Map/Reduce におけるバケット再グループ化を用いたハイブリッドハッシュ結合アルゴリズム
A Hybrid Hash Join Algorithm with Bucket Regrouping on MapReduce

廣瀬 繁雄 ♥ 大森 匡 ♦
新谷 隆彦 ♠

Shigeo HIROSE Tadashi OHMORI
Takahiko SHINTANI

Map/Reduce 上の巨大データ処理においてもデータ集合 R と S の結合演算は重要な処理の 1 つであり，典型的には RepartitionJoin に代表される単純なハッシュ分割・マージソートで結合が行われている．しかし，n:m(多対多) の結合演算といった制約の緩い等結合などでは，わずかなデータの偏りでも結合の CPU コストが増加し，容易に負荷偏りが生じる．本稿ではこの n:m 結合を対象に，Map/Reduce 上のハイブリッドハッシュ結合において，ビルド処理の map/reduce ジョブでデータ偏りを検出し，プローブ処理の map/reduce ジョブで reducer 間の負荷分散を行なう方法 HSJ+BR を提案する．

Relational join between two data sets is still significant in mapreduce applications. This paper describes a new mapreduce algorithm of hybrid hash join over two data sets R and S in cases where many-to-many relationship between R and S is computed. To control the unbalance of data skew, our algorithm embeds a bucket regrouping strategy after the build-phase mapreduce job of R, so that appropriate partitioning be utilized in the mapreduce job of the probe phase of S.

1. はじめに

現在の巨大データ研究では, Web のアクセスログや科学技術データ, 多様なコンテンツ, リレーショナルデータ等, 様々な種類の大量データを対象にして価値のあるデータ空間に変換し, 傾向の特定や特異なデータの発見等を行うことが重視される [2][3]．こうした大規模データ処理は元々並列データベース分野で研究されてきたが, 現在では, そのデータ処理基盤として, Hadoop[1] に代表される Map/Reduce システムが使われ始めている．Map/Reduce 上の大量データ分析問題においても，データ集合を表すテーブル間の結合演算は重要な処理の 1 つであり，時間を多く使用する処理でもある．例えば Facebook などでは，ユーザが訪問したクリックストリームの分析のため，ユーザの情報テーブルとアクセスログ・テーブルの間の結合処理を絶えず数十分程度の間隔で数千台のノードで行っている [2].

Map/Reduce での結合演算は，元来並列データベースで行われていた結合演算アルゴリズムのうち，ナイーブなアルゴリズムである HashJoin が主に使われている [5]．Hadoop においても HashJoin は単純な MapReduce 処理によって実現されているが，並列データベースで行われてきた様々な結合演算の処理アルゴリズムや skew 処理技法等 [7][8][9][10] を鑑みると，現状の Hadoop での結合演算方式だけで十分とは考えられない．

Hadoop 上での結合演算は主にマスターテーブルとログテーブルの結合といった 1:n の等結合が多い．しかし, 類似度結合や θ-結合, ユーザ定義関数を用いた結合など [10][6], データ集合 R と S の要素間の多対多の関係性を計算する結合 (n:m 結合) 演算も重要である．例えば R を論文のデータ集合，S を別の論文集合，A を期間とすると，同じ期間の 2 つの論文集合の要素間の類似した組み合わせを求める条件式は，find (r, s) from R x S where r.A = s.A and f(r,s) という式で書くことが出来る．この場合 f(r,s) の類似度判定関数の内容によっては CPU コストが非常にかかるものと予想できる．また，こうした n:m の結合を Hadoop のような分散処理基盤でナイーブな HashJoin を行うと，1:n の結合に比べてノード毎の処理量の偏りが更に増大してしまう．

そこで本稿では，巨大なデータに対する頑健で効率的な Map/Reduce 上の n:m 結合演算のアルゴリズムの実装を行うことを目的として，パーティションの再グループ化により負荷分散を行う実行方法を提案する．また，結合条件が類似度や範囲制約といった θ 条件が与えられている n:m 結合においても，本質的には何らかの形で等結合の形で実行されることが多いことから [6][10]，本稿では n:m の等結合演算を対象として Hadoop 上の負荷分散実行方式を論じる．

以下, Hadoop 上の一般的な結合処理技法を 2. で述べ, 3. で Hadoop 上のハイブリッドハッシュ結合と簡単な改良法を提案した後, 4. で HSJ+BR と呼ぶ負荷分散実行方式を提案して, 5., 6. で評価とまとめを述べる．

2. Hadoop 上での一般的な結合演算

2.1 RepartitionJoin

Hadoop で結合演算を行う場合, 一般的にはハッシュ分割とマージソートを利用した RepartitionJoin と呼ばれる結合方法が用いられる [2]．即ち, データ集合を表す 2 つのテーブル R,S

♥ 学生会員 電気通信大学大学院情報システム学研究科 shi1153020@hol.is.uec.ac.jp
♦ 正会員 電気通信大学大学院情報システム学研究科 omori@is.uec.ac.jp
♠ 正会員 電気通信大学大学院情報システム学研究科 shintani@is.uec.ac.jp

の結合を行う場合，Map 処理では R,S の各レコードを結合キーでハッシュ分割を行う．Reduce 処理では R,S の各レコードは同様のハッシュ値で振り分けられるので，結合を行うことが出来る．

片方のデータに偏りが多少あっても計算コストが低いような，例えばマスターデータとログデータのような 1:n の等結合演算を行うだけの場合などでは RepartitionJoin で十分な速度となる [2]．しかし 1. で述べたような n:m の結合演算の場合，データに偏りがあると Reducer の CPU コストが非常に大きくなり，データの偏った 1 つのノードで非常に時間がかかってしまう．また，MultiJoin ではいくつかのテーブルのハッシュテーブルを作る必要があるなど，結合条件やデータによっては Reducer のメモリ制約が存在する [4]．

このようにデータが偏ることで処理が重くなってしまう場合，Hadoop では現在行っている処理を空いているノードに分担させ協調して動くような，動的に負荷分散を行う機能は存在しない．そのためサンプリングを行うことで予めに処理が偏るハッシュ値を求め，Map/Reduce ジョブの実行前に処理の分担させせると定義しておくといった，静的に負荷分散を実現している．

2.2 DirectedJoin

また，DirectedJoin と呼ばれる 2 つのテーブルを同じハッシュ値で分割し HDFS と呼ばれる共有ファイルシステムに保存し，Map 処理のみで結合を行う結合演算が存在する [2]．

2 つのテーブルのパーティショニングを行っているということは，テーブルの偏りを調べることも可能なので，ユーザが負荷分散機能を持たせることも可能である．また，Hadoop において 1 つの Mapper の担当するデータ範囲は，予め決められている HDFS のチャンクサイズに依存する．このため，Map のみで結合を行う場合，チャンクサイズによって負荷分散が行える場合も存在する．

しかしこの DirectedJoin では，両方のテーブルについてパーティショニングを行い，HDFS に保存を行わなければならないため，巨大データの場合この HDFS への保存のコストが大きい．

2.3 問題点の整理

既存のアルゴリズムの問題を以下にまとめる．

- 負荷分散を行うにはサンプリングを行うか，両方のテーブルをパーティショニングする必要がある
- Reducer のメモリ制約を満たさなければならない場合がある

そこで本稿ではこれら 2 点を解決するために，ハイブリッドハッシュ結合 [7] を用いることとした．

3. Hadoop 上でのハイブリッドハッシュ結合とその改良

3.1 Hadoop 上での単純なハイブリッドハッシュ結合

Hadoop 上の標準的なハイブリッドハッシュ結合アルゴリズムの説明に移る．図 1 にハイブリッドハッシュ結合の概略を示す．ハイブリッドハッシュ結合は，片側のテーブル R をパーティショニングするビルドフェイズと，もう片側のテーブル S と結合するプローブフェイズを，各々1 回の Map/Reduce ジョブで行うことで実現される．DirectedJoin に比べて片方のテーブル R のパーティショニングだけでよいため HDFS への保存のコストが少なく，更に R 側のデータの分布状況を調べることが出来る．

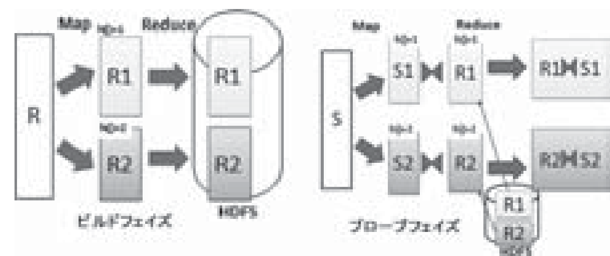

図 1 HybridHashJoin 概要図
Fig. 1 HybridHashJoin on M/R

Atta ら [11] は，Hadoop 上でのハイブリッドハッシュ結合においてレンジ分割でパーティショニングを行い，負荷分散に対応するため，RepartitionJoin と同様にビルドフェイズ前に R と S のデータのサンプリングを行って，各パーティションの担当レンジを調整する実装法を提案している．しかし，RepartitionJoin と同様にサンプリングを行えないようなデータの場合では対応することができない．また，プローブフェイズにおいてテーブル R のパーティションのハッシュテーブルを作成しているが，ビルドフェイズでパーティションをメモリサイズ以下に作らなければ実行できなくなる恐れがある．

3.2 HybridSkewJoin(HSJ)

我々は，まず，ビルドフェーズの出力である R のパーティションを，ある一定のサイズとすることで，R に対応した負荷分散が行えると考えた．同時に，頑健性を維持するため，R のパーティションを，後段のプローブフェイズ時に（当該パーティションの）ハッシュテーブルがメモリ上に収まる大きさに制限することにした．

具体的には，ビルドフェイズにおいて，Reduce が R のパーティションを作成する時に，プローブフェイズの Reduce 処理で作成する予定のハッシュテーブルが確実にメモリに乗ることが出来るような大きさとなるように更に分割を行なうこととした．例えば，R のパーティションサイズの上限が 256MB の時に，ある Reducer が 512MB のパーティションを振り分けられてしまった場合，これを 2 つの 256MB のパーティションに分割する．この 512MB パーティションは結合キーでソートされているため，パーティションの中でどこで区切ったかというレンジ情報を記録することで，Reduce 時に更に分割しても結合キーがどのパーティションであるかということを明確にすることが出来る．当該レンジ情報はビルド処理終了時にコントローラが集め，次のプローブ処理を開始する．

プローブフェイズでは，テーブル S に対して元々のハッシュ関数による分割を行い，得られたハッシュ値のパーティションが R

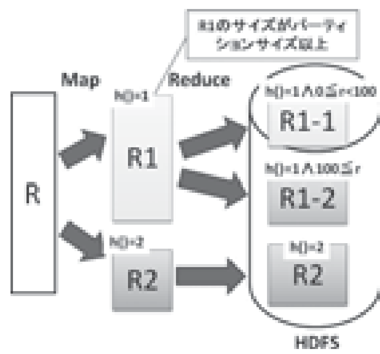

図 2　HSJ ビルドフェイズ
Fig. 2　HSJ build phase

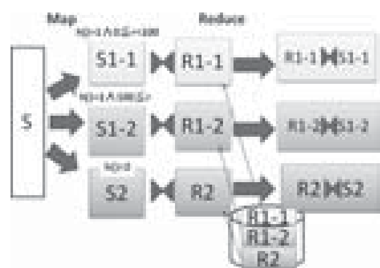

図 3　HSJ プローブフェイズ
Fig. 3　HSJ probe phase

の時にさらにレンジ分割されたものなら，当該レンジ情報を元に正しいパーティションへと振り分ける．そして, Reduce 処理が, 対応する R のパーティションとの間で結合を行う．以上のアルゴリズムを, HybridSkewJoin (HSJ) と呼ぶ．

以上の処理の概要を図 2，3 を用いて説明する．

1. ビルドフェイズの Map 処理ではテーブル R に対してハッシュ分割を行う (図 2).
2. Reduce 処理では出力パーティションが予め決めた一定サイズ以下になるように分割する．Reducer がパーティションを複数に分ける時には, 分割したレンジ (キーの範囲) 情報も同時に出力する．(図 2 中では R1 が大きい場合で R1-1 と R1-2 の 2 つにレンジ分割した).
3. プローブフェイズの Map 処理ではテーブル S を読み取り, key を結合キーとして, R と同じハッシュ関数で分割する．当該ハッシュ値に対応するパーティションが (上の (2) で) さらに分割されていた場合，レンジ情報も使って分割を行う．(図 3．同図では R1 が (2) で 2 つに分けられたため，S1 は R1 のレンジ情報により，R1-1 と R1-2 のどちらのレンジに属するか判断し分割を行っている．)
4. Reduce 処理では，分割情報を元に生成されたテーブル S のパーティションと，それに合致するテーブル R のパーティションを読み出し結合処理を行う．(図 3)

3.3　HybridSkewJoin with adaptive probe (HSJ+AP)

HSJ ではテーブル R をパーティションサイズの上限で分割することで負荷分散を行う．しかし，テーブル S の偏りに応じた負荷分散も行いたい．そのため, S を 2 つ（例えば 20% と 80%）に分割してプローブフェイズを 2 回に分けて実行し，1 回目の結果を元に 2 回目の処理の分散を行う方法が考えられる．この方法を HybridSkewJoin with adaptive probe(HSJ+AP) と呼ぶ．

例えば 1 回目のプローブフェイズで，ある R のパーティション p_i のデータが偏り処理が重くなってしまい，p_i を担当する Reduce ノードが他の Reduce ノードと比べて時間がかかっていたとする．このとき, 2 回目のプローブフェイズでは, p_i を担当する Reducer を増加させ, S のレコードが p_i の担当範囲ならば, 増加した複数の Reducer のうちどれか 1 つに振り分けて処理する．このように Reducer を増やすことで処理の分散化を行う．(詳細は [12] 参照).

4.　HybridSkewJoin with Bucket Regrouping(HSJ+BR)

4.1　解決すべき問題

HSJ では，ハッシュテーブルが作れる大きさにパーティションを区切ると定義した．しかし，3.3 のように処理が偏り 1 つのパーティションに対し複数の Reducer が計算を行う場合，プローブフェイズでのデータ移動のコストが増加するという問題がある．例を挙げると，パーティションの大きさを 256MB とし，この内の 20% が同一な結合キーのレコードとなり，この 20% のレコード集合の処理が重く負荷分散を行わなければならない場合である．複数の Reducer が 20% のレコード集合の処理のためにパーティション全体を読み込まなくてはならない．

また，ある結合キーのレコードが両方のテーブルに大量に存在する場合，この結合キーを担当する Reducer は非常に遅くなる．しかし片方のテーブルのみある結合キーのレコードが非常に多いが，もう片方はそれほど存在しない場合では，処理の内容とデータの量によるが極端に遅くはならない．このため，片方のテーブルのみでデータの偏りの大小を判定して Reducer の負荷分散を図ることは有用と考えられる．

4.2　アルゴリズム

そこで我々は，i) ビルドフェイズで本来のパーティションよりも小さいサイズのバケットと呼ぶ単位にデータを細分割しておき, ii) 適当なバケット群をパーティションへと再グループ化する構成案を決め, その構成案に沿ってプローブフェイズで各 Reducer が結合処理を行うことにした．バケットを細かく分け過ぎるとプローブ処理時の Reducer 数が増えすぎるため，処理が軽いと考えられるバケットはパーティションサイズに再グループ化することで，このグループを 1 つのパーティションとして扱う．逆に処理が重いと考えられるバケットはグループ化せず，単独のバケットをパーティションとして扱う戦略をとった．このため，ビルドフェイズでは, R の各バケットの度数分布 (結合キーに

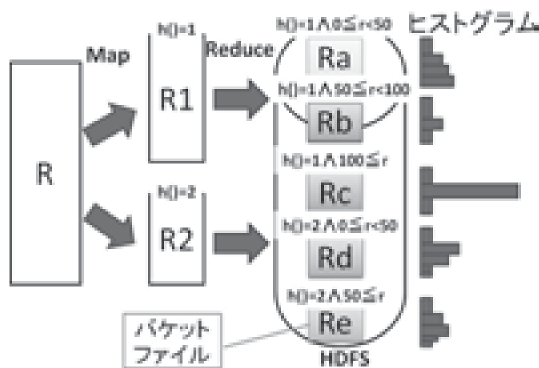

図 4　HSJ+BR ビルドフェイズ
Fig. 4　HSJ+BR build phase

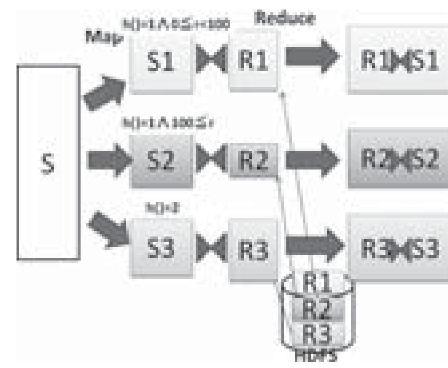

図 6　HSJ+BR プローブフェイズ
Fig. 6　HSJ+BR probe phase

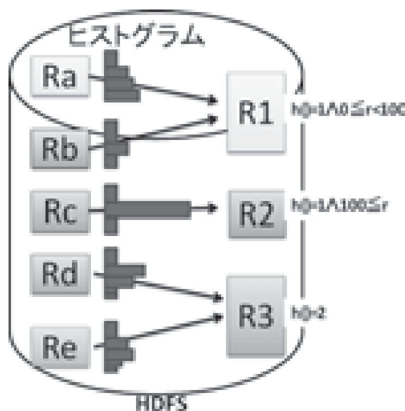

図 5　HSJ+BR 再グループ化
Fig. 5　HSJ+BR regrouping

関するレコードの分布情報) を生成することとした．この方法を HybridSkewJoin with Bucket Regrouping(HSJ+BR) と呼ぶ．

以上の処理の概要を図 4，5，6 を用いて説明する:

1. ビルドフェイズでは Map 処理でテーブル R に対してハッシュ分割を行う．(図 4).
2. Reduce 処理は一定のファイルサイズのバケットに分割して出力. 同時に, バケット毎の度数分布も出力する (図 4).
3. ビルドフェイズが終わったら再グループ化案をマスターノードが決める．即ち, 度数分布を元に適当なバケット集合を 1 パーティションとする構成案を決定する (図 5)．このとき, バケットのハッシュ値と結合キーのレンジ情報から, 新たなパーティションを構成する分割情報を記録する.
4. プローブフェイズの Map 処理ではテーブル S のみ読み取り, 上の (3) で作られた分割情報に従ってパーティショニングを行う (図 5).
5. Reduce 処理では, 各 Reducer は, 担当するパーティションに属す R の全バケットを読み取り, S の入力レコード列とでハッシュ結合を行う (図 6).

4.3　バケットの再グループ化と処理の分散化について

4.3.1　度数分布の生成

HSJ+BR では, ビルドフェイズのバケットのグループ化案を, 各バケットの度数分布を元に決定する．バケット単位の度数分布の求め方は, 次の 2 つが考えられる．即ち, (方法 1) 独立な結合キー毎に当該キーを持つレコード数をカウントして, その度数分布を直接使う方法と, (方法 2) 結合キーに適当なハッシュ関数を適用して, そのハッシュ値の分布を度数分布として用いる方法, である．今回は, 方法 1 を使用した.

この時, バケット内の全レコードが相異なる結合キーを持つ場合など, バケット内のデータ分布によっては度数分布の作成・利用は高コストになる．しかし, テーブル R 側のある結合キーを持つレコード数が少なければ，S 側の同じ結合キーのレコード数が極端に多くなければ結合処理時間は少なくてすむ．そこで, ある閾値を定め, バケットの中で閾値以上のレコード数を持つような結合キー k_n と，そのキーを持つレコード数 br_n だけを対象にし, $[k_n, br_n]$ を当該バケットの度数分布として出力することにした.

4.3.2　バケットの再グループ化の判定方法

4.3.1 により，ある一定以上のレコード数を持つ結合キーとそのレコード数によりバケット内の分布状況を求める方法を示した．この情報を元にバケットの処理の分散化と再グループ化の判定を行う.

ここで，バケット b_i のコスト c_i を， b_i の中である閾値以上のレコード数を持つ独立な結合キー k_n のレコード数 br_n の総和として与える．このコスト c_i が小さいようならば他の小さいバケットと再グループ化を行い，このグループをパーティションとして扱う．逆にコストが大きければ，バケット単体をパーティションとして扱う．また，分布状況によっては HSJ+AP のようにパーティションの Reducer 数を増加させる処理も行うことで, 処理の分散化を行う.

・再グループ化の判定

バケットの再グループ化によるパーティションの作成は，1 グループになるバケットのコストの和がある閾値 t よりも小さい場合に行う．この閾値は全体のコストとノード毎の平均予想コ

ストから求めることとした．よって t は, 全バケットのコストを $\sum_{i=1}^{all} c_i$，クラスタが同時実行できる Reducer 数を N_R とおいた場合，式 (1) となる．

$$t = \frac{\sum_{i=1}^{all} c_i}{N_R} \tag{1}$$

グループ内のバケットの最大個数はパーティションがハッシュテーブルを作れる大きさを上限とし，これらバケットのコストの和が t よりも大きくならない様に再グループ化を行う．

・バケットの分散化

コストが大きいと判断されるバケット b_i は，コスト c_i が式 (1) で求められる閾値 t よりも大きい場合と定めた．つまり式 (2) を満たす場合に b_i の分散化を行う．

$$c_i > \frac{\sum_{i=1}^{all} c_i}{N_R} \tag{2}$$

この時バケット b_i の担当 Reducer 数は，式 (3) を満たす α とすることでコストの大きいバケットの処理の分散化を行う．

$$\arg\min_{\alpha \in \mathbb{N}} \left(\frac{c_i}{\alpha} \leq \frac{\sum_{i=1}^{all} c_i}{N_R} \right) \tag{3}$$

また，再グループ化を行なったパーティションのコストがそれぞれ平均化されるように再グループ化を行う等，再グループ化を行うバケットの組み合わせの戦略も考えられるが，本稿では単純にファイル名でソートされた先頭のバケットから順番に上記の条件を満たすように再グループ化を行なった．

5. 実験

5.1 実行環境

実行環境は 1 台のマスターノードと 5 台のスレーブノード (12GB メモリ, CPU 2.6GHz) からなり，スレーブ 1 台が同時実行できる Map タスク，Reduce タスク共に 1 とした．(Hadoop 0.20.2, VineLinux6.0(64bit), レプリカ数 2)．

使用したデータはテーブル R，S 共に 1 レコード 100byte，結合キーを 50byte とし，共に 1GB (1000 万件) のデータとした．偏りとしてキー 1 が 50,000 レコード，キー 2 が 25,000 レコード，キー 3 が 12,500 レコード... といった傾き 1/2 となるような偏りのあるデータとし，R，S 共に同じ偏りの分布を持つデータとした．結合条件は R.key=S.key を条件とした n:m の等結合演算とし，計算部分として HDFS への出力を行わないが，結合結果をメモリ上に作成するとした．

5.2 結果

RepartitionJoin と HSJ，HSJ+AP，HSJ+BR 各々の全体実行時間を図 7 に示す．

パーティションのハッシュテーブルを作成できる最大のサイズは 256MB とした．プローブフェイズを 2 回に分ける HSJ+AP では，S は 20% と 80% に分けるとした．HSJ+BR での 1 バケットのサイズは 64MB とし，独立な結合キーのレコード数の閾値は 10 とした．

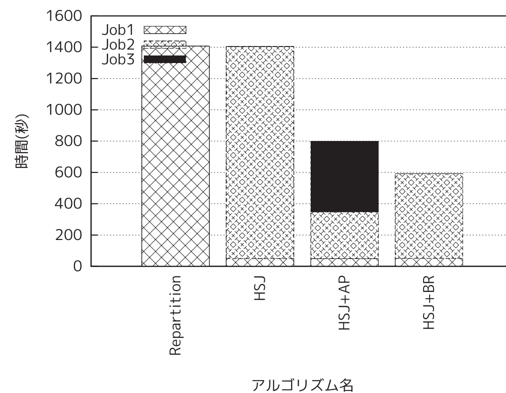

図 7　両方のテーブルに同様な偏りの分布がある場合 (R5 万 S5 万)

Fig. 7　total time of the four join methods

図 7 中の Job1 は RepartitionJoin では全体の実行時間，HSJ, HSJ+AP，HSJ+BR ではビルドフェイズの実行時間となる．Job2 以降はプローブフェイズの実行時間であり，HSJ+AP では Job3 が 2 回目のプローブフェイズの実行時間である．

また，このときの RepartitionJoin と HSJ，HSJ+BR におけるプローブフェイズの各ノードの Reduce 処理の所要時間をそれぞれ図 8 から 10 に示す．図中の Task1 とは各ノードが一番初めに行なった Reducer の時間であり，Task2 とは Task1 の Reducer の後に実行した Reducer の時間である．図 10 では Reducer が同時実行数以上存在したのでいくつかのノードは Reducer を複数回実行することとなった．

以上の結果を見ると, HSJ+BR は他の手法に比べ, 全体実行時間では RepartitionJoin と HSJ に比べ 2.3 倍程度の高速化を行い, reduce 処理ノードあたりの負荷分散も優れた結果となった．また, HSJ+AP のように明示的に負荷分散を図る手法よりも少ないジョブ数で負荷分散を実現できている．

最後に, 表 1 に, HSJ における各パーティションと，HSJ+BR のバケットによる再グループ化と分散化を行なった時の各パーティションの実行時間の比較を示す．HSJ+BR では, 実験では全てのパーティションは 4 バケットに分割された．Partition1 から 4 ではそのバケットが再グループ化により HSJ の場合と同じ範囲のパーティションとなった．Partition5 では 3 つのバケットが 1 つのパーティションとなった (表 1 の 14 秒)．残りの 1 つのバケットは 4 つの Reducer に分散して実行された．この 4 つに分散されたバケットは {} でまとめた処理時間となっている．

6. おわりに

本稿では Map/Reduce 上における n:m の結合演算において，データが偏り結果として全体の処理時間が低速になる場合に有効なアルゴリズムである Hybrid Skew Join with Bucket Regrouping (HSJ+BR) を提案した．このアルゴリズムでは細かいバケットに分割し，その度数分布からプローブ処理の負荷を

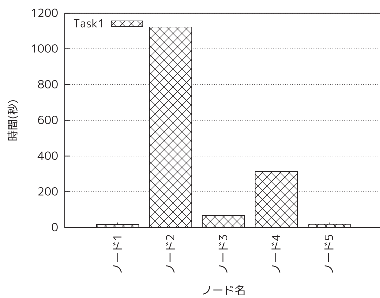

図 8 ノード毎の Reducer の所要時間 (RepartitionJoin)
Fig. 8 time/node(RepartitionJoin)

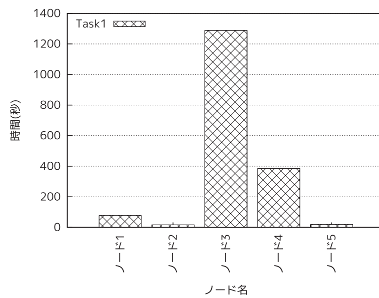

図 9 ノード毎の Reducer の所要時間 (HSJ)
Fig. 9 time/node(HSJ)

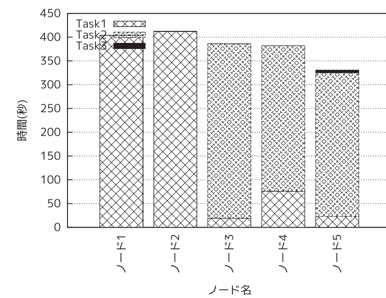

図 10 ノード毎の Reducer の所要時間 (HSJ+BR)
Fig. 10 time/node(HSJ+BR)

表 1 HSJ と HSJ+BR のパーティションの速度比較 (秒)
Table 1 reducer's time(sec.) per partition(HSJ,HSJ+BR)

	HSJ	HSJ+BR
Partition1	18	16
Partition2	17	19
Partition3	77	76
Partition4	389	412
Partition5	1322	14,{367,306,302,404}

推定し，バケットの再グループ化と処理の分散化の決定を行うという機構を用いた．結果として両方のテーブルに同じ分布の偏りがある場合では，提案方法の有効性を示せた．

今後検討すべき課題としては, R と S の分布が違う場合の負荷分散方法や, 各バケットのデータ偏りの度数分布情報を別の細粒度ハッシュ関数を使って計算してグループ化判定のコストを下げる手法など, が挙げられる．

最後に，本稿では n:m の等結合演算を扱ったが，θ-Join[6] や SetSimilarityJoin[10]，BandJoin といった様々な条件の結合においては，ネットワークコストや負荷分散の最適化モデルが更に重要になってくると考えられる．このような様々な条件の結合演算の高速化に対応することも今後の課題である．

[文献]

[1] Hadoop. http://hadoop.apache.org/.
[2] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, Y. Tian "A Comparison of Join Algorithms for Log Processing in MapReduce", ACM SIGMOD, pp. 975-986, 2010.
[3] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters", OSDI , pages 10-10, 2004.
[4] F. N. Afrati , J. D. Ullman. "Optimizing joins in a map-reduce environment", EDBT, pp. 99-110, 2010.
[5] 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", SIGMOD, pp. 165-178, 2009.
[6] A. Okcan, M. Riedewald "Processing theta-joins using mapreduce", ACM SIGMOD, pp. 949-960, 2011.
[7] D. J. DeWitt, et al., "GAMMA - A High Performance Dataflow Database Machine.", VLDB, pp. 228-237, 1986.
[8] D. J. DeWitt, J. F. Naughton, D. A. Scheneider, S.Seshadri, "Practical Skew Handling in Parallel Joins", VLDB, pp. 27-40, 1992.
[9] M. Nakayama, M. Kitsuregawa,M.Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", VLDB, pp. 468-478, 1988.
[10] S. Dhaudhuri, V. Ganti, R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning", ICDE, pp. 5-16, 2006.
[11] F.Atta, "Implementation and Analysis of Join Algorithms to handle skew for the Hadoop Map/Reduce Framework", Master's thesis, U.Edinburgh, 2010.
[12] 廣瀬, 大森, 新谷, "Map/Reduce におけるバケット再グループ化を用いたハイブリッドハッシュ結合アルゴリズム", DEIM フォーラム 2013, F2-4, 2013 年 3 月.

廣瀬 繁雄 Shigeo HIROSE
電気通信大学大学院修士課程 2013 年 3 月修了, 工学修士. 現 東芝 (株). DBSJ 学生会員.

大森 匡 Tadashi OHMORI
電気通信大学大学院情報システム学研究科・教授. ACM, 情処, DBSJ 各会員.

新谷 隆彦 Takahiko SHINTANI
電気通信大学大学院情報システム学研究科・准教授. ACM, 情処, DBSJ 各会員.