DEIM Forum 2013 F2-4

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

廣瀬　繁雄† 　大森　匡† 　新谷　隆彦†

† 電気通信大学大学院情報システム学研究科　〒 182–8585 東京都調布市調布ヶ丘 1-5-1
E-mail: †shi1153020@hol.is.uec.ac.jp, ††{omori,shintani}@is.uec.ac.jp

あらまし　Map/Reduce における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 つである．従来の研究では RepartitionJoin や DirectedJoin といった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，容易に負荷が生じる．本稿では Map/Reduce 上のハイブリッドハッシュ結合においてデータ偏りの検出と負荷分散を行なう方法を提案する．具体的には，ビルドフェイズの Map/Reduce 処理の中で同時にバケット内のデータ偏りの度数分布を調べ，そのバケットの再グループ化を行なうことで望ましいパーティション構成を求め，それによってプローブフェイズ時の reducer の負荷分散を行なう方法を述べる．

キーワード　MapReduce

Hybrid Hash Join Algorithm with Bucket Regrouping on Map/Reduce

Shigeo HIROSE†, Tadashi OHMORI†, and Takahiko SHINTANI†

† The University of Electro-Communications,Graduate School of Information Systems
1–5–1 Chofugaoka, Chofu, Tokyo, 182–8585 Japan
E-mail: †shi1153020@hol.is.uec.ac.jp, ††{omori,shintani}@is.uec.ac.jp

Abstract　Even in map/reduce applications, relational-style joins of big data sets under various constraints are significant. Currently repartition join or directed join is a major algorithm for 1-to-many equality-join. However, in cases of many-to-many join such as equality-join having weak constraints or general theta-join, data skew easily causes unbalance among reducers. To solve this problem, this paper describes a new algorithm of hybrid hash join using two map/reduce jobs. Intuitively, we embed a skew detection ability in reduce tasks of the build phase, and regroup resulting data-buckets into new partitions so as to decrease map/reduce overhead of the next probe phase.

Key words　MapReduce

1. はじめに

BigData と呼ばれるように，近年大量のデータが日々生成されている．BigData とは web のアクセスログやコンピュータによる科学技術計算の結果，多様なコンテンツやリレーショナルデータ等様々な種類のデータである．この大量なデータに対して価値のあるデータ空間への変換を行うことで，傾向の特定や特異なデータの発見等を行っている．

これらは元来並列データベースにおいても行われてきた処理だが，近年 Map/Reduce や NoSQL といったフレームワークやデータ管理システムが開発され，BigData に対して使用され始めている [2] [3]．これらはスキームを持たないような，不定長データを処理することを想定している場合や，並列分散性を高め台数効果を容易に出すことを目的としているといった特徴が存在する．本稿で使用する Hadoop [1] とは，大規模分散処理フレームワークである Map/Reduce [6] を参考にし，Apache が JAVA によって実装したフレームワークである．

この BigData の分析の中で，テーブルとテーブルとの結合演算は重要な処理の 1 つであるが，時間を多く使用する処理である．例えば Facebook などでは，ユーザーの情報テーブルとユーザーが訪問をしたクリックストリームのなどの分析を行うため，このテーブルの結合処理を絶えず数分や数十分といった間隔で，数千台のノードで行っている．

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 [8]．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アルゴリズムや

skewhandling 等を鑑みると [11] [12] [13]，現状の Hadoop での結合演算では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 の等結合が多いが，一方では，類似度結合やユーザー定義関数等を用いた結合など，データセット 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，パーティションの再グループ化により負荷分散を行う方法を提案する．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 [9] [14]，本稿では n:m の等結合演算を扱うとする．

以下 2. では Hadoop 上で一般的に行われている結合アルゴリズムの解説，3. ではハイブリッドハッシュ結合と改良方法の提案，4. では提案方法の詳細な説明，5. では実験結果を示す．

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 RepartitionJoin と呼ばれる，ハッシュ分割とマージソートを利用した結合方法が用いられる [4]．テーブル R,S といった 2 つのテーブルの結合を行う場合，Map 処理ではテーブル R,S のそれぞれの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 [5]．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 [7]．

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 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，ユーザー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 上において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．ハイブリッドハッシュ結合は片側のテーブル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と結合するプローブフェイズを 2 回の Map/Reduce ジョブを行うことで実現されている．DirectedJoin に比べて片方のテーブル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，更に片方のテーブルのデータの分布状況を調べることが出来る．

この時の処理の概要を図 1，2 を用いて説明する．

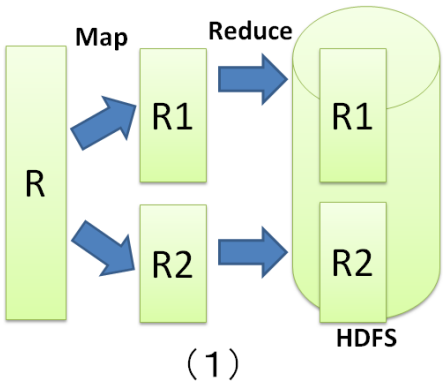

図 1　HybridHashJoin ビルドフェイズ

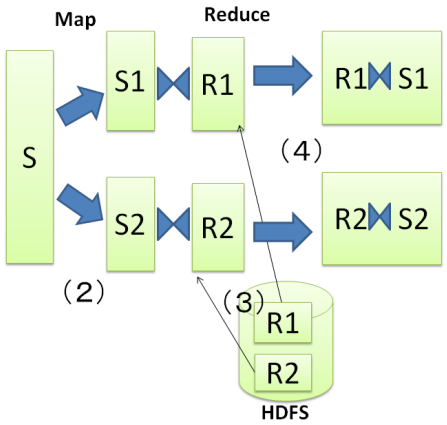

図 2　HybridHashJoin プローブフェイズ

（1） ビルドフェイズの Map/Reduce ジョブでは，テーブル R に対してハッシュ分割を行い，各 Reducer がそれぞれ HDFS に書きこむ．

（2） プローブフェイズの Map/Reduce ジョブの Map ではテーブル S を読み取り，そのまま key を結合キー，value を結合属性として出力し，テーブル R に使用したハッシュ値でハッシュ分割を行う．

（3） Reduce では，ハッシュ分割されたテーブル S のパーティションが存在する．このテーブル S に対応するテーブル R をそれぞれ HDFS より読み取り，ハッシュテーブルを作成する．

（4） Reduce 処理で (3) で作成したハッシュテーブルを用いて結合し出力する．

Fariha Atta らは，これら Hadoop 上でのハイブリッドハッシュ結合の，ハッシュ分割の部分をレンジ分割で実装した [15]．負荷分散に対応する場合，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い，各パーティションの担当レンジを調整することで実現していた．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)

まず，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 のパーティションがプローブフェイズ時にハッシュテーブルをメモリ上に作れる大きさとした．そこでビルドフェイズの Reduce の出力時に，プローブフェイズの Reduce 処理で作成する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，このアルゴリズムを HybridSkewJoin(HSJ) と呼ぶ．例えば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．

プローブフェイズではテーブル S に対してハッシュ値による分割を行う．得られたハッシュ値のパーティションが更に分割されていた場合，レンジ情報を元に対応するパーティションを求める．Reduce 処理では対応する R のパーティションと結合を行うこととした．

この時の処理の概要を図 3，4 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．

（2） Reduce 処理では出力のハッシュテーブルがメモリに乗るように分割し，Reducer がパーティションを複数に分ける必要があるのならば分割したレンジの位置を出力する．図 3 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，value を結合属性とし，テーブル R に使用したハッシュ値と，ハッシュ値に対応するパーティションが分割されている場合，レンジ情報も参考に分割を行う．図 4 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．

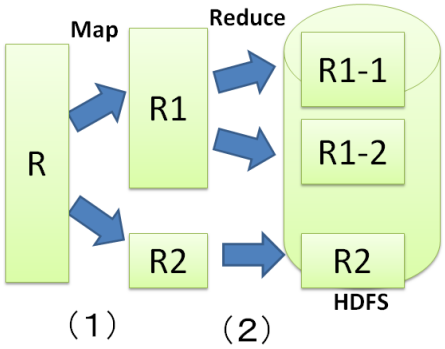

図 3　HSJ ビルドフェイズ

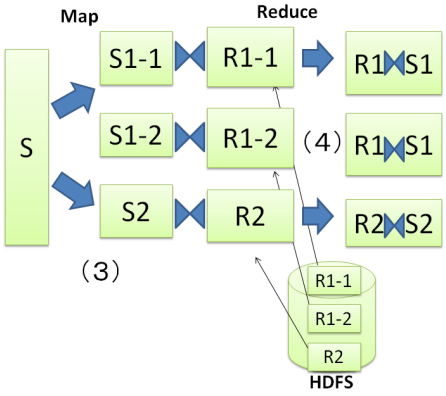

図 4　HSJ プローブフェイズ

3.3 HybridSkewJoin with adaptive probe(HSJ+AP)

3.3.1 アルゴリズム

HSJ ではテーブル R がパーティションサイズの上限で分割されることで負荷分散を行う．しかしこれでは不十分だと考え，テーブル S に対応した負荷分散を行うとした．具体的にはテーブル S を 2 つに分割し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こととした．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場合，2 回目のプローブフェイズではパーティション p を担当する Reducer を増加させる．2 回目のプローブフェイズでは，S のレコードが p の担当範囲ならば増加した複数の Reducer のうちどれかに振り分ける．このように予め Reducer を増やすことで処理の分散化を行う．

この時の処理の概要を図 5，6 を用いて説明する．

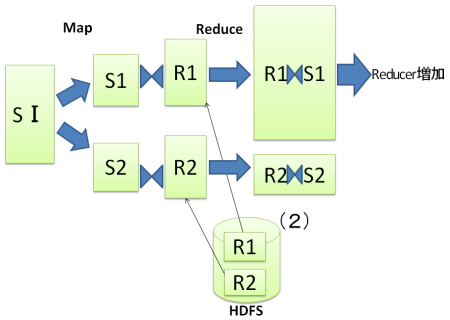

図 5　HSJ+AP 1 回目のプローブフェイズ

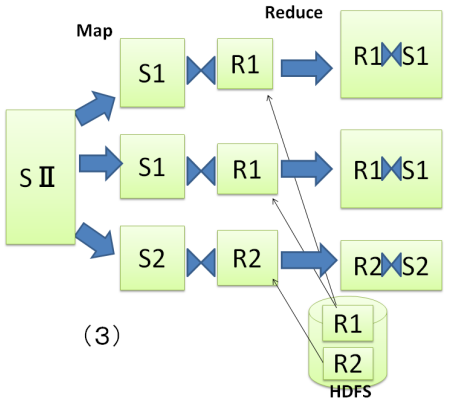

図 6　HSJ+AP 2 回目のプローブフェイズ

（1） ビルドフェイズと 1 回目のプローブフェイズは 3.2 と同様である．

（2） 処理の重い Reducer があるか判断を行う．式 (1) を満たす Reducer があるならば，対応する Reducer 数を α 個に増加させる．図 5 では R1 と S1 の結果が偏っていると判断し，R1 と S1 の Reducer 数を 2 とした．

（3） 2 回目のプローブフェイズでの Map 処理では，Key が 1 回目のプローブフェイズで時間のかかったパーティションならば複数の Reducer のうちどれか 1 つに出力する．図 6 では S1 のハッシュ値のレコードは 2 つの Reducer のうちどちらかに出力を行う．

（4） 同様に Reduce 処理では S のパーティションと合致するテーブル R のパーティションを読み出し結合処理を行う．

3.3.2 分散化を行う条件について

3.3.1 で述べた 1 回目のプローブフェイズの結果を元に，処理の分散化を行う時の条件について説明を行う．1 回目のプローブフェイズでは p_i 毎に結合条件に合致したレコード数を同時に計算し出力し，これを各パーティションのコスト c_i とおく．あるノードが明らかに他のノードよりもコストが大きくなる場合は，c_i がノード毎の平均コストを上回る場合である．各ノードの平均コストは総コスト $\frac{\sum_{i=1}^{all} c_i}{N_R}$ と同時実行できる Reducer 数 N_R から求めることができ，式 (1) を満たす c_i があるならば，そのパーティション p_i は分散を行うべきだと判断する．

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

式 (1) を満たす場合，ノード毎の平均総コストを下回るように P_i の担当 Reducer を出来る限り小さい値で増加させる．つまり式 (2) を満たす最小の α を増加後の Reducer 数とする．

$$\arg\min_{\alpha \in \mathbb{N}} \left(\frac{c_i}{\alpha} \leqq \frac{\sum_{i=1}^{all} c_i}{N_R} \right) \tag{2}$$

このように分散化を行うことで，最も終了時間の早いノードと最も終了時間の遅いノードの実行時間の差が最悪でも $\frac{\sum_{i=1}^{all} c_i}{N_R}$ とすることができる．

4. HybridSkewJoin with Bucket Regrouping(HSJ+BR)

4.1 問題定義

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してしまう場合が存在する．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することは有用であると考えられる．

4.2 アルゴリズム

そこで我々は，ビルドフェイズでパーティションよりも小さいサイズのバケットで分け，更にバケットを収集し再グループ化を行うこととした．バケットを細かく分けすぎるとプローブ

フェイズの Reducer が大量に増加してしまうため，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を生成することとした．この方法を HybridSkewJoin with Bucket Regrouping(HSJ+BR) と呼ぶ．

この時の処理の概要を図 7，8，9 を用いて説明する．

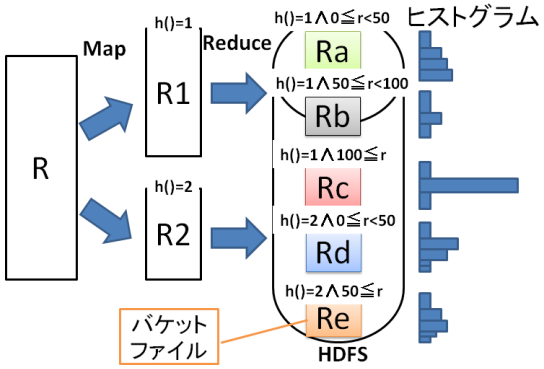

図 7　HSJ+BR ビルドフェイズ

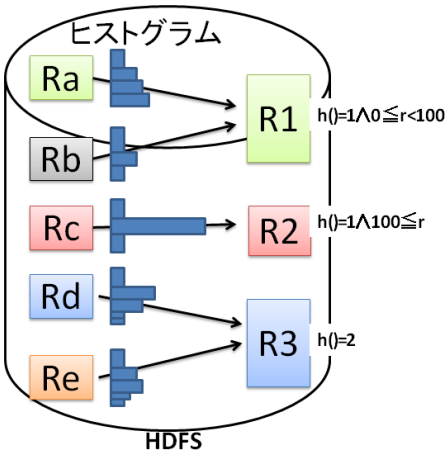

図 8　HSJ+BR 再グループ化

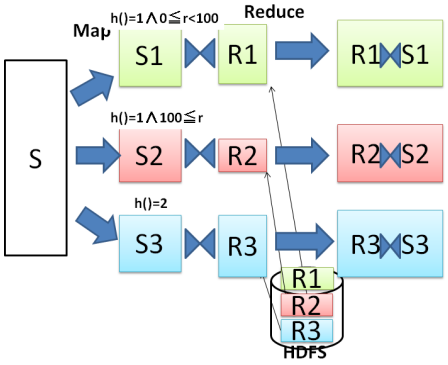

図 9　HSJ+BR プローブフェイズ

（1）　ビルドフェイズでは Map 処理でテーブル S に対してハッシュ分割を行う．

（2）　Reduce 処理で一定のファイルサイズでバケットを出力し，このとき同時にバケット毎の度数分布も出力する．

（3）　ビルドフェイズが終わったら再グループ化をマスターノードが行う．度数分布を元にバケットを纏めパーティションを作成する．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の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，(3) で作られた分割情報を元に出力する．

（5）　Reduce 処理では各 Reducer が対応するパーティションを読み取り結合を行う．

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時に，各バケットの度数分布を元に決定している．このバケットの度数分布の求め方は以下の 2 つの場合が考えられた．

（1）　独立な結合キー毎にレコード数をカウントする

（2）　結合キーに対して適当な長さのハッシュ関数を用いて，ハッシュテーブルの値で度数分布を得る

ここでは (1) の方法を使用した．バケットの分布状況を得るとき (1) を使う場合，バケット内の結合キーが全て異なる場合等，分布状況によっては作成と判定のコストが多くかかってしまう場合が存在する．しかしテーブル R 側のある独立な結合キーのレコード数が少なければ，それと同一なテーブル S 側の結合キーのレコード数は極端に多くならなければそれほど時間はかからないと考えられた．そこで独立な結合キーのレコード数に閾値を定めることで，閾値以上のレコード数を持つ独立な結合キー k_n と，そのレコード数 br_n を得ることが出来るように度数分布を出力するとした．

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，あるバケット b_i のコスト c_i を，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 の総和とする．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，グループを行う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 t は式 (1) と同様に，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 (3) となる．

$$t = \frac{\sum_{i=1}^{all} c_i}{N_R} \quad (3)$$

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，複数の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 (3) 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 (4) を満たす場合分散化を行う．

$$c_i > \frac{\sum_{i=1}^{all} c_i}{N_R} \quad (4)$$

この時バケット i の担当 Reducer 数は，式 (5) 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．

$$\arg\min_{\alpha \in \mathbb{N}} \left(\frac{c_i}{\alpha} \leqq \frac{\sum_{i=1}^{all} c_i}{N_R} \right) \quad (5)$$

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 のバージョンは 0.20.2 を使用し，OS は VineLinux6.0(64bit) とした．

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 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 共に同様な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 の結果を図 10 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 10 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+AP，

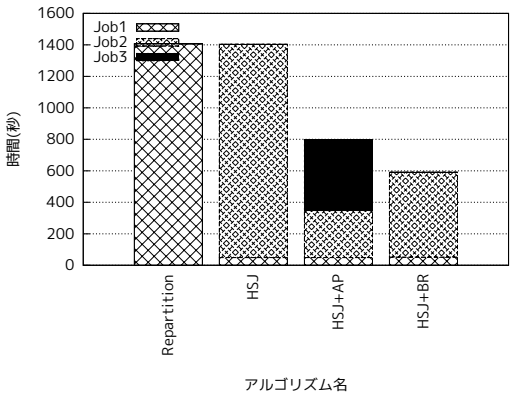

図 10　両方のテーブルに同様な偏りの分布がある場合 (R5 万 S5 万)

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 11 から 15 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 14，15 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．

表 1 には HSJ によるパーティションと，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パーティションの実行時間の比較を示す．実験では全てのパーティションは 4 つの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 と同様の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 {} でまとめた時間となった．

再グループ化を行う場合，HSJ でのパーティションを跨いでグループ化を行うため，HSJ のパーティションと HSJ+BR のパーティションは一致しない場合が存在するが，本稿のデータでは HSJ のパーティションを跨いでグループ化を行うことはなかったため，表 1 のように示す．

表 1　HSJ と HSJ+BR のパーティションの速度比較 (秒)

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

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

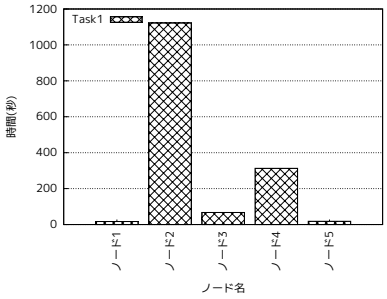

図 11　ノード毎の Reducer の所要時間 (RepartitionJoin)

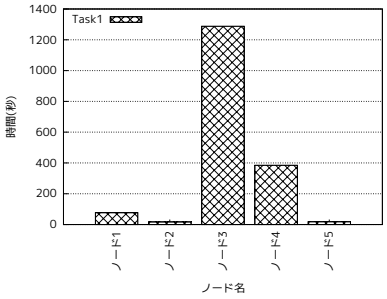

図 12　ノード毎の Reducer の所要時間 (HSJ)

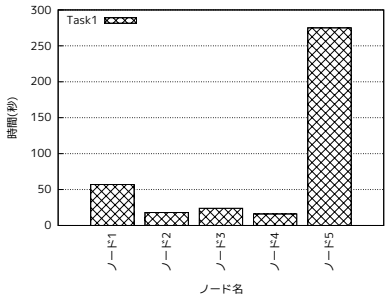

図 13　ノード毎の Reducer の所要時間 (HSJ+AP Job2)

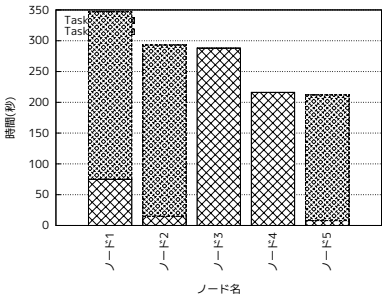

図 14　ノード毎の Reducer の所要時間 (HSJ+AP Job3)

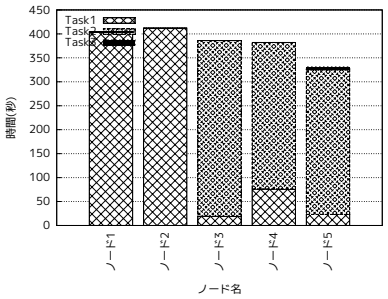

図 15　ノード毎の Reducer の所要時間 (HSJ+BR)

の偏りがある場合では，提案方法の有効性を示せた．

しかし，R と S が分布の違う偏りの場合や，R は偏っているが S が一様分布な場合などで追加実験を行う必要がある．

また，各バケットの度数分布では単純に独立な結合キー毎にレコード数をカウントし，閾値を設けることでマスターノードにおける再グループ化と分散化の処理量を減らしている．しかし汎用性の観点からある程度の長さのハッシュ値を用いた分布判定を行なうような，Fine-grain なハッシュテーブルを用いたほうが普遍性があると考えられる．

最後に，本稿では n:m の等結合演算を扱ったが，θ-Join や SetSimilarityJoin，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．

謝　　辞

本研究の一部は，科研費 (課題番号：24500109) の助成によるものである．

文　　献

[1] Hadoop. http://hadoop.apache.org/.
[2] MongoDB. http://www.mongodb.org/.
[3] Cassandra. http://cassandra.apache.org/.
[4] J. Lin, C. Dyer. "Data-Intensive Text Processing With MapReduce", Morgan and Claypool Publishers.
[5] 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.
[6] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters", OSDI , pages 10-10, 2004.
[7] F. N. Afrati , J. D. Ullman. "Optimizing joins in a map-reduce environment", EDBT, pp. 99-110, 2010.
[8] 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.
[9] A. Okcan, M. Riedewald "Processing theta-joins using mapreduce", ACM SIGMOD, pp. 949-960, 2011.
[10] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets", VLDB, pp.1265-1276, 2008.
[11] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. "GAMMA - A High Performance Dataflow Database Machine.", VLDB, pp. 228-237, 1986.
[12] D. J. DeWitt, J. F. Naughton, D. A. Scheneider, S.Seshadri, "Practical Skew Handling in Parallel Joins", VLDB, pp. 27-40, 1992.
[13] M. Nakayama, M. Kitsuregawa,M.Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", VLDB, pp. 468-478, 1988.
[14] S. Dhaudhuri, V. Ganti, R. Kaushik "A Primitive Operator for Similarity Joins in Data Cleaning", ICDE, pp. 5-16, 2006.
[15] Fariha Atta, "Implementation and Analysis of Join Algorithms to handle skew for the Hadoop Map/Reduce Framework", Master's thesis, University of Edinburgh, 2010.