検索条件を指定し，部分グラフを発見することになる．部分グラフの発見方法はいくつか実装が考えられるが，関係モデルとして扱うことが可能である．実際，RDF 問合せのほとんどの演算は関係モデルの演算に置き換え可能である[6]．本研究では MapReduce を用いて RDF-DB の問合せ処理を実装するが，それは関係モデルの演算に置き換えたものである．

3. MapReduce フレームワークを用いた並列データ処理

3.1 MapReduce

Google の MapReduce フレームワークでは**図 2** のようにデータ処理が実行される．各 Mapper は GFS からファイル断片を読み込み，ユーザが定義した map 処理を適用する．処理結果はいくつかの部分集合に分けられ，その 1 つの集合が各 Reducer に送られる．Reducer では全 Mapper から送られてきた集合をソート，マージして別のユーザ定義の reduce 処理を適用する．Reducer の出力は通常，最終結果であり，GFS に出力される．

MapReduce の利点は極めて単純なプログラミングによって並列処理が可能な点であり，大規模なデータ処理に適している．そして，データベースの JOIN 操作をそのフレームワークに沿って自然な形で実装可能である．Google の MapReduce の実装は公開されていないが，Apache Project によってオープンに開発がなされている Hadoop[5] は GFS と MapReduce のクローンを実装している．本研究では Hadoop を用いて RDF-DB のためのデータベース処理を実装する．

3.2 MapReduce/Hadoop を利用したデータベースの JOIN 操作の基本的な実装

Hadoop における MapReduce の実行の詳細と MapReduce を用いて単純に実装されたデータベースの JOIN 操作を行うプログラムの動作について説明する．以下の記述は Hadoop release 0.16.4 に基づいており，紹介する JOIN のプログラムも release 0.16.4 の contrib パッケージに含まれるものである．

初めに，入力データとして A と B のテーブルを考える．各テーブルはそれぞれ 1 つのファイルとして HDFS☆に保存されている．通常，HDFS のブロックサイズは 64MB

☆ HDFS は Hadoop Distributed File System の略であり，Google の GFS に相当する．

に設定されており，64MB を越えるファイルは HDFS 内部で複数のデータブロックに分割されて保存されている．ファイル，またはデータブロック内では行単位でレコードが格納され，各レコードは区切り文字によって，JOIN の際に key となるデータとそれ以外のデータ（value）に分けることができる．ここではこの A と B に対して $A \bowtie_{A.key=B.key} B$ を行うこととする．

1) ジョブ実行とタスク割り当て

MapReduce で書かれた JOIN のプログラムの実行は，Hadoop のシステムにおいて唯一である JobTracker にジョブを投入することで開始する．ジョブが受け付けられると，データノードで動作する TaskTracker が JobTracker にアクセスする際に map または reduce 処理を行うタスクが割り当てられる．map タスクは入力ファイル A または B の各データブロックに対して 1 つずつ割り当てられる．その割り当てアルゴリズムは，TaskTracker が動作するノード上にあるデータブロックの処理を優先的に割り当てる．各 TaskTracker が 1 度に受け付けることのできるタスク数はパラメータとして指定でき，全ての map タスクを割り当ててもリソースが余るようであれば，続いて reduce タスクが割り当てられる．

2) map タスクの実行

map タスクでは HDFS に保存されたデータブロックを行単位のテキストで読み込み，1 レコードとしてユーザ定義の map メソッドを用いて処理する．map メソッドではこのレコードに対して，入力データソース（A または B）のタグを埋め込むと同時に key を取り出して，key とレコード全体を引数に指定して MapReduce ライブラリが提供するデータ出力のメソッドを呼ぶ．MapReduce ライブラリの内部では，各レコードの key にハッシュ関数を適用して，同じ key を持つレコードが同じ reduce タスクに送られるようにレコードの送信先を決定し，レコードを送信先毎のバッファに溜める．バッファがほぼ埋まるとスレッドを起動して，バッファ上のデータを key でソートした上でローカルのファイルシステムに書き出す．map メソッドを用いて全入力レコードを処理するまでこれを繰り返し，最後にローカルのファイルシステムに書き出した複数のファイルにマージソートを適用する．こうして，各 map タスクは中間データとなるファイルを 1 つ出力する．ここではデータブロックの入力処理を a)，データ出力に関する MapReduce ライブラリ内部での一連の処理を b) と定義する．

3) reduce タスクの実行

reduce タスクではまず，各 map タスクが出力した中間ファイルのうち，自身で処理すべきデータ領域の取得を MapReduce の内部で行う．デフォルトではデータ取得用のスレッドを 5 つ起動し，map タスクの完了通知を受け取るとデータの転送を開始する．取得したデータは Hadoop が実装するメモリファイルシステムに保存し，メモリファイルシステムの容量（デフォルトでは 75MB）が十分でなければ，メモリファイルシステム上のデータをマージしてローカルのファイルシステムに書き出す処理をスレッドを用いて適宜実行する．このマージ処理は，多くの場合，中間データの転送処理とオーバーラップして行うことになる．データの転送，およびマージ処理が完全に終了した後，ユーザ定義の reduce メソッドを呼ぶ．このメソッドにて，key が同じレコードを実際に結合する処理を行い，結果を HDFS 上のファイルに出力する．reduce タスクが複数ある場合は，JOIN されたレコードを reduce タスク数と同じ数のファイルに分割して出力することになる．ここでは中間データの取得開始からマージを完了するまでの処理を c)，reduce 処理を行うためのマージしたデータの入力処理を d)，結果の出力処理を e) と定義する．

表 1　実験ノードのスペック

Hardware	Pentium 2.8GHz×2, 1 GB memory, 3-drive RAID-5 disk array (111GB for Hadoop), Intel 82544EI Gigabit Eternet.
Software	CentOS 4.5, Hadoop release 0.16.4.

表 2　JOIN を行うテーブル（A，B 共通）

Record size	128 bytes (Key: 10 bytes, Value: 116 bytes)
# records	262,144 / 524,288 / 1,048,576 / 2,097,152
Table size	32 / 64 / 128 /256 MB
Block size	8 / 16 / 32 / 64 MB

3.3 JOIN 操作の基本性能

表 1 に示すスペックのマシンを 8 台用いて Hadoop のクラスタを構築し、前項で説明した JOIN のプログラムを実行し，その性能を計測した．データとしては**表 2** を用い，レコード数の違いによりテーブルサイズが異なる計 4 種類のデータに対して測定を行った．map タスク，reduce タスクの数は 8 とし，サイズが小さい場合にはデータが 8 分割されるように Hadoop のブロックサイズを調整した．JOIN の結果として残るレコードはいずれのデータにおいても全体の 1/8 とした．Hadoop のパラメータは基本的にデフォルト値を用い，メモリファイルシステムのサイズを 75MB，ソートバッファのサイズを 100MB とした．ただし，レプリカ数は 8 として全処理ノードにレプリカが作られるようにした．

JOIN のプログラム全体の実行時間，および MapReduce における各タスクの入出力やデータ転送に要した時間を**表 3** に示す．表中の a)〜e) は前節の定義に対応しており，b) や c) は MapReduce 内部で行われるソートやマージ処理等を含んでいる．結果より，レコード数の増加にほぼ比例して全体の実行時間が増加しているのが分かる．I/O のオーバーヘッドが実行時間に占める割合はテーブルサイズが 32MB の時は 28%であるのに対し，64MB の時は 55%，128MB と 256MB の時は 68%と増加している．特に b) や c) ではデータサイズの増加に対して急激に時間がかかるようになった箇所があり，その原因は扱うデータサイズの増加により，メモリ内で処理

表 3 JOIN 操作の基本性能

Table size [MB]	32	64	128	256
Total execution time [sec]	20.6	26.4	40.8	55.4
a) Input in the map task [sec]	0.69	1.22	2.29	6.40
b) Output in the map task	1.89	3.40	11.3	13.8
c) Transfer in the reduce task	2.55	8.59	11.5	11.9
d) Input in the reduce task	0.41	0.78	1.54	3.01
e) Output in the reduce task	0.26	0.44	1.12	2.45

できないデータをローカルのディスクに書き出す作業が加わったためであると考える．一方，レコード数に対する処理効率が最も良いのもテーブルサイズが 64MB の時であり，75,709[records/sec] であった．

次に，HDFS におけるレプリカ数とデータブロックの所在を考慮したタスク割り当ての関連性を調べた．その結果，レプリカ数が 1 の時は 54%の確率で入力となるデータブロックが保存されたノードに map タスクが割り当てられた．同様に，レプリカ数が 3 の時は 71%，4 の時は 92%であった．一方，テーブルサイズが 256MB の時に，アクセスするデータブロックがローカルにある場合とそうでない場合とで読み込み速度を比較したところ，本実験環境では違いが見られなかった．ただし，データブロックがリモートにある場合に 7.4%の確率で，読み込みに通常の 4 倍以上の時間を要することがあった．

3.4 考 察

JOIN 操作の基本性能の評価により，Hadoop が提供する MapReduce では，1 回の MapReduce 処理のオーバーヘッドが比較的大きく，最も効率が良い場合でも約 210 万レコードのテーブルを JOIN するのに 55 秒程度かかることが分かった．性能改善のためにはメモリ内処理を増やすようなパラメータの調整や，無駄な処理を省いたり複合的な処理を行うなどの改修を JOIN 操作の実装や MapReduce ライブラリの実装に施すことが考えられるが，これらは今後の課題である．

一方，データベース処理の実装において工夫できる点もある．第 1 に，複数の演算をできるだけ 1 回の MapReduce で行えるようにするのが望ましい．例えば，選択演算や射像演算は map や reduce タスクの中で行える．特に，map 処理の最初に選択操作を行うことで，MapReduce の内部で処理されるデータ量を減少させ，ソートやマージの処理時間の削減，転送される中間データ量の削減を図れるだろう．I/O 処理にかかる時間もデータ量に比例するので，この効果は非常に高いと考える．第 2 に，MapReduce では同じ key に対する 3 つ以上のテーブルの JOIN を 1 回の MapReduce で行える．これも 1 回のデータベースの問合せに対する MapReduce の実行回数の削減に活用できると考える．

4. 提 案 手 法

4.1 RDF-DB の特徴

MapReduce を用いたデータベース処理の実装に関して，3.4 節で述べた知見が得られたため，それをもとに，さらに RDF-DB の特徴を考慮して RDF-DB 処理に適したデータ分散格納方法を提案する．RDF-DB の実装では，データを格納するツールとして関係データベースのようなものを考える場合，テーブルのカラムが RDF のトリプル（subject (S)，predicate (P)，object (O)）に限定される．この構造の単純性を利用して処理の高速化を図ることが可能であり，MapReduce における実装では以下の特徴を利用できると考える．

- subject や object に比べて predicate の数は極端に少ない．数百万のトリプルを考える場合でも predicate は 1,000 もあれば事足りる．
- 問合せにおいて predicate は変数になることがほとんどなく，多くの場合は検索条件として指定される．
- 問合せにおいて subject は検索条件として指定されることがほとんどなく，多くの場合は変数になる．
- "ucode" を管理するシステムへの応用を考える場合は，RDF-DB のレコードの更新はそれほど頻繁でない．すなわち，レコードの更新にある程度の時間がかかっても問題ないと仮定してよい．

これらを踏まえて，次項より 3 つのデータ分散格納方法を提案する．

4.2 Vertical Partitioning

本手法は D. J. Abadi らによる Vertical Partitioning の RDF-DB への応用[7] に基づいており，**図 3** のように predicate 毎にテーブルを用意する．predicate の 1 テーブルに対して 1 ファイルを用いることとし，predicate の値をファイル名に付けて，ファイルに subject と object の組み合わせを行単位で格納する．検索条件に predicate が指定されれば，その predicate に対応したテーブル（ファイル）を MapReduce で処理すればよい．predicate の数は高々1,000 なので，対応する predicate のテーブルを見つけるのは容易である．例えば，$<?X, P_1, ?Y_1> \bowtie <?X, P_2, ?Y_2>$ を行う場合，P_1 と P_2 のファイル，またはデータブロックに対して map 処理を適用する．そして，subject を key にハッシュを計算して同じ key のレコードを同じ reduce タスクに集め，reduce 処理において JOIN を実行する．

本手法の欠点は，レコードサイズが小さい時に predicate のテーブルの大きさにばらつきが生じやすいことであるが，predicate の偏りが少なかったり，レコードサイズが十分に大きければあまり問題とならない．新しくレコードを追加する作業は，対応する predicate のファイルの最後尾にデータを追加するだけでよい．また，あらかじめ object をソートしておけば検索条件によっては，さらに高速化を図ることができる．

4.3 Horizontal Partitioning

本手法では**図 4** のように subject，predicate，object 毎にテーブルを用意する．図において T はトリプル (triple) の ID を表す．そして，subject テーブルと object テーブルはあらかじめトリプルの順にソートして

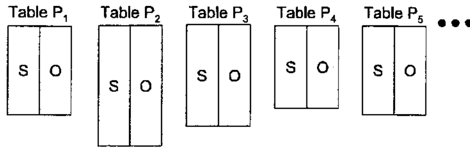

図 3 Vertical Partitioning のデータ格納方法

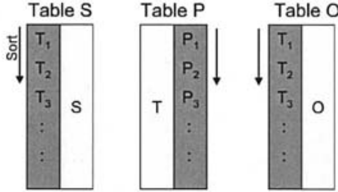

図 4 Horizontal Partitioning のデータ格納方法

おき，predicate テーブルは predicate の順にソートしておく．$<?X, P_1, ?Y_1> \bowtie <?X, P_2, ?Y_2>$ の問合せに対しては，指定された predicate に対応するトリプルの ID を抜き出す作業を MapReduce 処理の前処理として行う．predicate の数は少なく，かつ predicate テーブルは predicate を key にソートされているので，多くの場合，全てのレコードをスキャンする必要がなく，前処理は瞬時に完了するはずである．その後，抜き出したトリプル ID を MapReduce のタスクを担当する全ノードに配布する．MapReduce 処理では，subject テーブルを処理する map タスクと object テーブルを処理する map タスクを実行し，まずトリプル ID を key に JOIN を適用する．次に，2 つ目の MapReduce を適用し，subject を key に JOIN を行うことで問合せの結果が得られる．

本手法の最大の欠点は MapReduce が 2 段の処理になることである．しかし，RDF-DB の問合せによっては，射像演算により最終的に object が削除される場合があり，その場合は subject とトリプル ID の JOIN だけを 1 段の MapReduce で行えばよいことになる．あるいは，JOIN の結果として残るレコード数が十分に少なければ，MapReduce の後処理として object を付与してもそれほど時間がかからない場合も考えられ，それらの場合には本手法を活用できる．新しくレコードを追加する際にはトリプルに ID を振り，ID と S，P，または O の組み合わせを 3 つのテーブルの適切な位置に追加することになる．

4.4 3 Orderd-Sorting

本手法はデータをソートして格納するというごく一般的な手法の応用であり，**図 5** のようにトリプルのテーブルを 3 つ用意する．そして，それぞれ subject 順，predicate 順，または object 順にあらかじめソートしておく．データ量が多い場合は，1 つのテーブルを複数のファイルに分割し，分割されたファイル毎にソートをすることとする．例えば，$<?X, P_1, ?Y_1> \bowtie <?X, P_2, ?Y_2>$ の問合せに対しては predicate 順にソートされたテーブル

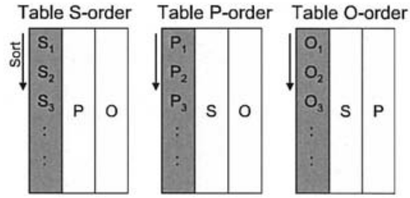

図 5 3 Ordered-Sorting のデータ格納方法

を処理する．ソート済みであることにより，map タスクでは全レコードの predicate の値にアクセスする必要はなく，P_1，P_2 の部分にのみアクセスして処理を完了できる．レコード数が多くなったとしても，ファイル内の必要な predicate の発見はバイナリサーチなどを用いればよい．その後，subject を key としてハッシュを計算し，reduce タスクで JOIN を行うことで問合せの結果が得られる．

本手法の欠点はデータの格納に 3 倍のスペースを要求することであるが，工夫の余地が残っている．subject が問合せの条件になることは稀であることを利用し，subject 順のテーブルは用意しない，さらに predicate を第 1 key，object を第 2 key としてソートすることとし，その 1 テーブルしか用意しない等の方法が考えられる．一方，HDFS ではデフォルトでデータブロックのレプリカを 3 つ持つようになっている．この仕組みを改造して，レコードの格納順が異なるレプリカを管理できるようにするというアプローチも考えられる．新しくレコードを追加する際には，3 つのテーブルにおいて，レコードをソート済みのデータブロックの適切な位置に挿入することになる．

5. 予 備 評 価

前節で述べた提案手法を評価するため，predicate を検索条件に subject で JOIN を行う a)$<?X, P_1, ?Y_1> \bowtie <?X, P_2, ?Y_2>$ の問合せと b)$<?X, P_1, ?Y_1> \bowtie <?X, P_2, ?Y_2> \bowtie <?X, P_3, ?Y_3>$ の問合せの実行時間を測定した．ただし，Horizontal Partitioning は MapReduce を 2 回行う必要があり，今回用いる問合せにおいては，明らかに性能上不利であるため，この実験から除外した．ここでは他の 2 つの手法に対して行った実験結果を報告する．実験環境としては 3.3 節で用いた 8 ノードからなるクラスタを用い，**表 4** に示す RDF データをあらかじめ格納しておいた．RDF データの中身は乱数を用いて生成したものである．a) の問合せの結果として得られるレコード数は 173，データサイズにして 55KB である．b) の問合せの結果はレコード数が 15，データサイズは 6.7KB である．

Vertical Partitioning では表 4 のデータを predicate 毎のファイルに分割し，かつファイルには subject と object のみを格納するため，1 ファイルあたりの平均サイズ

表 4 評価に用いる RDF データ

Record size	196 bytes (S:64 bytes, P:64 bytes, O:64 bytes, delimiters:4 bytes)
# records	2,739,138
Total record size	510 MB
# resource patterns	50,000
# property patterns	1,000

表 5 提案手法の性能

	Vertical Partitioning	3 Ordered-Sorting
Query a)	9.0 [sec]	9.6 [sec]
Query b)	9.4 [sec]	9.8 [sec]

は 348KB であった．a) の問合せでは P_1 と P_2 の 2 ファイルを MapReduce に与えるため，MapReduce の map 数は 2 とした．同様に b) の問合せでは map 数は 3 とした．扱うデータサイズが比較的小さくなるため，reduce 数は常に 1 とした．本手法の測定においては，predicate のテーブルを見つけた後の MapReduce の結果を測定しているが，前処理を含んだとしても増加分は 0.01[sec] 程度である．

3 Ordered-Sorting では表 4 を 64MB 単位で分割し，そのデータブロック毎にソートを行った上で HDFS に格納した．HDFS のブロックサイズは 64MB であり，これらのファイルを MapReduce の入力として与える場合，検索条件に指定する predicate の数に依存せず，map 数は常に 8 となる．reduce の数は Vertical Partitioning の実験と揃えて，常に 1 とした．また，本実験ではファイル内の predicate を発見するのにバイナリサーチが利用できると仮定して，JOIN に必要なレコードを先頭から *logN* 行以内にできるだけ収まるように工夫した．結果として，本手法では得られるべき 173 レコードのうち，171 レコードを結果として得た．この差は両手法を比較するにあたり，無視できる程度のデータサイズの違いだと考えている．

実験の結果を**表 5** に示す．結果より，Vertical Partitioning が 3 Ordered-Sorting より若干短い時間で問合せ処理が完了できるといえるが，その差は小さい．b) の問合せは a) の問合せよりアクセスするデータ量が 1.5 倍に増加しているが，問合せの実行時間はほとんど変わらない．これは，この規模のデータ処理においては MapReduce のオーバーヘッドが大きく，その差が現れにくかったためだと考える．一方，理論的には Vertical Partitioning の場合は map 数が増えても並列に処理されるため，実行時間への影響は少ないはずである．

6. まとめと今後の課題

本研究ではスケーラブルな RDF-DB を構築するために，バックエンドに HDFS を用いた分散ストレージと MapReduce フレームワークを利用した並列データ処理技術を利用する方法を検討した．Hadoop が実装する MapReuduce は，約 210 万レコードのテーブルの JOIN において約 55 秒かかるという時間オーダであった．レコードの増加に対しては map 数を増やすことで対処できるにしても，このオーダは関係代数の演算を繰り返し行う RDF データの問合せには不利であり，map 操作あるいは reduce 操作の中で複数の演算を行う方法を検討する必要があることが分かった．この知見と RDF-DB の特徴を活用して，データベース処理における既知のデータ格納手法を拡張した 3 つの手法を提案した．そのうち，Vertical Partitioning と 3 Ordered-Sorting について予備評価を行い，約 274 万レコードに対する問合せを 10 秒程度で処理できることを確認した．これは 3.3 節で行った単純な結合演算の 5 倍以上の性能改善である．

今後は得られた知見を最終的に構築するシステムの設計に活用していくとともに，実際的な RDF データを用いた評価実験や大規模な環境での評価実験に取り組む予定である．

謝辞 本研究の遂行にあたり，貴重なご意見を頂いた川辺治之（日本ユニシス株式会社）氏に深く感謝致します．

参 考 文 献

1) T-Engine Forum. ユビキタス ID アーキテクチャ. http://www.t-engine.org/japanese/archives/UID-C00002-0.00.24.pdf. 910-S002-0.00.24/UID-CO00002-0.00.24.
2) World Wide Web Consortium (W3C) Recommendation. Resource Description Framework (RDF). http://www.w3.org/RDF/, 2004.
3) Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating System Principles*, 2003.
4) Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI'04)*, 2004.
5) Hadoop. http://hadoop.apache.org/.
6) Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, Digital Media Systems Laboratory, HP Laboratories Bristol, 2005.
7) DanielJ. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pp. 411–422, 2007.
8) Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *2007 ACM SIGMOD International Confnerence on Management of Data*, pp. 1029–1040, 2007.