MapReduce環境におけるアドホックなクエリを対象とした，Adaptive indexing適用モデルの提案とその評価

奥寺　昇平†　　横山　大作†　　中野美由紀†　　喜連川　優†

† 東京大学生産技術研究所　〒 153–8505　東京都目黒区駒場 4-6-1
E-mail: †{okudera,yokoyama,miyuki,kitsure}@tkl.iis.u-tokyo.ac.jp

あらまし　近年のデジタルデータの急増に伴い，大規模データを効率的に処理することが求められている．MapReduce環境は大規模分散データ処理に適しているが，常にデータスキャンを行うため，入出力コストが高い．本論文では，入出力コストを削減するために，MapReduce環境に適用的インデックスを導入した統合フレームワークを提案する．フレームワークでは，同じカラムに対して異なる選択条件が与えられた時に，問い合わせが繰り返されるにつれ処理を高速化する．また，データの更新時に性能を大きく劣化させない．本論文では，提案手法を，基本的なデータ処理に適用し，実行時間の削減効果を検証した．シミュレーションによる評価を行い，少しずつ選択条件が変化する問い合わせを繰り返し発行し，累積の実行時間を比較した．その結果，インデックスを利用しないデータ処理と比べ，提案手法が大きく実行時間を削減することを確認した．

キーワード　MapReduce，適応的なインデックス，シミュレーション，アドホック問い合わせ

Modeling and evaluation on Ad hoc query processing with Adaptive Index in Map Reduce Environment

Shohei OKUDERA†, Daisaku YOKOYMA†, Miyuki NAKANO†, and Masaru KITSUREGAWA†

† Institute of Industrial Science，the University of Tokyo
Komaba 4–6–1，Meguro-ku，Tokyo，153–8505 Japan
E-mail: †{okudera,yokoyama,miyuki,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract　We are required to process large-scale data efficiently as the volume of digital data we generate increases rapidly. MapReduce environment is suitable for large-scale distributed data processing and getting more important as a ad-hoc data analysis platform. However, all data scan is required every query when performing ad-hoc analysis in MapReduce environment, so the I/O cost will be badly huge. In this paper, in order to reduce the I/O cost, we propose an integrated MapReduce framework with adaptive index. This framework gradually shrinks the processing cost, as those queries are issued, each of which has a selection condition on the same column but different selection values. And the framework can keep the performance when data updates happens. We applied our proposed adaptive indexing technique into basic data operations *projection, aggregation, join* and evaluated the reduction of the execution time. We conducted the simulation and compared the accumulated execution time in the workload where the queries with the selection conditions slightly changed are repeatedly issued. As a result of simulation, our proposed framework showed higher performance than those processing without adaptive indexing.

Key words　MapReduce，Adaptive indexing，Ad-hoc query，Simulation

1. はじめに

今日，我々の社会が生み出すデジタルデータが急増している．SNS サイトの Facebook では，2008 年 8 月に 1 億人だったユーザー数が 2013 年 1 月には 10 億超に急増し，ユーザーによって投稿され共有されるコンテンツ数は月間 300 億に及ぶと報告されている [1], [2]．また, インターネットオークションサイト大手の eBay では，2 億人に近いアクティブなユーザーと 3 億 5000 万の商品を抱え，ユーザーによる商品の売買で毎日大量のデータが生成されている．また，我々が扱うデータの種類が多様化している．従来の構造的なデータに加え，アクセス履歴やセンサデータといったログデータ，マイクロブログなど

のソーシャルデータ，画像や地図情報といった多次元データなど様々なデータが存在する．Web 企業を代表とする大規模データを管理する組織では，こうした多種多様で日々増加し続ける大規模データを，データ解析基盤に蓄積し即時的にデータ分析を行い，組織の活動に活かすことが求められている．

大規模データの典型的な解析として，アドホック解析がある．アドホック解析とは，トライアンドエラーで興味深いデータの特性や関係性を探っていく解析方法を指す．データについて事前知識がない場合，データの特徴をすぐに把握することは困難である．そのため，データを様々な切り口で，繰り返しデータを分析することで，データの特徴を探していく．

MapReduce フレームワーク [3] とそのオープンソース実装である Hadoop [4] は大規模データの解析基盤として主流なプラットフォームであり，MapReduce 環境ではアドホック解析が頻繁に行われている [5]．しかしながら，MapReduce 処理は大規模なデータスキャンを中心としており，毎回のデータ読み込み負荷が高い．

本稿では，MapReduce 環境における大規模データ解析の高速化のために，MapReduce 環境にクエリに適用的なインデックスを導入した統合フレームワークを提案する．提案フレームワークでは，同じカラムに対して異なる選択条件が与えられた場合，問い合わせが繰り返されるにつれ処理を高速化する．提案する適応的インデキシング技術を基本的なデータ処理である射影演算処理，集合演算処理，結合演算処理に適用し，実行時間の削減効果を検証した．シミュレーションによる評価を行い，少しずつ選択条件が変化する問い合わせを繰り返し発行し，累積の実行時間を比較した．シミュレーション結果から，適用的なインデックスを利用しないデータ処理と比較し，高い性能が得られることを示した．

本論文の構成は，以下の通りである．第 2 章で MapReduce 環境におけるアドホック解析を説明し，第 3 章では適応的インデキシング統合フレームワークを提案する．第 4 章では MapReduce の実行時間に関するコストモデルを構築し，第 5 章ではコストモデルを基にしたシミュレーションで提案手法を評価する．最後に，第 6 章で関連研究を，第 7 章でまとめを述べる．

2. MapReduce 環境におけるアドホック解析

この説では，MapReduce 環境におけるアドホック解析における問題点を明らかにする．本稿で想定する問い合わせを下記に示す．問い合わせは，射影演算，集合演算，結合演算の 3 つの基本的なデータ処理であり，すべてレンジ条件をもつ．また，選択条件に *discount* カラムが指定されているが，データ解析を行う前に選択条件に指定されるカラムを予測することはできないと想定する．

(1) 射影演算

```
SELECT id, discount FROM R WHERE x <= discount < y;
```

(2) 集合演算

```
SELECT partID, AVE(discount) FROM R
WHERE x <= discount < y GROUP BY parrID
```

(3) 結合演算

```
SELECT S.partID, S.partName, R.discount FROM R, S
WHERE x <= R.discount < y AND R.partID = S.partID
```

2.1 データ解析処理の実行フロー

MapReduce 環境におけるデータ解析処理の実行フローを説明する．本稿では，現在もっとも主流なオープンソース実装である Hadoop の MapReduce 処理を想定する．アドホック解析では前述したような選択条件を持つ基本的なデータ処理が繰り返し発行される．そこで，基本的なデータ演算である射影演算，選択演算，結合演算，集合演算を MapReduce 上で実行するフローを述べる．MapReduce では，Map フェーズ，Reduce フェーズの 2 段階でデータ処理を行い，各フェーズでは，複数タスクがデータ並列で並列処理を行う．Map タスクでは，通常 Hadoop 分散ファイルシステム（HDFS）に格納されたデータをスキャンし，選択条件による選択処理や射影処理などの map 処理を行う．Reduce タスクでは，Map タスクから分散されたレコードを集積し，集合演算，結合演算などの Reduce 処理を行い，処理結果を HDFS に出力する．

図 1 は，Map タスクにおけるレンジ条件 $20 \leqq discount < 80$ を持つクエリの実行フローを表している．Map タスクでは，データスプリットと呼ばれる分割データ単位をスキャンし，レコードに変換後，Map ステップにおいてレンジ条件判定と射影演算を行う．そして，Map ステップから出力したレコードを Collect, Spill, Merge といったステップで処理し，Reduce タスクに分配する．Reduce タスクでは，Map タスクから分散されたレコードを集積し，集合演算，結合演算などの Reduce 処理を行い，処理結果を HDFS に出力する．

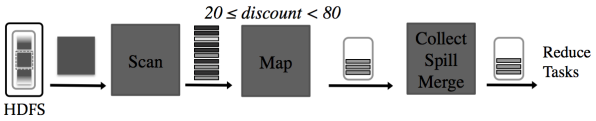

図 1　MapReduce 環境における選択条件を持つクエリの処理フロー

2.2 アドホック解析の問題点

MapReduce 環境におけるアドホックな解析の問題点は，大規模なデータスキャンを中心としており，処理コストが高いことである．アドホックな解析では，ユーザーが興味を持ったカラムに対して，少しずつレコードの選択条件を変えながらクエリを繰り返し発行する傾向がある．例えば，図 1 で発行したクエリ結果にユーザーが興味をもち，さらに選択度の高いレンジ条件 $30 \leqq discount < 50$ をもつクエリを発行したとする．この場合，再度，全データスキャンを行い，同じようなデータ処理を繰り返し行う．このように，MapReduce 処理では，同じような選択条件を持つ問い合わせを発行したにもかかわらず，毎回全データを読み込まなければならない．我々は，このデータ入出力コストを削減することで，全体のデータ解析を高速化できると考えた．

3. 適応的インデキシング統合フレームワーク

MapReduce 環境でのアドホック解析におけるデータ入出力コストを削減するために，MapReduce 環境にクエリ実行時にインデックスを作成，更新する Adaptive index 技術 [6], [7] を導入する．MapReduce 処理では，クエリ実行時に必ずすべてのデータを読見込まなければならないので，インデックス作成オーベーヘッドは小さいと期待できる．Adaptive indexing 技術の導入により，データアクセス方式を全データスキャンからインデックスアクセスに変更し，不要な I/O コストの削減する．さらに，クエリ実行時にインデックスを作成することで，アドホックな処理に対して柔軟に対応する．

3.1 フレームワーク概要

提案する適応的インデキシング統合フレームワークの概要を述べる．図 2 が提案フレームワークの全体象を表す．提案フレームワークでは，データスプリット毎にインデックスを構築する（図 2 左）．MapReduce 処理がもつ Map タスクでのデータ処理の独立性を失わないように，Map タスクのデータ処理単位であるデータスプリットにインデックスを構築する設計を取った．また，Map タスク実行中にインデックスの構築を行う．Map タスクでは，担当のデータスプリットに対してデータ処理を行うと同時に，インデックスを構築していく．インデックスが存在しない場合にはインデックスを作成し，インデックスが既に存在する場合にはインデックスを更新する．インデックスの保管先に関しては，各インデックスを HDFS 上の単一ファイルとして管理する．これは，HDFS がファイルの部分内書込が出来ない制約の中で，入出力コストを低く押さえインデックスを更新していくためである．インデックスの更新時には，古いインデックスファイルを削除し，新しいファイルを作成する．

インデックスの構造と構築方法に関しては，データベース分野で提案された Adaptive indexing 技術の中から，Database Cracking [6] を参考にした．インデックスの実際の構造は，全レコードへのポインタ配列と，レコードポインタへのポインタをエントリにもつインデックスツリーで構成されている（図 2 右）．さらに，レコードポインタはインデックスツリーのエントリの値で，レンジ分割されている（図 2 では，20 以下の値の赤色のレンジ，20 以上 80 以下の黄色のレンジ，80 以上の値の青色のレンジの 3 つにわかれている）．

提案フレームワークは，レンジ条件による選択条件をもつクエリを対象にする．クエリ応答時には，始めにインデックスツリーを検索し適切なレンジを選び，レンジに含まれる各レコードポインタごとにランダムアクセスでデータを取得していく．また，クエリ応答ごとにクエリレンジに応じてレコードポインタ配列をレンジ分割する．これにより，よくクエリされるレンジほどレンジ領域が細かく分割され，問い合わせが繰り返されるにつれ，応答性能を高めていく．データ更新時におけるインデックス管理に関しては，インデックスの構築同様にデータスプリット毎に行う．この章では，提案フレームワークにおけるインデックスの作成方法，条件値変更時のインデックス更新方法，データ更新時のインデックス更新方法を説明する．

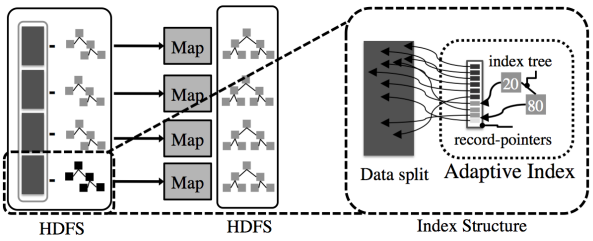

図 2　適応的インデキシング統合フレームワークの概要

3.2 インデックスの作成

レンジ条件に指定されたカラムのインデックスが存在しない場合に，インデックスの作成を行う（図 3）．提案フレームワークでは，Map ステップでインデックスの作成を開始し，データ処理終了後にインデックスを完成させ，分散ファイルシステムに書き込む．インデックスの作成は，1. レコードポインタ配列の作成，2. インデックスツリーの構築の順番に行う．

まず，Map ステップにおいて，レコード選択されなかったレコードを含めたすべてのレコード関してレコードに対するポインタを作成し，クエリレンジ $20 \leqq discount < 80$ に応じて異なるバッファに一時的に格納する（$discount < 20$ を満たすレコードポインタ, $20 \leqq discount < 80$ を満たすレコードポインタ, $80 \leqq discount$ を満たすレコードポインタを 3 つのバッファに分けて格納した）．そして，すべてのデータ処理終了後に，バッファを順番に連結させ，レンジ分割されたレコードポインタ配列を取得する．次に，レコードポインタ配列のレンジの開始アドレスに対してインデックスエントリ（図では，20, 80 と表記した茶色いオブジェクトで表記) を作成し，インデックスツリーに格納することで，インデックスツリーを完成させる．これで，インデックスの生成が完了し，生成したインデックスを HDFS の単一ファイルとして書き込み行います．

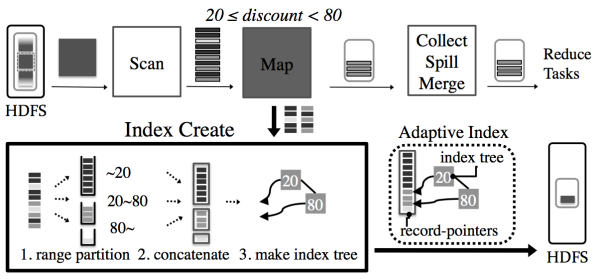

図 3　インデックスの作成フロー

3.2.1 条件値変更時のインデックスの更新

条件節で指定されたカラムのインデックスが存在し，かつ条件節の値が新しい値に変更された時にインデックスの更新を行う．ここでは，先ほどのクエリの直後に，レンジ条件 50 以上 70 以下のクエリが発行されたとする．図 4 は，条件値変更時のインデックスの更新フローを表す．提案フレームワークでは，まず，インデックスすべてを分散ファイルシステムから読み込みメモリに展開後，インデックスを検索してクエリに必要なレンジを取得する（図では，前回インデキシングを行った黄色のレンジ 20 ～ 80 を取得した）．次に，このレンジに含まれるレ

コードポインタを用いてランダム・アクセスでレコードを取得する．取得したレコードを従来どおりのデータ処理を行いクエリに応答すると同時に，Map ステップにおいてインデックスの更新を開始する．インデックスの更新では，インデックス生成と同じフローで，Map で作成したレコードポインタをレンジ分割しする．最後に入力したインデックスとは別のファイルとしてインデックスを HDFS に書き込むことで，インデックスの更新が完了する．

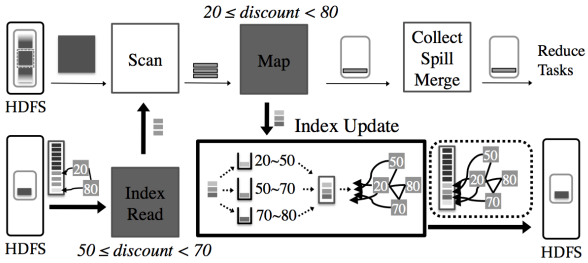

図 4　条件変更時のインデックス更新フロー

3.2.2　データ更新時のインデックスの更新

インデックス管理手法の概要を述べる．HDFS では，ファイル粒度でデータ更新を検知するため，データの更新も通常はファイル粒度で行う．図 5 左は，インデックスが構築されたデータを更新した様子を示している．今回，真ん中のデータスプリットが更新されたとする．この場合のインデックス管理方法に関して，3 つの手法を提案する．提案インデックス管理手法の概要を述べる．

まず，1 つ目は最もナイーブな手法（図 5 の 1.*all-delete*）である．この手法では，前述したように素の HDFS ではファイル粒度に更新を検知することを理由に，データ更新時にインデックスをすべて破棄する．

2 つ目は，データスプリットごとのインデックス破棄を行う手法（図 5 の 2.*split-delete*）である．HDFS を改変することで，データスプリットごとの更新が可能である．HDFS ではデータスプリットごとに hash 値を保持しており，この hash 値を利用して更新前後のデータスプリットを比較することで，更新をチェックする．hash 値が一致する場合にはインデックスを利用し，hash 値が異なる場合にはインデックスを破棄する．

最後の手法では，データの更新元でデータの更新履歴を管理していて，データ更新の際にデータと共にこの更新情報も取得できる状況を考える．この場合，この外部情報を利用して，データスプリットの中のどのレコードが更新されたかを知ることが可能である．そこで，すべてのインデックスを継続利用し，クエリ実行時に更新情報を参照にインデックスを更新していく（図 5 の 3.*partial-update*）．クエリ実行時には，インデックスで検索後，検索結果を更新履歴を参照して修正し，データアクセスを行う．修正した内容はインデックスの更新に反映する．

3.2.3　インデックス運用コストの低減

ここでは，インデックスを運用する上で必要となるコストに関して，そのコストを低減する改良手法を説明する．インデックス運用における課題として，特に 2 つが挙げられる．1 つめ

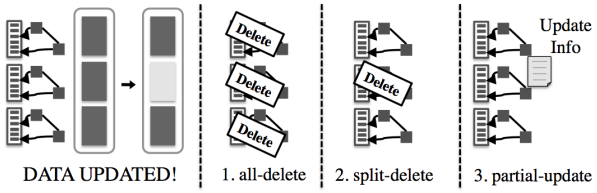

図 5　データ更新時における 3 つのインデックス管理手法

は，アダプティブインデックスの特性として，インデックスが張られていないレンジのクエリの場合，大きなデータを読み込む必要があることである．さらに，インデックスの更新に伴う入出力コストも無視出来ない．インデックスの更新に関して，ほんの僅かな更新がある場合でも，HDFS の制約から，インデックスファイルすべてを読み込み，インデックスすべてを書き込まなければならない．これらの課題を解決する，2 つの低減手法を提案する．1 つ目は，値付きレコードポインタである．ポインタに付加されたレンジの値を参照することで，実際のクエリに必要なデータのみをアクセスが可能になる．2 つ目は，値付きレコードポインタに加え，インデックス作成時にレンジ分割ではなく，ソートをかける手法である．(値付きの）レコードポインタ配列をソートすることで，レコードポインタ配列の更新が不要になる．本稿では，これ以降，これまで提案したインデックス構築手法を「ポインタ手法」と呼ぶ．

a）　値付きポインタ

ポインタ手法では，アダプティブインデックスの特性から，インデックスが張られていないレンジのクエリの場合，大きなデータを読み込む必要になってしまう．インデックスエントリに値を付加させることで，データアクセスを行う前にインデックスの更新を行えるようになる．その結果，クエリ条件を満たすレコードのみを取得可能になり，データの入出力コスト（ランダムアクセスアクセス回数）を大きく削減できる．インデックスの生成フローに関しては，レコードポインタ作成時に値を付加させる点を除き，変わらない．条件値変更時のインデックスの更新フローについては，データアクセスを行う前にインデックスの更新を行い，Map ステップではレコードポインタの作成は行わない．一方，値付きポインタ手法ではインデックスのサイズが増加するため，インデックスの入出力コストが増加してしまう．また，全体のデータサイズ増加を考慮し，最新のカラムのみに値を持つようにした．

b）　インデックス作成時のソート

インデックスの更新に伴う入出コストを下げるために，値付きレコードポインタに加え，インデックス作成時にレコードポインタ配列をレンジ分割ではなく，ソーティングを行う手法を提案する．ソーティングを行った場合，レコードポインタ配列が更新不要の収束状態になっているので，以後のクエリでレコードポインタ配列を更新する必要はない．インデックスツリーとレコードポインタ配列は別ファイルで管理しておき，条件値変更時にインデックスの更新ではインデックスツリーファイルのみを更新することで，インデックスの入出力コストを大きく削減することができる．条件値変更時にインデックスの更新にお

いて，インデックスの読込では，インデックツリーと必要なレンジのみを読み込めばよい．また，インデックス書き込みの際には，レコードポインタ配列を書き込む必要がない．このように，レコードポインタ配列に伴う入出力が削減される．一方で，インデックス作成時にレンジ分割より負荷が高いソートを行うため，インデックス作成に伴うオーバヘッドは増加する．

4. シミュレーション環境

MapReduce 環境に適応的インデックス全てを埋め込む実装は難しいため，本稿では入出力コストに着目してシミュレーションを行い評価を行った．この説では，本稿で構築したシミュレーション環境を述べる．我々は, [8] らが提唱した Hadoop におけるジョブ実行時間に関するコストモデルを基に，シミュレーション環境を構築した. [8] のコストモデルでは， ジョブ実行時間をタスク実行時間までブレークダウンし，これらをクラスタ環境が提供する同時タスク実行回数を考慮して累積することで算出する．タスク実行時間については，タスクにおけるデータフローに沿って，ステップに要する実行コスト（I/O 時間と CPU 時間の単純和）を累積することで算出する．Map タスクの実行時間では，図 1 に示される *Scan* ステップ,*Map* ステップ, *Collect* ステップ,*Spill* ステップ,*Merge* ステップにおける 5 つの実行コストの和を求める．我々は，Map タスクの実行時間算出において，インデキシングに関するステップの実行コストを加えた．さらに，データの読み込みを行う *Scan* ステップについては，データアクセス方式を意識した実行コストの算出を行うように変更を加えた．各ステップにおける I/O，CPU コストは，実環境で計測を行い求めた．計測したサーバーのスペックは，Intel Xeon E5530 2.40GHz 2P/8C，DDR3 24GB, HDD × 1 である．使用した HDD の連続読込性能，連続書込性能は，共に 115MBps であった．OS には Linux2.6.32 を, Hadoop はバージョン 0.20.3 を用いた．シミュレーションでは，10GBps のネットワーク環境で接続された 10 台の前述のサーバーを計算ノードとして利用し，ノードあたり 4 タスクが同時実行可能とした．また，1 タスクを 1 コアで実行することを想定している．単純な設定において，構築したコストモデルで算出される値が実際のジョブ実行時間とほぼ同じ挙動に従うことを確認した．

5. 評　　価

第 4. で説明したシミュレーション環境で，提案フレームワークを評価した．データ処理には，大規模データ解析処理としてデータベース演算を採用した．シミュレーションは大きく 2 つに分かれる．1 つめのシミュレーションでは，インデックスを利用しないデータ処理と比較し，適応的インデックス（ポインタ手法）の利用効果を検証する．インデックスを使わないデータ処理とは，従来の MapRedcue の処理や，インデックスを構築する技術を適用した際にインデックスが張られていないカラムに問い合わせを行った時のデータ処理を表している．2 つ目のシミュレーションでは，インデックス運用コスト低減手法の比較を行う．

5.1 評 価 環 境

まず，2 つのシミュレーションに共通する評価環境を述べる．データセットには，17 個のカラム，平均レコード長 170 バイトのリレーション R（1TB）と，10 個のカラム，平均レコード長 168 バイトのリレーション S（100GB）の人工的なデータセットを用いた．リレーション R は *id* カラムによって，S は *partID* カラムによってソートされているまた全てのカラムの値は一様分布に従っている．集中的とランダムの 2 つのワークロードにおいて，第 2 章で示した射影演算，集合演算，結合演算を行う問い合わせを，ワークロードのポリシーに応じてレンジ条件節を変化させながら発行した．集中的なワークロードでは，クエリレンジを決定する変数 x, y は，初期の選択率が 0.01%と固定し，レンジが直前のレンジに含まれるように 80%ずつ縮小していくように変化させた．このワークロードは，本研究が対象とするアドホック解析における一連の問い合わせを模擬しており，提案手法が得意とするワークロードである．ランダムなワークロードでは，クエリレンジを決定する変数 x, y は，選択率を 0.01%に固定し，ランダムな位置にクエリを発行するように変化させた．ランダムなワークロードは，これまでのクエリで作成したインデックスを上手く活用できないため，提案手法にとって苦手なワークロードである．提案手法では，レンジ条件に指定されたカラム *discount* にインデックスを構築していく．MapReduce の主な設定としては，Map タスクに与えられるデータスプリットサイズは 512MB, Reduce タスク数は 1 とした．

5.2 インデックスを利用しない処理との比較

1 つめのシミュレーションでは，提案するフレームワークをインデックスを利用しないデータ処理と比較する．提案手法には，ポインタ手法を用いる．まず，射影演算処理，集合演算処理，結合演算処理の基本的なデータ処理における比較を行い，その後，データ更新時におけるインデックス管理手法を検証する．このシミュレーションでは，本研究が対象とする集中的なワークロードを用いる．

5.2.1 射影演算処理

第 2 章の (1) 射影演算を問い合わせとして発行した．図 6 が，その結果を表している．縦軸が発行したクエリ数を表し，横軸がこれまで発行したクエリの累積ジョブ実行時間を示している．図 6 を見ると，インデックスを利用しない場合の累積実行時間 (*no-index*) が，クエリ数に応じて線形に増加している．初回のクエリにかかる時間が 691 秒，20 個のクエリを発行する時間が 13,800 秒であった．これは，問い合わせ毎に全データスキャンを行うためである．一方，提案手法 (*adaptiveindex*) はクエリ数に応じて微増する．1 回目のクエリに要する時間が 725 秒で，20 回目は 2382 秒であった．1 回目のクエリは全データスキャンによる処理とインデックスの作成で多くの実行時間を要するものの，2 回目以降ではインデックスの利用によるデータ入力の削減効果がインデックス更新のオーバヘッドを勝り，実行時間を大幅に減少させた．以上の結果から, 提案手法は，1 回目のクエリ実行では 5%のインデックス作成オーバヘッドを要するものの，2 回目以降ではインデックスを利用しない処理の累積

実行時間を上回り，最終的には約 6 倍の性能向上を達成した．

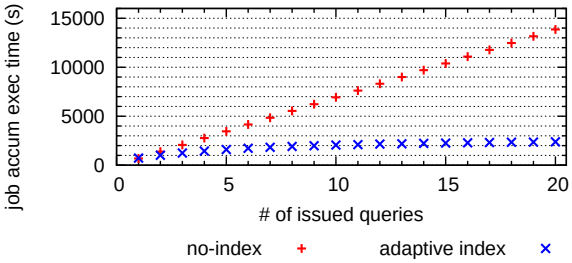

図 6　射影演算処理に要するジョブ累積実行時間の推移

5.2.2　集合演算処理

次に，集合演算処理の結果を述べる．集合演算処理のシミュレーションでは，第 2 章の (2) 集合演算を問い合わせとして発行した．集合演算におけるレコードの集約率は 10%とした．図 7 は，発行したクエリ数と集合演算処理に要するジョブ累積実行時間の推移を表したグラフである．インデックスを用いない処理 (*no-index*) は，クエリ数に比例して累積実行時間が増加し，初回のクエリに 692 秒，20 回目のクエリで 13,822 秒であった．これは，クエリ発行の度に全データスキャンを行うためである．一方，提案手法では累積実行時間が微増で収まり，初回のクエリで 718 秒，20 回目のクエリでは 2,240 秒であった．以上の結果から，集合演算処理に関して，1 回目のクエリではインデックス作成オーバヘッドにより実行時間が 5%増加したものの，2 回目の問い合わせで提案手法が上回り，最終的には 20 回目のクエリ発行時点で 6.1 倍の性能向上を確認した．

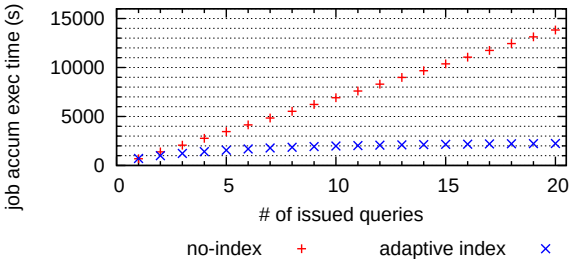

図 7　集合演算処理に要するジョブ累積実行時間の推移

5.2.3　結合演算処理

結合演算処理のシミュレーションでは，第 2 章の (3) 結合演算を問い合わせとして発行した．(3) 結合演算の問い合わせでは，リレーション R に関してインデックスを用いた選択処理を行い，他方のリレーション S では従来どおりデータをスキャンして処理する．図 8 は，集中的なワークロードにおいて，発行したクエリ数と結合演算処理に要するジョブ累積実行時間の推移を表したグラフである．インデックスを利用しない処理 (*no-index*) は，クエリ数に比例して累積実行コストが増加し，1 つのクエリ，20 個のクエリを処理する時間がそれぞれ 774 秒 15,478 秒であった．提案手法では，1 つのクエリに 801 秒かかり，その後累積実行コストはゆるやかに増加し，20 個のクエリを実行するのに 3,895 秒を要した．結果をまとめると，初回のインデックス作成オーバヘッドが 3.5%，20 回のクエリを実行する時間について 4 倍の性能向上を達成した．この結果から，適応的インデキシングにより，結合演算に関して累積実行コストを大幅に削減できることがわかった．

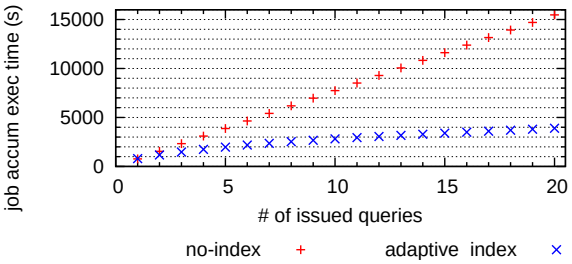

図 8　結合演算処理に要するジョブ累積実行時間の推移

5.2.4　データ更新時におけるインデックス管理

最後に，データ更新時におけるインデックス管理手法の評価を述べる．評価項目は，クエリ実行時間とした．データ更新時における評価環境を述べる．データセットには前述のリレーション R（1TB）を利用し，射影演算問い合わせを行う．シミュレーションでは，5 回のクエリ発行毎にデータ更新を生じさせる．データの更新内容は，データスプリットの更新比率に関するパラメタ 2 つと，更新されたデータスプリット内の更新レコード比率に関するパラメタ 2 つで表現する．データスプリットの更新比率パラメタのデータスプリット追加率は，全体のデータスプリット数に対する新規に追加されたデータスプリット数の比率を表し，同様に，データスプリット更新率は（更新前の）全体のデータスプリット数に対する更新（削除も含む）があった既存データスプリットの比率を表す．更新データスプリット内の更新レコード比率に関するパラメタに関しては，データスプリット内の全レコード数に対する更新されたレコード数の比率と，削除された削除されたレコード数の比率を用意した．更新するデータスプリットの選択方法については，ランダムに選択する方法と最新のデータスプリットから順に（データスプリット ID が大きい順に）選択する手法を検討したが，結果にほとんど差が見られなかったため，本稿ではランダムに選択する結果を紹介する．

図 9 がその結果である．更新に関するパラメタ設定に関しては，データスプリット更新率，追加率を 80%，更新，削除レコード比率を 0.1%とした．図 9 を見ると，インデックスを利用しない処理 (*no-index*) に関しては，発行されたクエリ数にほぼ比例して累積実行時間を増加させ，5 回のデータ更新毎に，増加率を上昇させていく．これは，各クエリごとに全データスキャンを行うためである．それ以外のインデックス管理手法では，前述した 2 つのデータ更新パターン同様に，5 回のデータ更新ごとに実行時間を急増させ，それ以外はインデックスの効果で微増する．データ更新の際，インデックスの部分更新を行う手法 (*partial-update*) とデータスプリットごとのインデックス破棄 (*split-delete*) では，既存のデータスプリットと新規のデータスプリットに対してインデックスを作成するオーバヘッドが

生じる．インデックスの部分更新を行う手法 (*partial-update*) では，新規のデータスプリットに対してインデックスを作成するだけのオーバヘッドが生じる．クエリ実行時にインデックスを更新するコストが加わるものの，クエリ実行時にも微増にとどまった．

これらの結果から，データ更新が発生する場合に，提案手法を用いてインデックスを管理することで，インデックスを用いない処理に比べて，大きく処理コストを削減できることが分かった．また，今回の設定において，インデックスの部分更新を行う手法 (*partial-update*) が最も実行コストを抑えるインデックスを管理できるとわかった．

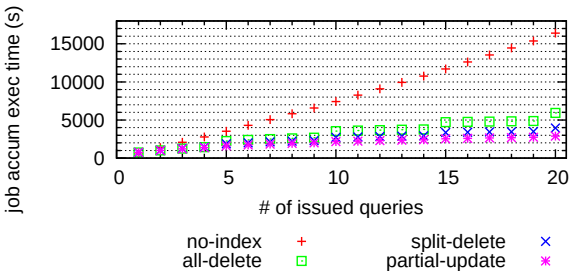

図 9　データ更新時におけるクエリ累積実行時間の推移

5.3　インデックス運用コスト低減手法の比較

2 つ目のシミュレーションでは，インデックス運用コスト低減手法の比較を行う．ここでは，提案するポインタ手法と，値付きレコードポインタ手法，インデックス作成時のソートを手法のコスト低減手法の 2 つと，インデックス作成時にフルインデックスを作成する手法の 4 つを比較する．インデックス作成時にフルインデックスを作成する手法は，関連研究 [9] のカラムに対して適応的にフルインデックスを作る手法に相当するものを表している．この 4 つの手法を，集中的なワークロード，ランダムワークロード，データ更新が発生する場合の 3 つのケースにおいて検証した．

5.4　集中的なワークロード

集中的なワークロードの結果（図 10）を見ると，すべての手法において，累積コストがゆるやかに増加している．個々の結果を比較すると，まず 1 回目のクエリでは，累積コストが低い順に，ポインタ手法，値付きレコードポインタ手法，インデックス作成時のソート手法，インデックス作成時のフルインデックス手法となっている．これは，インデックス作成における作業コストの少なさの順番と一致している．2 ～ 4 回目のクエリにおいては，インデックス作成オーバヘッドのリードによりポインタ手法と値付きレコードポインタ手法が優れているが，5 回目以降はインデックス作成時のソート手法やインデックス作成時のフルインデックス手法が最も良い結果を示す．これは，クエリ実行時のインデックスの入出力コストがその他手法よりも低く，クエリ数が増加するに連れ性能差が拡大したためである．最終的には，20 回目のクエリ発行時点で，ベストケースであるインデックス作成時のソートがポインタ手法に比べて約 1.2 倍の性能向上を達成した．

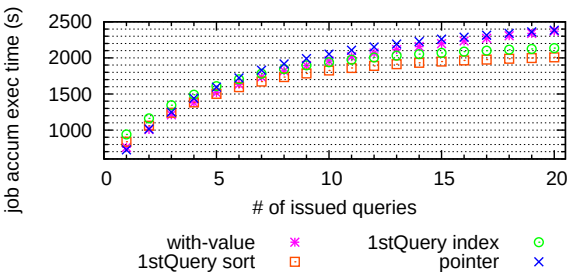

図 10　集中的なワークロードにおけるインデックス運用コストの低減手法の比較

5.5　ランダムワークロード

図 11 は，ランダムなワークロードの結果を表している．このワークーロードは我々が想定しないアクセスパターンに従うが，低減手法によりデータのアクセスコストを下げることができる．ポインタ手法の場合に関して，2 回目以降に性能が大きく悪化し，従来の MapReduce 処理にかかる以上の実行時間を必要とする．これは，インデックスが貼られていないレンジにアクセスが頻繁に行くからである．そのため，ポインタ手法に関しては，2 回目以降はインデックスを利用しない処理を行うこととした．結果を見ると，ポインタ手法を除く手法では，累積コストが線形に増加している．これは，ポインタ手法を除く手法では値付きのレコードポインタを保持するため，各クエリで実際にクエリするレコード数だけデータアクセスを行うためである（ランダムワークロードでは選択率が固定なので，クエリするレコード数はクエリによらず一定）．これらの結果から，ランダムワークロードに対しても，レコードポインタに値を負荷することで効率的に応答できることを確認した．また，ポインタ手法を除く手法の性能差はほとんど見られなかった．

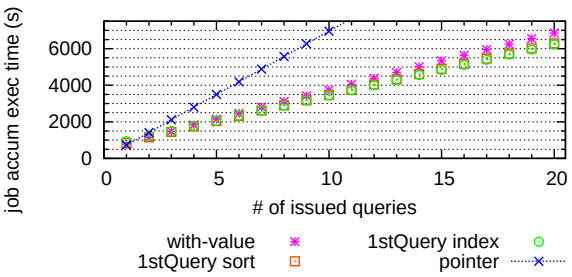

図 11　ランダムワークロードにおけるインデックス運用コストの低減手法の比較

5.6　データ更新時

図 12 は，データ更新時における各手法の累積実行コストの推移を表している．更新設定は前述した通りで，インデックスの更新手法には該当データスプリットのみ破棄を行う手法を用いた．また，ランダムなワークロードを用いた．結果をみると，すべての手法に関して，データ更新を行う 5 回ごとに実行時間が急増している．各手法を比較すると，インデックス作成時のフルインデックス手法が常に最大であるとわかる．これは，インデックス作成のオーバーヘッドが大きいためである．コストが最小となるのは，2~4 回目まではポインタと値付きポインタ

手法，5 回目以降はインデックス作成時のソート手法であった．2〜4 回目まではポインタと値付きポインタ手法が最小となるのは，データ更新がない場合と同様である．5 回目以降でインデックス作成時のソート手法が最小となるのは，インデックスの作成オーバヘッドが相対的に大きくなく，クエリ数が増加した時の実行時間が小さいためである．今回の更新設定では，クエリ数が増加した時の実行時間の削減が相対的に大きく，最小となった．

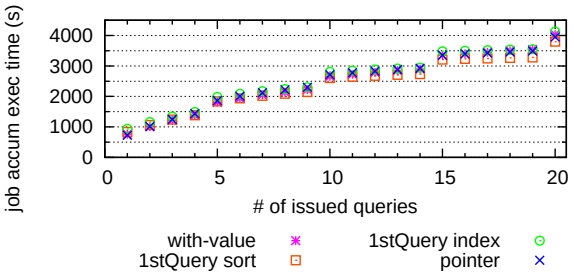

図 12　データ更新時におけるインデックス運用コストの低減手法の比較

6. 関 連 研 究

MapReduce 処理系の高速化手法に関して，インデックス機構を導入し，繰り返しのデータ入力コストを削減する研究が多くなされている [9]〜[12]．Dittrich らは，データロード時に，データスプリットごとにクラスターインデックスを構築し，クエリ実行時にインデックスを利用したアクセスを行う [11]．この手法では，クラスタリングされたカラムを選択条件にもつ問い合わせに関して，スキャン IO 量を大きく削減することができる．一方で，クラスタリングされたいないかカラムに対しては従来同様に全データスキャンを行うため，我々が想定するようにトライアンドエラーでどのカラムにユーザーが興味をもつかが予測できない場合には不向きである．また，我々の研究と並行される形で進められた，昨年 12 月に公開された Ritcher らの技術論文 [9] では，クエリ実行時にクラスターインデックスを構築する手法を提案している．ただし，インデックスの構築手法はフルインデックスからの改良で，データベース分野で研究されてきた Adaptive index [6], [7] の特長は活かされていない．クエリされた “カラム” に対して適応的にフルインデックスを構築するのみで，提案手法のようにクエリに適応的にインデックスを構築するわけではない．

大規模データにおけるアドホック解析の処理基盤として，*Dremel* [13] が提案されている．*Dremel* では，複雑なスキーマを持つデータをカラムごとに格納することでデータ読み込み時間を削減し，ツリー型でノード間通信を行うことで効率的にレコードを集約しネットワーク待機，処理を高速化する．我々は，提案手法が *Dremel* においても適用可能であると考えている．

7. ま と め

本稿では，MapRedcue 環境におけるアドホック解析に関して，同じカラムに似たようなレンジ条件を持つクエリが発行された時に，処理を高速化する手法を述べた．提案フレームワークでは，MapReduce 環境に適応的インデックスを導入し，不要な I/O コストを削減するとともに，アドホックな処理に対して柔軟に対応する．Map タスク実行中に担当のデータスプリットに対してインデックスを作成し，レンジ条件が変更される度にクエリに応じてインデックスを更新していく．インデックスの更新では，クエリされたレンジに対してインデックスを貼るため，よくアクセスされるレンジほど，応答性能が高まっていく．また，データ更新の際には，データスプリット毎にインデックスを管理を行う．シミュレーションによる結果，基本的なデータ処理に関して，インデックスを利用しない処理と比較し提案フレームワークの有効性を確認した．

文　献

[1] M. Zuckerberg, “One billion people on facebook,” Oct. 2012. One Billion People on Facebook

[2] “Big data: The next frontier for innovation, competition, and productivity,” 2011.

[3] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, pp.10–10, OSDI’04, USENIX Association, 2004.

[4] “hadoop http://hadoop.apache.org/”.

[5] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at facebook,” Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp.1013–1020, SIGMOD ’10, ACM, 2010.

[6] S. Idreos, M.L. Kersten, and S. Manegold, “Database cracking,” CIDR, pp.68–78, 2007.

[7] S. Idreos, M.L. Kersten, and S. Manegold, “Self-organizing tuple reconstruction in column-stores,” Proceedings of the 35th SIGMOD international conference on Management of data, pp.297–308, SIGMOD ’09, ACM, New York, NY, USA, 2009. http://doi.acm.org/10.1145/1559845.1559878

[8] H. Herodotou, “Hadoop performance models,” Dec. 2011.

[9] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich, “Towards zero-overhead adaptive indexing in hadoop,” 2012.

[10] D. Jiang, B.C. Ooi, L. Shi, and S. Wu, “The performance of mapreduce: an in-depth study,” Proc. VLDB Endow., vol.3, no.1-2, pp.472–483, Sept. 2010.

[11] J. Dittrich, J.-A. Quiané-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 Endow., vol.3, no.1-2, pp.515–529, Sept. 2010. http://dl.acm.org/citation.cfm?id=1920841.1920908

[12] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, “Only aggressive elephants are fast elephants,” Proc. VLDB Endow., vol.5, no.11, pp.1591–1602, July 2012. http://dl.acm.org/citation.cfm?id=2350229.2350272

[13] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” Proc. of the 36th Int’l Conf on Very Large Data Bases, pp.330–339, 2010.