社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

MapReduce環境におけるアドホックなクエリを対象とした，Adaptive indexing適用に関する一検討

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　MapReduce をベースとするデータ処理基盤は，定型的かつ定期的な問い合わせとともに，アドホックな問い合わせを行う解析基盤として重要性を増している．MapReduce 環境では， Map 処理時にすべてのレコードをスキャンし，処理を行う．例えば，同じようなレコード選択条件を持つクエリが繰り返し処理された場合でも，毎回，アドホックなクエリとして扱われ，レコードの全スキャンが繰り返される．本稿では，アドホックではあるが同じよう選択条件をもつ問合せ処理を MapReduce 環境において効率良く処理するために，データベースの技術である Adaptive indexing の導入を検討する．Adaptive indexing とはクエリの実行時にインデックスを生成，更新していく手法であり，異なるレンジ検索が繰り返し発行されるごとに，インデックスを更新し，次のクエリにてそのインデクスを利用することで，クエリの処理時間が速くなる．Adaptive indexing 適用に関する MapReduce 環境における課題を検討し，適用モデルを提案する．

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

Consideration 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　MapReduce-based processing infrastructure is getting more important as a data analysis system,which executes ad-hoc queries, in addition to regular batch queries. In MapReduce environment, each map task scans all records and processes them. This all-scan access is repeated if similar queries is repeatedly issued ——for example the queries which have the range condition on the same column. In this paper, in order to process efficiently *ad-hoc but with-similar-range-condition* query in Map Reduce environment, we will consider the introduction of adaptive indexing techniques into Map Reduce environment. Adaptive indexing is a DBMS technique ——index creation and update during the query execution. Every query, index is updated so that subsequent queries will take shorter execution time by using this improved index. We will discuss the challenge around the adaptation of the adaptive indexing into MapReduce environment and propose some model.

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．2012 年 9 月時点で毎月 10 億人が Facebook を利用し、ユーザーによって作成され共有されるコンテンツ数は月間 300 億に到達すると報告されている [1], [2].

こうした大規模データの解析は，多くの企業，研究機関で行われている．Google, Yahoo!, Facebook などは，ペタバイト級のデータを処理するデータウェアハウスを保持し，日々解析を行ない，ユーザーにサービスを提供している [3]〜[6].

MapReduce フレームワーク [5] とそのオープンソース実装である Hadoop [7] は，大規模データの解析基盤として現在主

流なプラットフォームである．MapReduce フレームワークでは，定期的なバッチ処理に加えてアドホックな問い合わせが実行されている [3]．

MapReduce アプリケーションは、 Map タスクと Reduce タスクで構成される．Map タスクは，分割された入力データ（データスプリット）からレコードを全スキャンし，処理を行う．一般的に，ユーザーがアドホックに解析を行っていく場合，一度興味深い切り口を発見したら，データを同じ切り口で繰り返し処理していく傾向にある．しかし、MapReduce 環境では，例えば同じようなレコード選択条件を持つクエリが繰り返し処理された場合でも，毎回アドホックなクエリとして扱われ，レコードの全スキャンが繰り返される．

本稿では，アドホックではあるが同じよう選択条件をもつ問合せ処理を MapReduce 環境において効率良く処理するために，データベースの技術である Adaptive indexing の導入を検討する．Adaptive indexing とはクエリの実行時にインデックスを生成，更新していく手法であり，異なるレンジ検索が繰り返し発行されるごとに、インデックスを更新し、次のクエリにてそのインデックスを利用することで，クエリの処理時間が速くなる．本稿ではデータ処理をレンジ条件をもつプロジェクションクエリに限定し，Adaptive indexing の導入を議論していく．

2. 研究背景

この章では，MapReduce フレームワークで代表的な実装である Hadoop の MapReduce と，動的なワークロードに適応的に応じる、データベース分野の技術 Adaptive indexing を説明する．

2.1 MapReduce

MapReduce アプリケーションは，Map フェーズと Reduce フェーズで処理が行われるが、本稿のターゲットであるプロジェクションクエリは Map フェーズのみで実行可能である．図 1 は，Map タスクでカラム a に対するレンジ条件 $2 \leqq a < 10$ を満たすレコードからカラム a と b を抜き出すプロジェクションクエリが実行される様子を示している．MapReduce アプリケーションの入力データは，通常 Hadoop 独自の分散ファイルシステムである hdfs [8] に格納されていて，アプリケーションの起動直後にタスクスケジューラによってデータスプリットと呼ばれる論理的な単位に分割され， Map タスクに割り当てられる．Map タスクでは，担当のデータスプリットを保持するノードに対して，あるバッファサイズ単位で I/O 要求をかけ，データを読み込む (1.scan)．次に，Map タスクは，バッファからキーバリューペアで表現されるレコードをパースし，map 関数を実行する (2.processing)．図 1 では，レコードのファイルオフセットをキーに、データの一行すべてをバリューにしたレコードを作成し，map 関数でプロジェクション操作とレンジ条件判定を実行している．レコードの読み込みから map 関数の実行までの一連の操作を、データスプリットのレコードがなくなるまで繰り返す．最後にレンジ条件を満たすレコードは，カラム a と b の値が専用のバッファに格納され，hdfs 上のファイルに出力される（3.write）．Mapreduce 処理では，毎回，データの全スキャンが行われる．

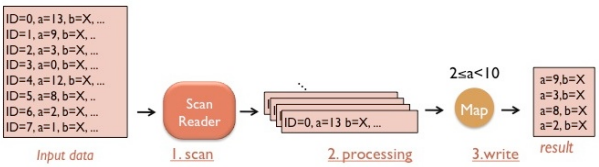


図 1　MapReduce 環境におけるレンジ条件をもつプロジェクションクエリの処理フロー

2.2 Adaptive indexing

Adaptive indexing [9] は，動的なワークロードに適応的に対応するために、クエリ実行時にインデックスの作成と更新を行う技術で，Ideos らによって提案された．Adaptive indexing は，インデックス構築時に最適なインデックスを作るのではなく，インデックスをクエリ実行時に徐々に最適化していく．クエリが発行されると，既存のインデックスの恩恵を受けてクエリ処理を行いつつ，次のクエリでインデックスの恩恵をより受けられるように，インデックスを更新していく．Adaptive indexing 技術の中でも，オンメモリ上でインデックスデータを扱うことを想定した Database cracking [9] を紹介する．

2.2.1 Database Cracking

Database cracking は，レンジ条件を持つクエリに対して適応的にインデックスを作成、更新していく技術である．図 2 は，レンジ条件を持つクエリが 2 回発行された時にカラム a に対するインデックスが生成，更新されていく様子を表している．クエリ発行前には，インデックスは存在しない．1 つめのクエリでは、すべてのレコードがスキャンされ，カラム a に対するレンジ条件判定を行う．レコードを処理する際に，クラッカーカラムとレンジ検索用ツリーからなるインデックスを作成する．まず、各レコードのカラム a の値とレコードアドレスをエントリとしてレコード順に並べた配列クラッカーカラムを作成する (図 2 上段:hbneci···fsi)，次に，クエリレンジ条件の始点 d と終点 i をピボットとしてレンジ分割し，クラッカーカラムのエントリの順番を入れ替える．その結果，クラッカーカラムは，レンジ (1)d を下回るカラム値を持つエントリの集合，レンジ (2)d 以上 i を下回るカラム値を持つエントリの集合，レンジ (3)i 以上のカラム値を持つエントリの集合，の 3 つレンジに分割される (図 2 中段)．最後に，レンジ 2. を利用してクエリ結果

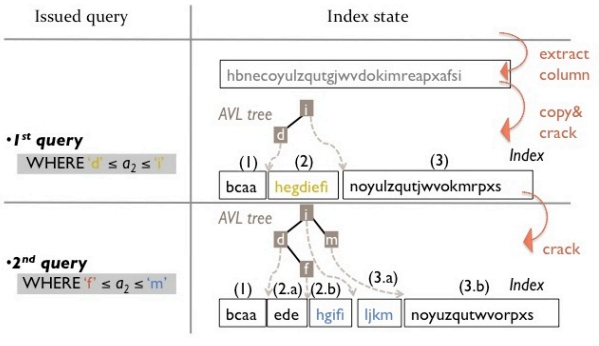


図 2　Database cracking によるインデックスの作成、更新例

を返却すると同時に，次のクエリでレンジに直接アクセスが行えるように，レンジ検索用ツリー（AVL ツリー）にレンジを登録しておく．具体的には，d をキーにレンジ 2. のアドレスを値としたノード，i をキーにレンジ 3. のアドレスを値に持つノード 2 つを挿入する．次のクエリでは，作成したインデックスを活用することで，クエリ処理時間を削減できる．クラッカーカラムは，クエリレンジに応じて，クエリ処理に必要なレンジのみを読み込み更新していく．レンジ条件 $f \leqq a_2 < m$ をもつ 2 つ目のクエリが発行されたとき，レンジ検索用ツリーをからクエリ処理に必要なレンジを選択し，レンジ (2) とレンジ (3) が選択される，最初のクエリと同様に，レンジ条件の端点である f と m をピボットにして，それぞれのレンジを 2 分割する．クエリ処理は，レンジ (2) とレンジ (3) が分割されて新しくできたレンジ (2.b) と (3.a) を利用して行う．最後に，新しくできたレンジを直接アクセスできるようにレンジ条件の端点 f と m をキーとしたノードがレンジ検索用ツリーに挿入される．レンジ条件節の統合，不等号により，カラム値がレンジに含まれるかといった情報はレンジ検索用の AVL ツリーのノードに付加される．

3. 提 案 手 法

同じカラムに対するレンジ条件をもつクエリが繰り返し処理された場合に，毎回，データスプリットの全スキャンが繰り返されるのは効率が悪い．提案手法では，Database Cracking 技術を適用し，インデックスを Map タスク実行中に作成，更新していく．Map タスクでは，データスプリットを論理的なレコード集合であるレコードユニットに水平分割し，レコードユニットごとにデータの処理とインデックスの作成，更新が行う．レコードユニット単位で処理を行う理由は，レンジ条件を満たすレコード取得の際に，一度データをすべてシーケンシャルにアクセスする必要が生じた場合に一時的な作業用メモリが大量に必要となることを防止するためである．詳細は，3.3 章で説明する．この説以降，提案手法の説明のために，レンジ条件を持つプロジェクションクエリの例として $Query1$ を用いる．

$$Query1: \ SELECT \ \ id, discount \, FROM \, linetable$$
$$WHERE \ \ x \leqq discount < y$$

インデックスは，クラッカーカラム，レンジ検索構造体，レン

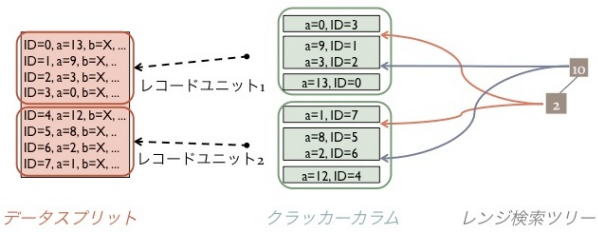


図 3　インデックス構造の例: $2 \leqq a < 10$ のレンジ条件をもつクエリが発行された時のインデックスの状態

ジテーブルの 3 つの要素で構成される．クラッカーカラムは，レコードユニットごとに作成され，クエリに応じてレンジ分割されていく．レンジ検索構造体は，レンジ分割されたレンジに直接アクセスするための検索構造体であり，クエリレンジの端点キーとしたノードを格納する．ノードの値には，キーに対応するすべてのクラッカーカラムのレンジアドレスを保持する．レンジテーブルは，各レコードユニットにアクセスするための表であり，テーブルのエントリはレコードユニット ID，レコードユニットの開始アドレス，終了アドレスの 3 つからなる．また，インデックスとデータを関係付けるには，ファイル名を用いる．

3.1 インデックスの作成

インデックスの作成は、クエリがはじめて発行された時に行う (図 4)．従来の MapReduce 処理と異なる点は，Map 関数実行時にインデックスエントリを作成することである．map 関数でプロジェクション処理を行い結果を出力すると同時に，カラム値とレコードアドレス（ファイルオフセット）からなるインデックスエントリを作成し，クエリレンジに応じた 3 つの専用バッファに格納する（図 4 の 3. index create）．バッファ (1)$a_2 \leqq 3$ を満たすエントリの集合，バッファ (2)$3 \leqq a_2 < 8$ を満たすエントリの集合，バッファ (3)$8 < a_2$ を満たすエントリの集合，の 3 つに振り分けれている．同じレコードユニットに属するすべてのレコードが map 関数で実行された後に，バッファ (1)，(2)，(3) の順番で結合し，クラッカーカラムを作成する．この時，レンジアドレスに相当する，結合されたバッファの先頭アドレスを保持しておく．すべてのレコードユニットでクラッカーカラムが作成された後，レンジ検索構造体を作成し、レンジ条件の端点である 3 と 8 をキーに、キーに対応するすべてのクラッカカラムのアドレスリストをバリューにもつノードが挿入される．この段階で，Database Cracking によるインデックスが作成された．また，次のクエリでレコードユニットごとにデータ処理を行うために、データスプリットにおける各レコードユニットの範囲を保持する表レンジテーブルを作成する．最後に，クエリ処理の出力結果とともにインデックスをストレージに書き込む．

3.2 インデックスの更新

インデックスの更新は，インデックスを作成したカラムに対するレンジ条件をもつクエリが再び発行された時に起きる（図 5）．データ処理とインデックスの更新はレコードユニット単位で行なわれ、クエリ処理は以前のインデックスを活用して効率的に行われる．Map タスクは，データスプリットに対応するレンジ検索構造体を読み込み，処理に必要なレンジを決定し，レコードユニットごとにレンジを読み込む (1.*read index*)．次に，クラッカーカラムを更新し (2.*index update*) し，更新されたレンジを利用してデータスプリットからのデータ (3.*read data*) を取得する．この時，インデックスの活用によって，レンジ条件を満たすレコードだけが map 関数で処理することが可能になる (4.*processing*)．これをレコードユニット数だけ繰り返す．最後に，出力結果と更新されたインデックスをストレージに書き込み、インデックスの書き込み完了後に古いインデックスを削除する．

3.3 該当レコード数に応じたレコードアクセスポリシー

インデックスを利用して，データスプリットから必要なレコードを取得するときに，できるだけコストを低く抑えたい．hdfs 上のファイルのランダムアクセスコストが高いため、該当

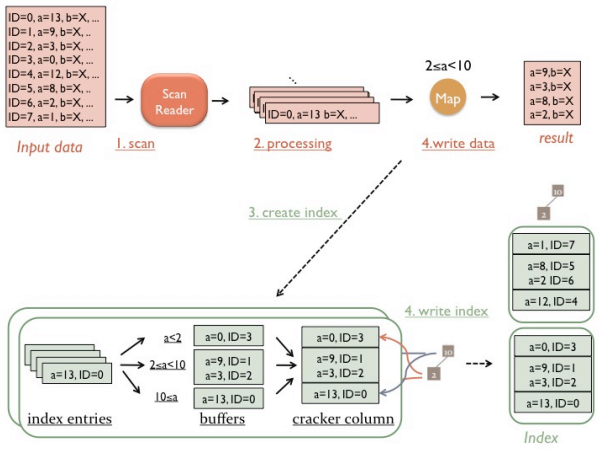


図 4　インデックスの作成例: $2 \leqq a < 10$ のレンジ条件をもつクエリが発行．太線の囲いがレコードユニットによるデータ、処理の分割を示す．インデックス生成は次の 3 ステップで行われる．1. すべてのレコードに対してインデックスのエントリを作成し，レンジ条件に応じて 3 つのバッファに分配，2. 同じレコードユニットのすべてのレコードを処理後，バッファを連結させクラッカーカラムを作成，3. すべてのレコードユニットを消費後，レンジ検索ツリーを作成

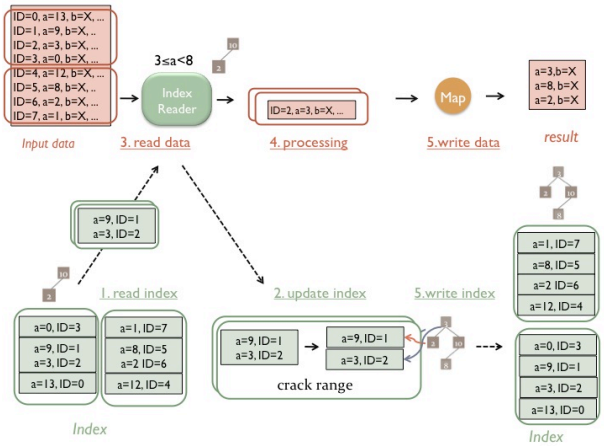


図 5　インデックス更新例．$3 \leqq a < 8$ のレンジ条件をもつクエリが発行．太線の囲いがレコードユニットによるデータ、処理の分割を示す．データ処理とインデックスの更新がレコードユニット単位で行われる．

するレコード数が多い場合非常にコストが高くなってしまう．そのため，条件句を満足するレコード数に応じてレコードアクセス方法を選択する．具体的なレコード数は，次の章で紹介するコスト計算とマシン環境によって決定する．

1. 条件に該当するレコード数が存在しない場合　データスプリットにアクセスしない．
2. 条件に該当するレコード数が少ない場合　ストレージからランダムアクセスを行い必要なレコードを取得する．
3. その他　データをシーケンシャルアクセスして，ローカルでランダムアクセスを行う

3.4　データの追加

データの解析を行なっている間に，新しいデータは定期的に追加されていく．提案手法では，個々の Map タスクが独立して，割り当てられたデータスプリットに対して データ処理を行うため新たなデータの追加に対しても，効率良く処理を行える．つまり，古いデータスプリットが割り当てられた Map タスクはインデックスの恩恵を受けデータ処理を行い，新しいデータスプリットを割り当てられた Map タスクはインデックスの生成を行いつつ，データ処理を行う．

表 1　コストモデルで利用する変数の一覧

変数	意味
JobCost	MapReduce ジョブ実行コスト
MapTaskCost	Map タスク合計コスト
ReadCost	読み込みコスト
MapCost	map 処理コスト
WriteCost	書き込みコスト
DataReadCost	データ読み込みコスト
IndexReadCost	インデックスデータの読み込みコスト
IndexUpdateCost	インデックス作成/更新理コスト
DataWriteCost	データ書き込みコスト
IndexWriteCost	インデックスデータの書き込みコスト
MapTaskCnt	Map タスク数
MapWaveCnt	各 Map スロットにおける平均タスク実行回数
csMapInRecs	map 関数に渡す平均レコード数
csAveRecSize	平均レコード長
csMapOutputSize	平均出力データサイズ
csMapOutputRecs	平均出力レコード数
csRecordUnitCnt	レコードユニット数
csHdfsSeqRead	Hdfs のシーケンシャルリードスループット
csHdfsRandReadCost	Hdfs のランダムリードコスト
csLoaclRandReadCost	ローカルでのランダムリードコスト
csHdfsSeqWrite	Hdfs のシーケンシャルライトスループット
csMapCost	単位レコードあたりの map 関数実行コスト
pTaskSlotCnt	同時に Map タスクを実行可能出来る数
pDataSize	データスプリットのサイズ

4.　コストモデル

提案手法の効果を図るために, [10] らの技術報告を参考に実行時間に関するコストモデルを構築した．コストモデルで使用する変数は，MapReduce ジョブにおけるコストと統計値を表す変数と，データ，アプリケーション，マシン環境と 3 つに依存する cs で始まる変数，ユーザーが指定するパラメタを表す p で始まる変数（と p で始まる変数から算出される値）からなる（表 1）．ジョブ実行コスト $JobCost$ は、map タスクが繰り返し行われる回数 $pWaveCnt$ と Map タスクの実行コスト $MapTaskCost$ を用いて，式 1 と表す．

$$JobCost = pWaveCnt \times MapTaskCost \quad (1)$$

map タスクが繰り返し行われる回数 $pWaveCnt$ 自体は，Map タスク $MapTaskCnt$ と同時に Map タスクを実行可能出来る数 $pTaskSlotCnt$ から，式 2 と表現できる．

$$pWaveCnt = mapTaskCnt \div pSlotNum \quad (2)$$

従来の MapReduce 処理における Map タスクのコスト $MapTaskCost$ は，読み込みコスト，map コスト，書き込みコストの 3 つ和で表現できる（式 3,4,5）．

$$ReadCost = pDataSize \div csHdfsSeqRead \quad (3)$$

$$MapCost = csMapInRecs \times csMapCost \quad (4)$$

$$WriteCost = csMapOutputSize \div csHdfsSeqWrite \quad (5)$$

提案手法におけるコストを，レコードユニットごとに考える．Map フェーズの各ステップのコストに添字 i をつけた変数を，レコードユニット i における各ステップのコストとする．Map タスクにおけるコストは，レコードユニット数である $csRecordUnitCnt$ との積で表現できる．読み込みコスト $ReadCost_i$ は，データの読み込みコスト $DataReadCost_i$ とインデックスデータ $IndexReadCost_i$ の読み込みコストに別れる．データの読み込みコストは，レコードの選択率によってアクセス方式が変更されるため、変化する．選択率が 0 の場合，データの読み込みコスト $DataReadCost_i$ は 0 であり，選択率が 0 以上で，ランダムアクセスでレコードを取得する場合には，式 6 のように，該当レコード数とランダムアクセスコストの積で表せる．

$$DataReadCost_i = csMapOutputRecs_i \times csHdfsRandRead \quad (6)$$

シーケンシャルにデータを取得する場合は、ローカルでランダムアクセスコストを行うコストも加えて，式 7 で表現される．

$$DataReadCost_i = pRecordUnitSize \div csHdfsSeqRead + csMapOutputRecs_i \times csLoaclRandReadCost \quad (7)$$

map 処理コスト $MapCost_i$ は，条件を満たすレコード数と単位レコードあたりコストの積で表現できる（式 8）．

$$MapCost_i = csMapOutputRecs_i \times csMapCost \quad (8)$$

さらに書き込みコスト式 9 のように表すことができ，その他にインデックス更新コスト $indexUpdateCost_i$ がかかる．

$$WriteCost_i = DataWriteCost_i + IndexWriteCost_i \quad (9)$$

5. 評　　価

節 4. で構築したコストモデルを用いて，シミュレーションによる評価を行った．構築したコストモデルは，タスクのスケジューリングコスト、初期化コスト、IO 干渉によるスループットの低下は考慮されておらず，データ処理を行うシステムはホモジニアスな環境を想定し，圧縮，解凍などのデータ操作は含まれていない．また，Map タスクの 4 つの単純和は複数スレッドが並行して動作するために実際のタスク実行時間と完全に一致しない．今後，MapReduce 環境を詳細に模擬するためにモデルの精緻化をすすめる必要があるが，今回のコストモデルにおいても，手法の効果を比較する上で十分有効なモデルである．表 1 上の cs で始まる環境依存の変数の値は，HDFS のマイクロベンチマークや単純な Map ジョブを実行することで算出した．シミュレーションでは，計算ノードが 10 台あり各ノードにおいて 4 スロットが利用可能であるとした．

5.1 セットアップ

データセットは TPC-H を用いたその中でも，スケールファクタを 100 として，LINEITEM テーブル (約 950GB) を使用した．LINEITEM 表は，商品 ID$linenumber$ と割引率 $discount$ を含む 17 個のフィールドで構成される．ある商用 DB で計算した最大レコードサイズが 170Byte であり，この値を平均レコード長として採用する．割引率 $discount$ フィールドの値は，ある限定された範囲における一様分布で与えられる．ワークロードは，50 個のプロジェクションクエリ（$Query1$）である．ランダムに選択された割引率 $discount$ フィールドの値に対して，選択率を固定したレンジクエリを発行する．スプリットデータサイズを 512M バイト，レコードユニットサイズを 16M バイトとした．すべてのスプリットデータ，レコードユニットにおいて、同じデータ分布を持つことに留意する．そのため，今回の実験ではレコードユニットのサイズは性能に影響を与えない．

5.2 クエリ選択率の変化

図 6 は，単一 Map タスクにおける実行コストをステップ別に表したものであり，提案手法の実行コストがクエリの選択率によらず従来の処理よりも低いことがわかる．1 回目のクエリでは，インデックス作成のために従来の処理に比べコストが増加するものの、続くクエリでは従来の処理に比べ大きくコストを削減している．選択率が 0 の場合、データ読み込みコストと map 処理コストが 0 であるため、大幅にコスト削減を達成した（図 6(a)）．この時が最も提案手法の効果が現れるときである．選択率が $1E-5$ の時の場合，レコードに対してランダムアクセスが行われる（図 6(b)），この場合，読み込みデータ量と map 処理を行うレコード数が削減されるため，データの読み込みコスト，map 処理コストが削減される．インデックスの読み込み，更新，書き込みコストが新たに加わわるものの，全体としては従来の手法と比べ大きなコストを削減を達成した．図 6(c)) は，データをシーケンシャルにアクセスしローカルでランダムアクセスを行った場合である．この場合，読み込みデータ量は削減されないが，Map タスクの処理は削減できる．結果を見ると，他の 2 つの場合に比べて削減幅は小さいものの，従来の処理に比べて処理が削減できている．

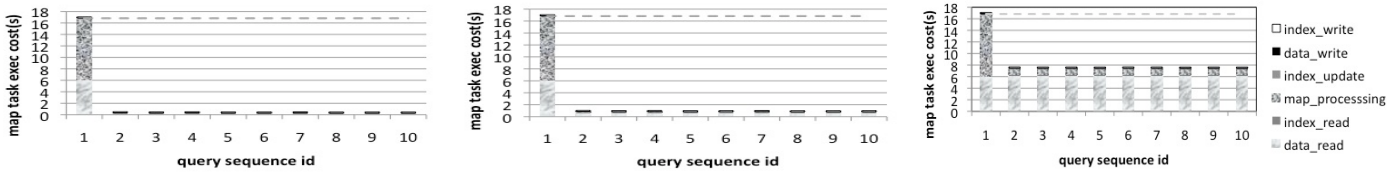
図 7 は， クエリ選択率に応じた MapReduce ジョブの実行コスト変化を表している．Map タスク実行時間と同様に，クエリ選択率によらず大幅な実行コストの削減を達成している．2 つの実験により，提案手法がクエリ選択率によらず、インデックス追加によるオーバヘッドを抑え実行コストを大幅に削減できることが確認できた．

5.3 データの追加

定期的に新しいデータが追加される状況をシミュレートするため，はじめに約 950GB のデータが存在し，10 回のクエリごとに 100G のデータを追加することを考える．図 8 がジョブ実行コストの時間変移を表す．従来の処理，提案手法ともに，データ追加時に実行時間が増える．従来の処理がデータ量に実行時間が比例するのに対して，提案手法ではデータ追加直後に実行時間が大きくなるものの，その後の実行時間は抑えている．提案手法は前に作成したインデックスを生かすことができるので，データの追加に対して性能が頑強であることを示している．

6. 関 連 研 究

MapReduce 処理系の高速化手法として，DBMS の技術を適用する研究が多くなされている [11]〜[14]．カラムストア技術を用いる RCFile [14] は，データスプリットを物理的に複数の



(a) 選択率: 0　　(b) 選択率: 1.0E-5　　(c) 選択率: 0.1

図 6　選択率に応じた Map タスクの実行コストの変化．点線が従来のコストを表す．

図 7　クエリ選択率に応じた MapReduce ジョブ実行コストの変化

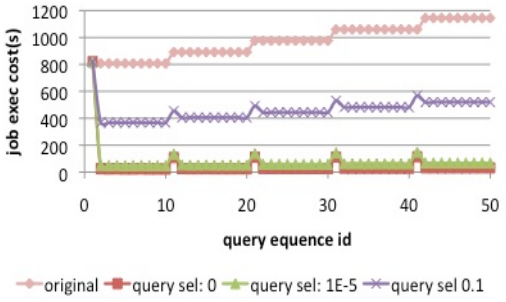


図 8　データの追加による MapReduce ジョブ実行コストの変化

レコードユニットに分割し、レコードユニットごとにカラムストアを構築する．RCFile におけるカラムストアの構築方法は本手法に適用可能であり，性能を向上できると期待できる．

本手法と類似した研究に，適応的なワークロードに対応するために，ジョブ間の中間データを再利用する Restore [15] である．Restore は，Hive [16] などの高レベルの言語でクエリが記述され複数の MapReduce ジョブにコンパイルされ実行される環境を想定する．Restore では，従来破棄されていたジョブ間の一時データをキャッシュし，同じようなジョブ構成を持つクエリの際にキャッシュを活用することで，クエリの高速化を図る．レンジ条件が同一でないと高速化できないなど，高速化可能な条件が本手法より厳しい．

7. 結　　論

本稿では，MapRedcue 処理系でアドホックなデータ解析を行う時に，アドホックであるが同じようなレンジ条件を持つプロジェクションクエリを高速化する手法を述べた．提案手法では，データスプリットを論理的なレコードユニットに水平分割し，レコードユニットごとに，クエリ実行時にインデックスの作成，更新を行う Database cracking を適用した．シミュレーションによる実験の結果，従来の MapReduce 処理と比較し，ジョブ実行コストを大きく削減することを確認した．今後の課題は，提案手法を実装し効果を確かめることである．

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] 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.

[4] M. Ahuja, C.C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, “Peta-scale data warehousing at yahoo!,” SIGMOD Conference, pp.855–862, 2009.

[5] 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.

[6] 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.

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

[8] “Hdfs”. http://hadoop.apache.org/hdfs/

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

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

[11] 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.

[12] 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

[13] A. Floratou, J.M. Patel, E.J. Shekita, and S. Tata, “Column-oriented storage techniques for mapreduce,” Proc. VLDB Endow., vol.4, no.7, pp.419–429, April 2011.

[14] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, “Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems”.

[15] I. Elghandour and A. Aboulnaga, “Restore: Reusing results of mapreduce jobs,” PVLDB, vol.5, no.6, pp.586–597, 2012.

[16] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy, “Hive - a petabyte scale data warehouse using hadoop,” ICDE, pp.996–1005, 2010.