

データ移動方式による PostgreSQL への結合演算実装の試み

瀧沢 亮太^{†1,a)} 川島 英之^{†2} 建部 修見^{†2}

概要：膨大量のデータの管理を行う情報基盤としてリレーショナルデータベース管理システム（RDBMS）が研究開発されてきた。RDBMS はデータをリレーションとして表現し、リレーションに対する様々な演算子をユーザに提供する。また、膨大量のデータに対応するために日進月歩で新しいシステムが開発されてきたが、開発されて間もないためシステム安定性という観点で一抔の不安が残る。そこで本研究では著名なオープンソースの RDBMS である PostgreSQL を対象として、等結合演算を高速化することを考える。PostgreSQL における等結合演算の実装は高速化のために単純非並列ハッシュ結合が使われている。そこで本研究では単純並列ハッシュ結合を実装し、データ移動方式を用いて PostgreSQL への結合演算を実装することを提案する。この実装を用いた提案手法と PostgreSQL との性能評価を行った。Node 数 2 での並列ハッシュ結合を用いた提案手法は、PostgreSQL の等結合と比較して、最大約 1.42 倍の高速化を達成した。

1. 序論

人類が扱うデータ量は拡大の一途を辿っている。例えば天体望遠鏡 LSST では毎晩 20GB、粒子加速器 LHC では年間 30PB、通信情報の交換を行うルータでは 322Tbps と莫大なデータが生成されている。このような膨大量のデータを管理するためにリレーショナルデータベース管理システム（RDBMS）が研究開発されてきた。RDBMS はデータをリレーションとして表現し、リレーションに対する様々な演算子をユーザに提供する。ユーザは SQL と呼ばれる言語でデータ処理要求を記述する。これを問合せと呼ぶ。その問合せは上記の演算子に変換され、最適な実行計画が探索された後、それが実行される。それらの演算子は選択、射影、結合、集約など、一見して単純なものばかりである。そこに機械学習・データマイニングの分野に存在するような複雑なものはない。

しかしながらこの単純な演算子に対する需要は大きいと考えられる。なぜなら膨大量のデータに対してこれらの単純なリレーショナル演算子を適用するために日進月歩で新しいシステムが開発されてきているからである。その例として、Facebook 社が開発した Presto [1]、Cloudera 社が開発した Impala [2]、DataBricks 社が開発した SparkSQL [3] などが挙げられる。これらのシステムはいずれも分散

ファイルシステム上で分散問合せ処理を実行可能であり、高い性能を有する。

一方、それらのシステムは開発されて間もないため、システム安定性という観点で一抔の不安が残る。システム安定性を保持しながら高速な演算子を得るには、長く使われてきたシステムの演算子を改良する方式が好ましい。

そこで本研究では著名なオープンソースの RDBMS である PostgreSQL を対象として、等結合演算を高速化することを考える。PostgreSQL における等結合演算の実装は高速化のために単純非並列ハッシュ結合が使われている。しかし近年一般化しているマルチコア環境ならびにクラスタ環境の特性を単純非並列ハッシュ結合は享受することができない。そこで本研究では単純並列ハッシュ結合を実装し、データ移動方式を用いて PostgreSQL への結合演算実装を試みる。単純並列ハッシュ結合は、複数のマシンを利用することで等結合演算を細分化し、単純非並列ハッシュ結合と比べた時に処理時間が向上していることを示す。我々の実装した単純並列ハッシュ結合はマルチコア・クラスタ環境の恩恵を十分に享受可能であり、PostgreSQL に組み込まれている結合よりも性能を高められることを示す。

本稿の構成は以下の通りである。2 節では、本研究の背景について述べる。3 節では、本論文の提案手法である並列ハッシュ結合について述べる。4 節では、並列ハッシュ結合の具体的な実装について解説する。5 節では、並列ハッシュ結合と PostgreSQL での評価を示す。6 節では、関連研究を紹介する。7 節では、まとめとして結論と今後の課

^{†1} 現在、筑波大学大学院 システム情報工学研究科

^{†2} 現在、筑波大学 計算科学研究センター

^{a)} takizawa@hpcs.cs.tsukuba.ac.jp

題を述べる。

2. 背景

この節では本研究を行うに至った背景について述べる。ここで、等結合と非等結合について述べる。等結合とは、結合条件によって特定の列の値が等しいものを結びつける。非等結合とは、結合条件によって特定の列の値が特定の列の範囲内であるものを結びつける [5]。RDBMS を運用するにあたって、複数のリレーションを同時に参照し、一括したデータとして扱う必要性は非常に高い。これを実現するために、RDBMS では、特定のキーで表を結合する等結合を用いる [6]。2.1 節では RDBMS である PostgreSQL における Hash Join について述べる。2.2 節では PostgreSQL における等結合の評価について述べる。2.3 節では本研究で扱う研究課題について述べる。

2.1 Hash Join

Algorithm 1 EXPLAIN: Hash Join

```

1: =# explain select * from regt-users as u join address as a
  on u.address-id = a.address-id;
2: QUERY PLAN
3:
4: Hash Join (cost=6410.27..8298.27 rows=10000 width=138)
5: Hash Cond: (u.address-id = a.address-id)
6: -> Seq Scan on regt-users u (cost=0.00..252.00
  rows=10000 width=88)
7: -> Hash (cost=3704.23..3704.23 rows=121523 width=50)
8: -> Seq Scan on addresses a (cost=0.00..3704.23
  rows=121523 width=50)

```

Hash Join は、最初に内側のテーブルのハッシュ表を作成する。ハッシュ表はメモリに作成される。外側のテーブルはハッシュ表と付き合わせて結合していく。ハッシュ表を一度作成することができれば高速で動作を行う。ハッシュ表はメモリに作成されるので高速であるが、ハッシュ表のサイズがメモリサイズより大きくなるとかなり遅くなってしまふ [4]。Merge Join と同様に、少なくとも片方のテーブルが結合対象のテーブルに対して読み込みが終わらなければ結果を返すことができないので、Nested Loop Join と比較すると、最初の一件の結果を戻す速度は劣る。また、結合対象行が多く十分なメモリが確保できる場合や、結合条件の索引がなくテーブルのフルスキャンが必要な場合 Nested Loop Join より早くなることが望める。また、Algorithm 1 は EXPLAIN コマンドの結果である [4]。最初に内側テーブルのレコード全件は、結合条件を用いたハッシュ関数によってハッシュ値を取得する。ハッシュ値を元にメモリ上にハッシュ表が作成される。ハッシュ表を作成した後に外側テーブルを捜査する。外側テーブルは、ハッシュ表を作成するために使用したハッシュ関数を

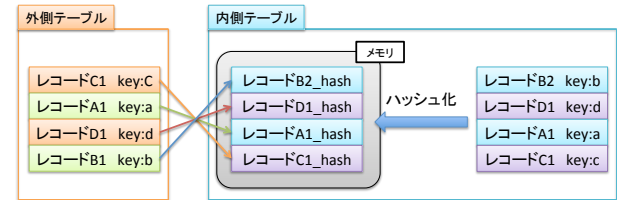


図 1 Hash Join

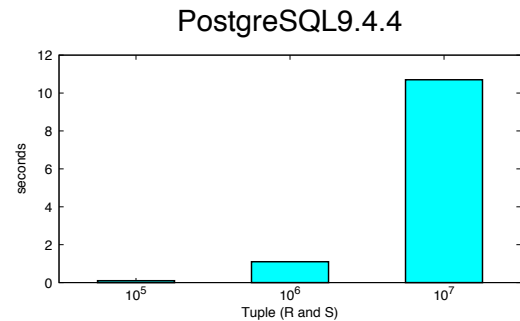


図 2 PostgreSQL9.4.4 における等結合の評価

用いながらハッシュ値を取得し、ハッシュ値に従ってハッシュ表と結合条件を満たすか調べる (図 1)。

2.2 PostgreSQL における等結合の評価

表 1 リレーション

タプル構成	int 型 <i>key, val</i>
R, S	10 ⁵ , 10 ⁶ , 10 ⁷ TUPLE

この節では、PostgreSQL における等結合の評価について述べる。評価には PostgreSQL 9.4.4 を使用した。表 1 に示す通り、2つのリレーション R, S のタプル構成は、int 型の key, val の 2 属性である。この時以下のクエリを行い、実行時間を計測した。

```

SELECT *
FROM R, S
WHERE R.key = S.key

```

表 1 に示す通りタプル数を変動させ、クエリを実行した結果が図 2 である。

図 2 の縦軸はクエリ実行時間であり、横軸は、リレーション R, S のタプル数である。リレーション R, S のタプル数が、10⁵ である場合の実行時間は 0.1 秒であったのに対し、リレーション R, S のタプル数が、10⁷ である場合の実行時間は 10.7 秒であった。この結果より、リレーションのサイズが大きいものであると実行時間が増加することがわかる。

2.3 研究課題

RDBMS である PostgreSQL における等結合は 2.1 節までで述べた。等結合は RDBMS では頻繁に使用され、必

要不可欠なものである。しかし、2.2 節で述べたように、PostgreSQL において等結合を行う際に、リレーションサイズが大きくなるにつれて、コストも大きくなってしまふ。そこで、本研究では、データ移動方式による PostgreSQL への並列ハッシュ結合演算を実装することを試みる。

3. 並列ハッシュ結合

この節では、データ移動方式による PostgreSQL への並列ハッシュ結合演算の実装を提案する。PostgreSQL はパーティショニング専用の組み込み機能ではなく、「継承」「CHECK 制約」「トリガ」を組み合わせて実現している [7]。そのため複数のリレーションを動的に結合する結合演算は処理負荷が重く、高コストとなってしまう。

そこで、分散結合アルゴリズムの一つである並列ハッシュ結合を取り上げる。2.1 節では、1 つのリレーションに対して 1 つのハッシュ表を用いて結合処理を行うことを述べた。並列ハッシュ結合では、最初にリレーションを細分化し、その後複数台のマシンに割り振り、各々のマシン毎にハッシュ表を用いて結合処理を行うことで、処理時間を向上させることを目的とする。2 つのリレーションで並列ハッシュ結合を行うとすると、次のような段階で述べることができる [8]。

- (1) メインマシンで動く master に 2 つのリレーションが読み込まれる
- (2) 2 つのリレーション内のタプルは同一のハッシュ関数によりハッシングされる
- (3) ハッシングされたタプルはハッシュ値に従い、master からハッシュ値に対応した worker が動くマシンへ転送される
- (4) worker は各リレーションのタプル群が結合条件を満たしているかハッシュ結合を用いて調べる
- (5) 各 worker の結合結果を master へ転送する

次節からは、並列ハッシュ結合について詳しく述べていく。3.1 節では、master が読み込んだリレーションを分割するハッシュ分割について述べる。3.2 節からは並列ハッシュ結合に用いられているアルゴリズムについて述べる。3.2.1 節では、リレーションを分割するために用いられるデータ分散のアルゴリズムを述べる。3.2.2 節では、master で分割されたデータを受け取った worker 側で用いる照合処理のアルゴリズムについて述べる。

図 3 は 2 つのリレーションでの並列ハッシュ結合の全体図を図示したものである。

3.1 ハッシュ分割

この節では、ハッシュ分割について述べていく。ハッシュ分割とは、あるリレーションをハッシュ値に基づいて各パーティションに均等に分割することである。例えば、 N 分割する場合には、 $0 \sim N - 1$ の整数を返すハッシュ関

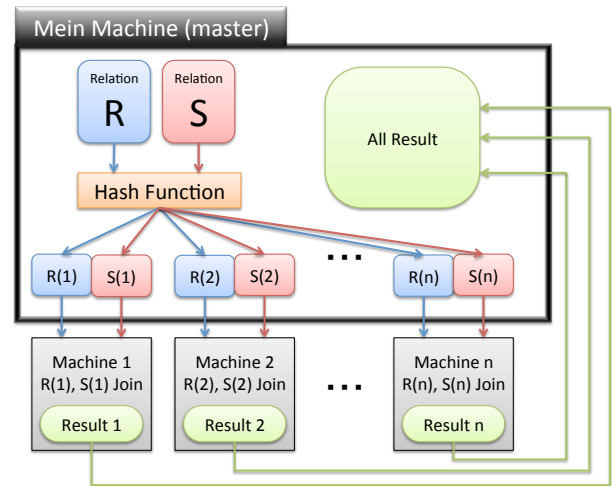


図 3 並列ハッシュ結合

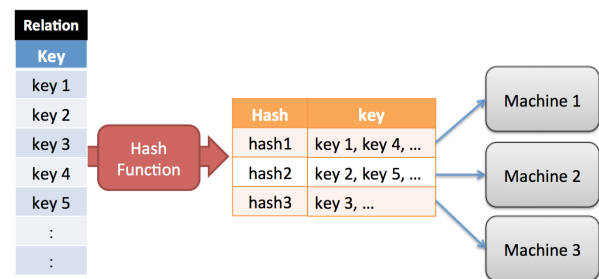


図 4 並列ハッシュ結合におけるハッシュ分散

数を定義し、その返り値に基づいて対応したパーティションへ均等になるように分割を行うことである。

並列ハッシュ結合におけるハッシュ分割とは、まず、メインマシンに読み込まれたリレーション内のタプルが、ハッシュ関数によりハッシュ値を獲得する。そして、そのハッシュ値を元に、対応した転送先のマシン（ノード）にタプル群が割り振られることである。これを図 4 に示す。

ハッシュ分割を行う利点は、リレーションが莫大なものであった時に小分けにすることで、大きな処理を並列かつ、非同期で作業できることや、キャッシュを有効に活用できることがあげられる。

3.2 アルゴリズム

この節では、並列ハッシュ結合で用いたアルゴリズムを記述する。

3.2.1 データ分散

データ分散についてのアルゴリズムを述べる。初めに、*MainMachine(master)* 上でのアルゴリズムについて記す。Algorithm 2 は、3.1 節で述べられたハッシュ分割のアルゴリズムである。2 つのリレーション R, S を読み込んだ master は、各リレーションのタプルを実行 $Node(worker)$ 数を用いたハッシュ関数にかけることで $R_i, S_i (i = 1, 2, \dots, n)$ にハッシングする。この時のハッシュ値である w_{id} は、

$worker$ の番号である．ここでこのアルゴリズムを用いてハッシングする理由は，結合条件を満たす可能性のあるタプル群を等しい $worker$ へ振り分けるためである．その後， R_i, S_i は $worker_i (i = 0, 1, \dots, n-1)$ へ転送される．

Algorithm 2 master: $Node$ 数を用いて分割

```

1:  $Node \leftarrow Node$  数
2:  $r \leftarrow R$  のタプル数,  $s \leftarrow S$  のタプル数
3:  $count_R[Node] \leftarrow 0, count_S[Node] \leftarrow 0$ 
4: for  $i = 0$  to  $r-1$  do
5:    $w_{id} = R[i].key \bmod Node$ 
6:    $NodeR[w_{id}][count_R[w_{id}]] = R[i]$ 
7:    $count_R[w_{id}] = count_R[w_{id}] + 1$ 
8: end for
9: for  $i = 0$  to  $s-1$  do
10:   $w_{id} = S[i].key \bmod Node$ 
11:   $NodeS[w_{id}][count_S[w_{id}]] = S[i]$ 
12:   $count_S[w_{id}] = count_S[w_{id}] + 1$ 
13: end for

```

3.2.2 照合処理

Algorithm 3 $worker_i$: ハッシュ表を作成

```

1:  $r_i \leftarrow R_i$  のタプル数
2:  $HashSize$ : ハッシュ表サイズ
3:  $Bucket[HashSize]$ : ハッシュ表
4: for  $n = 0$  to  $r_i - 1$  do
5:    $hash = \{(R_i[n].key - w_{id}) / Node\} \bmod HashSize$ 
6:    $Bucket[hash]$  で  $R_i[n]$  をリスト連結
7: end for

```

Algorithm 4 $worker_i$: 照合処理

```

1:  $s_i \leftarrow S_i$  のタプル数
2: for  $n = 0$  to  $s_i - 1$  do
3:    $hash = \{(S_i[n].key - w_{id}) / Node\} \bmod HashSize$ 
4:   for  $it = Bucket[hash] \rightarrow tuple$  リストの先頭 to 最後尾 do
5:     if  $it.key == S_i[n].key$  then
6:        $Result.R_i \leftarrow it$ 
7:        $Result.S_i \leftarrow S_i[n]$ 
8:     end if
9:   end for
10: end for

```

この節では， $worker$ での照合処理アルゴリズムについて述べていく． $worker_i$ は， $master$ より R_i, S_i を受け取る． $worker$ では，ハッシュ結合により照合処理を行う．初めに，Algorithm 3 を用いて R_i よりハッシュ表を 1 枚作成する．このハッシュ表と S_i を用いて照合処理を行う．その後，Algorithm 4 により， R_i, S_i の照合結果として $Result$ が生成される． $worker_i$ は $Result$ を $master$ へ転送する． $master$ は全 $worker$ の $Result$ を受け取ることにより， $R \bowtie S$ の結果を得られたことになる．

4. 実装

4.1 並列ハッシュ結合の詳細

この節では，並列ハッシュ結合の実装について順を追って説明していく．並列ハッシュ結合は C/C++ 言語で実装した．メインマシンとその転送先であるノードとの通信は IPoIB 通信を使用した．また，3.2.2 節で記述した Algorithm 4 はマルチスレッド方式で実装した．マルチスレッド方式には POSIX スレッド標準を実装したライブラリである Pthreads を使用した．int 型の二つの要素 “ key, val ” を持っている二つのリレーション R, S が与えられたとき，以下のクエリを実行する並列ハッシュ結合を実装した．

```

SELECT *
FROM R, S
WHERE R.key = S.key

```

並列ハッシュ結合の動作は次のようになっている．

- (1) メインマシン ($master$) に読み込まれた 2 つのリレーション R, S を転送ノード ($worker$) 数により R_i, S_i にハッシュ分割しノード ($worker$) へ転送する．
- (2) $worker$ は R_i よりハッシュ表を 1 枚作成する．
- (3) ハッシュ表と S_i より照合処理を行う．
- (4) 各 $worker$ の結合結果を $master$ へ転送する．

項目 1, 2, 3 について図を用いて詳しく説明する．

1. リレーションのハッシュ分割，転送

最初に，リレーションが $master$ へ読み込まれる．読み込まれたリレーション R, S は，3.2.1 節で記述した Algorithm 2 によって， R_i, S_i (i は $worker$ 数) にハッシュ分割される．その後，ハッシュ分割された R_i, S_i はハッシュ値に対応した各 $worker$ へ転送される．これを図 5 に示す．

2. $worker$ 内でのハッシュ表作成

$worker$ は分割されたリレーション R_i, S_i を受け取ると，最初に R_i よりハッシュ表をひとつ生成する．アルゴリズムは 3.2.1 節の Algorithm 3 を使用する．これを，図 6 に示す．

3. ハッシュ表と S_i の照合

ハッシュ表が生成された後，ひとつのハッシュ表と S_i をさらに分割した $S_i(n)$ をスレッドに引き渡す．その後，マルチスレッドで照合処理を行い照合結果を得る．これを，図 7 に示す．

5. 評価

本節では，実装した並列ハッシュ結合の評価，並列ハッシュ結合を対応させた PostgreSQL の性能評価について示す．

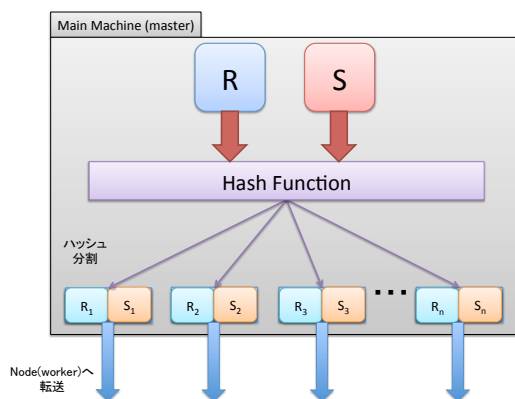


図 5 リレーションのハッシュ分割, 転送

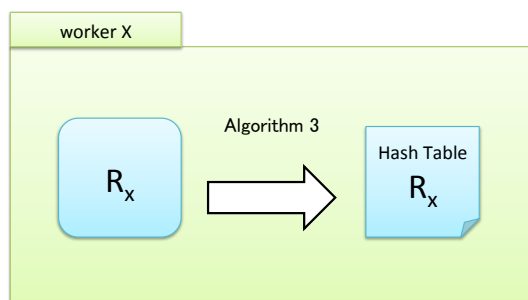


図 6 ハッシュ表の生成

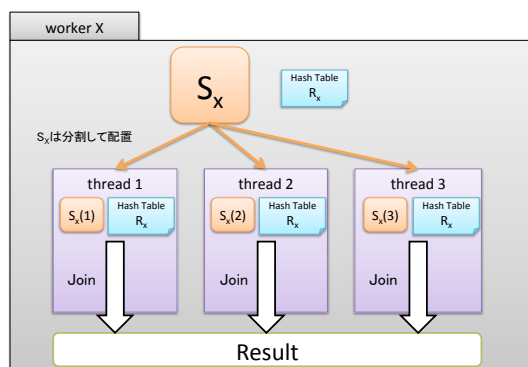


図 7 照合処理

表 2 実験環境

Main Machine	mds1
Node	ansys01 ~ ansys10
CPU	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz × 2
CPU cores	10 × 2
Main Machine Memory	128GB
Node Memory	64GB
OS	CentOS release 6.5 (Final)

5.1 実験環境

評価実験を行う環境を表 2 に示す。実装内にはマルチスレッドで処理を行う箇所がいくつかある。この時の最大スレッド数はいずれも最大物理コア数である 20 スレッドで実行し、評価している。

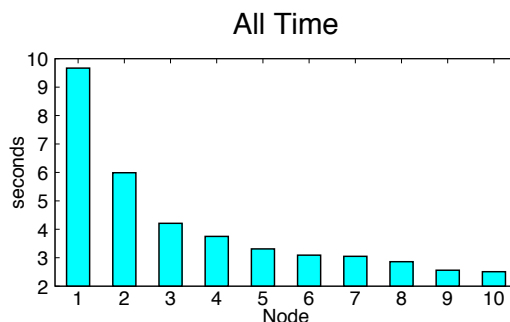


図 8 Node 数変化による全体実行時間

5.2 並列ハッシュ結合の評価

この実験では、2つのリレーション R, S の結合処理時間を測定する。2つのリレーション R, S はそれぞれ int 型の key, val の要素を持っている。具体的な実行クエリは、4.1 節で述べた以下のものである。

```

SELECT *
FROM R, S
WHERE R.key = S.key

```

5.2.1 Node 数変化による測定

表 3 データセット 1

R	10^8 TUPLE
S	10^8 TUPLE
最大 Node 数	10
HashSize	$10^8/5$
Selectivity	0

結合処理を行う Node (worker) 数を変えて測定を行った。表 3 は使用したデータセットである。リレーション R, S の TUPLE 数が等しい状態で結合処理を行った。Selectivity は 0 である。全体の処理時間を計測した。

図 8 は、TUPLE 数を 10^8 とした時の、Node 数の変化による全体実行時間の変化である。縦軸は全体の実行時間、横軸は Node 数を表している。Node 数が 1 である時 9.67 秒、Node 数が 10 である時 2.51 秒という結果が得られた。

図 9 は、図 8 中の Node 数が 1 である時と比較した Node 数変化による Speed Up を表している。縦軸は Speed Up、横軸は Node 数を表している。Node 数が 10 であるとき、Node 数が 1 である時と比較して 3.85 倍高速化する結果が得られた。

5.3 PostgreSQL との性能評価

5.3.1 条件設定

5.2 節で評価した並列ハッシュ結合を対応させた PostgreSQL と PostgreSQL における等結合の性能評価について述べる。それぞれ、リレーション R, S を用いて等しいクエリを実行する。PostgreSQL ヘクエリが発行されてから

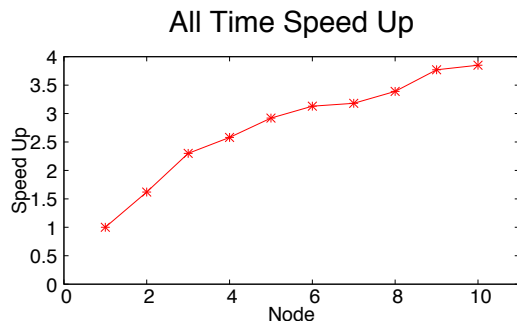


図 9 Node 数変化による全体実行時間の Speed Up

表 4 データセット 2

R	$10^5, 10^6, 10^7$ TUPLE
S	$10^5, 10^6, 10^7$ TUPLE
Selectivity	0
Parallel Hash Join	Node = 2
	HashSize = $10^5, 10^6, 10^7$

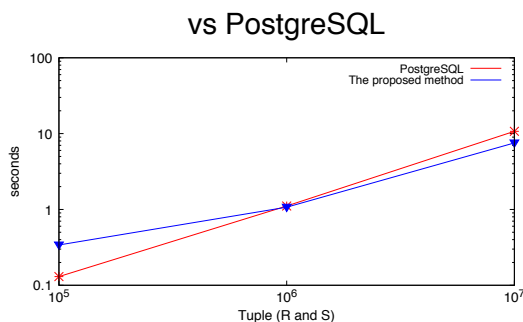


図 10 PostgreSQL と提案手法の比較

PostgreSQL へ結果が返ってくるまでの全体処理時間を測定した。データセットを表 4 に示す。

実行クエリ

実行クエリは以下の通りである。

```
SELECT *
FROM R,S
WHERE R.key = S.key
```

5.3.2 評価結果

5.3.1 節に従って、PostgreSQL で行う等結合と並列ハッシュ結合を対応させた PostgreSQL との測定を行った。結果を図 10 に示す。

図 10 の縦軸は全体の処理時間であり、横軸はリレーション R, S のタプル数である。タプル数 10^5 の時は、PostgreSQL の方が提案手法より高速に処理される結果であった。しかし、タプル数が $10^6, 10^7$ の時は提案手法の方が PostgreSQL 比べて高速に処理を行い、最大 1.42 倍の性能向上となる結果が得られた。以上より、タプル数が大きい場合 RDBMS における並列ハッシュ結合は、RDBMS における非並列な等結合よりも有効でことがわかる。RDBMS を運用するにあたって、高コストとなりやすい大きいサイ

ズの結合処理を並列ハッシュ結合を用いることによって低コストに抑えられるという結果が得られた。

6. 関連研究

6.1 Impala

Cloudera 社が開発した Impala は、Apache Hadoop [9] でネイティブに動作する、大規模な並列処理 (MPP) の SQL クエリエンジンである。Apache ライセンスのオープンソースである Impala プロジェクトが、最新のスケーラブルな並列データベース技術と強力な Hadoop を組み合わせた。これにより、ユーザーはデータの移行や変換をすることなく、HDFS や Apache HBase [10] においてクエリデータを直接格納することができる。Impala は、Hadoop エコシステムの基礎をなす一部として設計されており、MapReduce, Apache Hive, Apache Pig や他の Hadoop スタックコンポーネントで利用されているのと同じように、フレキシブルなファイルやデータフォーマット、メタデータ、セキュリティやリソース管理フレームワークを共有する [2]。

6.2 Hive

Hive は、オープンソースの大規模分散計算フレームワーク Hadoop 上で動作するデータウェアハウス (DWH) 向けのプロダクトである。Hive の特徴は、Hadoop 上の MapReduce (大量のデータを高速に処理するための分散処理フレームワーク) の処理を SQL 互換言語で操作を実行できることである。しかし、実際の Hive 処理の中身は、MapReduce の処理が動いている。「Select」文を実行すると裏で MapReduce の処理が走り、分散処理されて結果を得ることができる [11]。

6.3 SparkSQL

DataBricks 社が開発した SparkSQL は、構造化データを扱うための Spark のモジュールである [3]。Apache Spark は、Apache Hadoop エコシステムの凡庸的なオープンソースデータ処理フレームワークであり、パッチ処理やストリーミング処理、インタラクティブ分析を組み合わせた広範囲なビッグデータアプリケーションの開発が容易にかつ短期間で可能になる。

6.4 Presto

Presto は、Facebook 社が開発した数ペタバイト級の大規模データに対しても、対話的にアドホックな問合せを可能にする分散 SQL エンジンである。SQL が標準仕様であり ANSI SQL に準拠している。Hive は実行されると数段の MapReduce へと変換されるが、Presto は個々のタスクが並列で動く。また、中間データをメモリに持つことでタスク間のデータのやり取りが高速になり、Hive よりも CPU

の効率が優れ、データ取得時間が短くなる。また、HDFSに限らず、MySQL, PostgreSQL, Hive など複数のデータソースに対して接続し一度に集計を実行することが可能である [1]。

7. 結論と今後の課題

7.1 結論

並列ハッシュ結合を用いて結合処理を行った場合、Node (worker) 数の増加に伴い全体実行時間がより短くなることが確認できた。

RDBMS である PostgreSQL における等結合は Nested Loop Join, Merge Join, Hash Join の 3 つである。しかし、この 3 つは結合するリレーションが莫大なサイズのものであったり、複雑な結合条件の場合高コストとなってしまう。

そこで、本研究では PostgreSQL 上で並列ハッシュ結合を用いた結合処理を実装し評価した。リレーション R, S が 10^7 タプルであるとき、PostgreSQL での等結合には、10.73 秒かかる処理を、Node (worker) 数が 2 である並列ハッシュ結合を用いた PostgreSQL では 7.54 秒で処理することができるという結果が得られた。リリースされている PostgreSQL と比較して 1.42 倍の高速化を達成した。

以上より我々の実装した単純並列ハッシュ結合は、マルチコア・クラスタ環境の恩恵を十分に享受可能であり、PostgreSQL に組み込まれている等結合よりも性能を高められることが評価できた。

7.2 今後の課題

現在の並列ハッシュ結合の実装内では、メインマシンと Node 間の通信に IPoIB 通信を利用している。この実装であると、リレーションのサイズが大きくなると master と worker 間での転送コストが高くなってしまい性能が劣化してしまうことが確認できた。それなので、今回使用しなかった RDMA 転送をマシン間の通信に用いることで、転送コストがどのくらい軽減できるのか評価することが課題としてあげられる。

また、現在の並列ハッシュ結合は、結合結果のソート部分が未実装である。RDBMS 内で運用するためには重要な課題である。

謝辞 本研究の一部は、JST CREST「ポストペタスケールインテンシブサイエンスのためのシステムソフトウェア」、JST CREST「EBD: 次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」、JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」の支援を受けたものである。

参考文献

- [1] Presto — Distributed SQL Query Engine for Big Data <https://prestodb.io/>
- [2] Impala <http://impala.io/>
- [3] Spark SQL DataFrames — Apache Spark <http://spark.apache.org/sql/>
- [4] PostgreSQL Index Only Scan TECHSCORE BLOG <http://www.techscore.com/blog/2013/06/07/postgresql-index-only-scan->
- [5] ORACLE MASTER Bronze SQL 基礎 I 講座 (6) <http://www.atmarkit.co.jp/ait/articles/0510/07/news108.html>
- [6] SQL 表結合 (join) - 単純結合, 等価結合, 非等価結合, 外部結合, 再帰結合 - http://homepage2.nifty.com/sak/w_sak3/doc/sysbrd/sq_kj04_1.htm
- [7] Let's Postgres <http://lets.postgresql.jp/documents/technical/partitioning/2/>
- [8] Balkesen, C. Teubner, J. Alonso, G. Ozsu, M.T. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, 8-12 April 2013.
- [9] Apache Hadoop <https://hadoop.apache.org/>
- [10] Apache HBase <https://hbase.apache.org/>
- [11] Apache Hive <http://hive.apache.org/>