

## MapReduce 上の編集距離結合における 2 段階ハッシュ分割技法の効果

Effects of two-level hash-partitioning for edit-distance join on mapreduce

大森匡†

今野篤人†

新谷隆彦†

Tadashi OHMORI

Atsuhito KONNO

Takahiko SHINTANI

## 1. はじめに

mapreduce モデルで類似結合を計算する算法は多く提案されているが, 算法固有のパラメタとデータに応じて性能特性が大きく異なることが知られている [1]. 著者らは文献 [3] において, mapreduce 上の編集距離結合の既存技法の 1 つ landmark join [2] を例にとり, prefix-filtering 技法に特有なレコードコピー量を制御して map/shuffle コストの削減と reduce 処理の効率化を行うため, 2 段階に分けたハッシュ分割技法を 2 つ提案している. 本稿では, そのうちの 1 つである, Q1/Q2 分割法について実装評価を述べる.

## 2. 問題の概要

## 2.1 編集距離結合と既存技法 landmark join

文字列  $x, y$  の編集距離  $ed(x, y)$  とは,  $x$  を  $y$  に変換する単位操作として 1 文字の削除, 追加, 置換の 3 つを用意したとき, その総回数の最小値である. 編集距離の閾値を  $\tau$  としたとき, 文字列データの集合  $X$  と  $Y$  の編集距離結合 (edit-distance join) とは,  $x \in X, y \in Y$  のうち,  $ed(x, y) \leq \tau$  となる組み  $(x, y)$  を全て求めることである. 本稿では,  $X = Y$  の場合, つまり自己結合 (self-join) のみを扱う. また, 各文字列 1 つを 1 レコードとして扱い, そのレコード集合  $X$  を入力として mapreduce 処理を行う.

本稿では, MapReduce 向けの編集距離結合アルゴリズムの 1 つである landmark join [2] を題材に選び, map/shuffle/reduce の負荷効率化を考える.

landmark join は, 文献 [1] で言うところの部分列取り出し (subsequence) 戦略を prefix に適用した手法の一つである. 2 ジョブ構成であり, 1 ジョブ目 (job1) で重複した正解候補の集合を求め, 2 ジョブ目で重複除去を行う. この 1job 目 (job1) の map, reduce 処理は以下の通り [2,3]:

(1) map 側の処理:  $q$ -バケット分割

map タスクは, 各入力レコード  $x$  について,  $x$  の先頭から長さ  $q + \tau$  の接頭辞 (prefix) を考え, そこから長さ  $q$  となる文字の組み合わせを全て選び, 各組み合わせをラベル  $L_i$  とする. そして,  $\langle \text{key}, \text{value} \rangle$  形式の出力として,  $L_i$

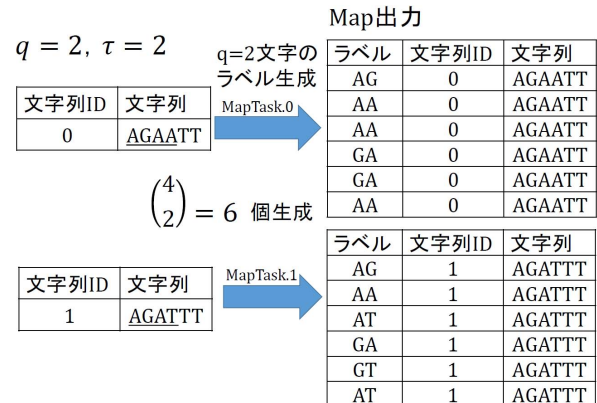


図 1: Landmark Join の map 処理

を key とし, 3 つ組  $[x$  のレコード ID,  $x$  の値, 付加情報] を value としたレコードを作り, shuffle に渡す. 結果, 入力 1 レコードあたり,  $L_i$  は  $\binom{q+\tau}{q}$  個生成され, それと同数のレコードが shuffle に出力される ( $x, y$  の先頭  $q + \tau$  文字の編集距離が  $\tau$  を超えているなら, 必ず  $ed(x, y) > \tau$  である, という性質を使っている).

例: 2 つの文字列レコード AGAATT と AGATTT に対して,  $q = \tau = 2$  のときの map 動作例を図 1 に示す. 図中, レコード  $x = \text{AGAATT}$  の場合,  $x$  の長さ  $q + \tau = 4$  の接頭辞 ( $x_0^4 = \text{AGAA}$ ) から 2 文字選んで作った長さ 2 の文字列をラベル  $L_i$  として作る.  $L_i$  は,  $L_0 = \text{AG}, L_1 = \text{AA}, L_2 = \text{AA}, L_3 = \text{GA}, L_4 = \text{GA}, L_5 = \text{AA}$  の 6 つである.  $L_1, L_2, L_5$  の値は AA で同じだが, 生成に使われた  $x$  中の文字の座標は異なる. この文字座標を, landmark join では付加情報  $pos(L_i, x)$  として value の一部として出力レコードに含め, reduce が  $L_i$  の一致するレコード間で行う照合処理の削減に用いる.

## (2) reduce 側の処理

reduce 処理は, ラベル  $L$  が一致するレコード集合に属すレコード  $x, y$  についてのみ編集距離の計算を行う. この時,  $\tau$  を越える  $ed(x, y)$  の計算回数を減らすため,  $pos(L_i, x)$  と  $pos(L_i, y)$  を用いて, 各  $x, y$  の接頭辞で生じる編集距離の上界値 ( $ub_1$ ) を先に求め, その値が  $\tau$  以下のときのみ, さらに  $x, y$  全体で生じる編集距離の上界  $ub(x, y; L)$  を計

† 電気通信大学, UEC

算する．そして， $ub(x, y; L) \leq \tau$  のときのみ，その値と  $(x, y)$  を解候補として出力する．

以上が job1 である．2job 目 (job2) では各  $(x, y)$  について上記の出力値の最小値を求めて正確な  $ed(x, y)$  値とする．

## 2.2 修正版 landmark join [3]

我々は文献 [3] で，下記のように landmark join を修正して用いている：

(1) map 側処理の修正：入力レコード  $x$  に対し， $x$  から生成したラベルを  $L$ ， $x$  のレコード ID (rid) を  $ridX$ ， $x$  の文字列を  $strX$ ，としたとき， $L.rid$  を  $key$ ，3 つ組  $[ridX, strX, pos(L, x)]$  を  $value$  として shuffle へ出力．( $L$  で行き先となる  $reduceID$  を決め， $L.rid$  の 2 フィールドでソートする)．

(2) reduce 側の修正：Shuffle からの入力レコード列を， $L$  が一致する入力の集合  $R_L$  を単位にして扱う．さらに， $R_L$  を， $L.rid$  が一致するレコード集合に分割して考える．この 1 分割は， $rid$  (レコード ID) が一致するレコード  $x$  から作られ， $pos(L, x)$  が違うだけのレコード群であり，これらを 1 つにまとめて， $[ridX, strX, [posX]]$  と表記する． $posX$  は，1 つの  $pos(L, x)$  を表し， $[posX]$  は，そのような  $posX$  の集合である．このとき， $R_L$  に対する reduce 処理  $reduce1()$  を，Algorithm1 に示す．

---

### Algorithm 1 $reduce1(R_L)$

---

**Input:** ラベル  $L$  に属すレコード集合  $R_L$

**Output:**  $ed(x, y) \leq \tau$  となる  $(x, y)$  と正確な  $ed(x, y)$  値

```

1: for all rid が異なる  $[ridX, strX, [posX]]$  と
    $[ridY, strY, [posY]]$  の対 (pair) in  $R_L$  do
2:   if  $Dmap(ridX, ridY) == false$  then
3:     for all  $(posX, posY)$  in  $[posX] \times [posY]$  do
4:        $C1 \leftarrow ub(strX, strY; L)$  の第一項;
5:       if  $C1 \leq \tau$  then
6:          $Dmap$  に  $(ridX, ridY)$  を追加;
7:          $ed(strX, strY)$  を計算し， $\tau$  以下なら  $(x, y)$  と  $ed$ 
           値を出力;
8:         goto 1 行目; // 次の rid の対 (ペア) の検査へ.
9:       end if
10:    end for
11:  end if
12: end for

```

---

$reduce1()$  の特徴は次の通り：説明の都合上，4-5 行目の「 $ub()$  式第 1 項計算値 ( $ub_1$  のこと) が  $\tau$  以下か」を試す枝刈り処理を FirstTest と呼ぶ．2 行目の  $Dmap(ridX, ridY)$  は，一度でも FirstTest が成功した  $rid$  の組み ( $ridX, ridY$ ) を記憶する表である．このとき， $reduce1$  は，FirstTest が成功した元レコード  $x, y$  の組み ( $ridX, ridY$ ) について， $Dmap$  に登録後，直接，レコード全体の正確な編集距離  $ed(x, y)$  を 7 行目で計算し， $\tau$  以下なら正解として出力して，次の ( $ridX, ridY$ ) の検査へ移る．その結果，1 つの reduce タスク内では，特定の ( $ridX, ridY$ ) の FirstTest と  $ed(x, y)$  の

計算は高々 1 回しか行わない．以下，landmark join というときは，この修正版のことを指す．

## 2.3 処理効率上の問題

我々は修正版 landmark join を遺伝子データセット上で  $q = 16, \tau = 2$  で試したところ，レコードのコピー量の多さによって次の 2 点が処理効率の問題になった [3]：

(1) map/shuffle コストの増加：  $q = 16, \tau = 2$  の場合，map 処理の際に 1 レコード  $x$  を  $\binom{18}{2} = 153$  個コピーする事になる．このコピーの量は無視できず，map 処理のコスト増加、shuffle フェイズの通信コストの増加になり，全体の計算時間の支配項になる．(reduce 処理の計算時間は軽くなっている)

(2) reducer 間の冗長計算：1 レコードを 153 倍にコピーして  $R$  個の reducer に分配するため，例えば，5 ノードクラスタで  $R = 5$  の場合、ほとんどの入力レコード  $x$  は全 reducer にコピーされて配布される．その結果，全 reduce タスクは，ほぼ同じレコードの組み合わせ  $(x, y)$  の検査を (異なるラベルの下で) 重複して行う．本来なら，複数の reducer 間で， $(x, y)$  の検査をできるだけ排他的に分割して担当したい．

## 3. 2 段階ハッシュ分割技法

### 3.1 効率化の考え方

2.3 の問題を解決するためには，map で入力 1 レコード  $x$  を変換しラベルをつけて reduce に送り出すとき， $x$  の行き先になる reduce タスクの数  $C_r$  を，システムが用意した reduce の総数  $N_r$  より大幅に小さくなるよう制御すれば良い．

例えば，landmark join で  $q = 16, \tau = 2$  なら，そのままでは map 処理は 1 レコード  $x$  を 153 個コピーして shuffle へ送るが，もしシステムの reducer 数が 20 個なら， $x$  の宛先になる reduce の総数を 10 程度に制御できれば良い．もちろん，同一ラベルを持つレコード群は必ず同一の reduce へ送る必要があるし，しかも，全 reducer に処理を分配すべきである．これができれば，システム的全ノード間の通信による総当たり処理よりも shuffle コストを下げ，しかも，複数 reduce タスク間での同一レコード対の照合の重複を削減できる．

この考えを landmark join に適用した手法として，我々は，Q1/Q2 分割法と呼ぶ方法と，ラベル prefix 分割法の 2 つを提案している [3]．以下に Q1/Q2 分割法の戦略と正確な実装方法を述べる．

### 3.2 Q1/Q2 分割法の設計と実装

まず，[3] で述べた Q1/Q2 分割法の設計方針を述べる：

[Q1/Q2 分割法：  $q = q_1$  と  $q_2$  を使った 2 段階ハッシュ分割方式]:

$q$ -バケット分割 (2.1 節の (1) の方法) を用いるが, map と reduce の各々において,  $q$  の値を変えて独立して  $q$ -バケット分割を行う方法である. すなわち, map 処理においては,  $q$  の値を小さく (例えば,  $q_1 = 3$  に) 設定して  $q$ -バケット分割を行い,  $q = q_1$  により生成されたラベル  $L_{q_1}$  を key に, レコード ID とレコード実体  $x$  を value にして shuffle へ送出する (正確には,  $L_{q_1}.rid$  をキーに, 行き先を  $L_{q_1}$  で決める). 例えば,  $q_1 = 3, \tau = 2$  なら, 1 レコード  $x$  のコピー数は  $\binom{5}{2} = 10$  個であり, 同一ラベルもありうるので,  $x$  の送り先になる reduce タスクの総数は 10 以下になる. reduce 処理では,  $L_{q_1}$  ラベルの一致するレコード集合ごとに, 当該レコードから新たに (効率的な照合に最適な)  $q = q_2$  (例えば,  $q_2 = 16$ ) によるラベル  $L_{q_2}$  を作成して,  $L_{q_2}$  の一致するレコード群ごとに landmark join の reduce 部の照合処理を行う. []

上記の設計方針に基づき, 本稿では, 次のように Q1/Q2 分割法の map/reduce 処理を実装した:

\* map 側処理: レコード  $x$  から  $q = q_1$  により生成されたラベル  $L_{q_1}$  を key に, レコード ID (ridX) とレコード実体  $x$  を value にして  $\langle key, value \rangle$  形式で shuffle へ送出する. 正確には,  $L_{q_1}.ridX$  を key にし, 行き先 reducer を  $L_{q_1}$  で決める. reducer 側のソートは  $L_{q_1}.ridX$  の 2 フィールドで行い, グループ化は  $L_{q_1}$  で行う. また, 修正版 landmark join やラベル prefix 分割法とは異なる点として, Q1/Q2 分割法では, reducer 側がラベル  $L_{q_1}$  で受け取るレコード  $x$  については, 付加情報  $pos(L_{q_1}, x)$  を必要としない. そのため, map 側では, レコード  $x$  から同一ラベルが複数生じて, 重複除去して, ユニークなラベル値についてのみ, 付加情報をつけずに shuffle へ送出する. (例: 図 2).

\* reduce 側処理:  $L_{q_1}$  が一致するレコード集合に属すレコード  $x$  について  $q = q_2$  のラベル  $L_{q_2}(x)$  を生成して,  $L_{q_2}$  が一致するレコードと照合する. 本稿では, reducer 内で  $L_{q_2}$  に関してエントリ ( $L_{q_2}$ , レコード  $x$ ) を登録するハッシュ表  $T$  を用意して,  $L_{q_2}$  と一致する登録済みレコード  $y$  と Algorithm1 に沿った照合処理を行う. すなわち,  $x$  から  $L_{q_2}(x)$  を 1 つ作る度に  $T$  を probe 照合し, その後に  $T$  に当該エントリを登録する. ここで重要な点として,  $L_{q_1}$  に属すレコード集合から生成する  $L_{q_2}$  については,  $L_{q_2}$  の先頭  $q_1$  文字が  $L_{q_1}$  に一致するものに限るように制約する. こうすることで, 相異なる  $L_{q_1}$  に属すレコード集合の処理の際に同じ  $L_{q_2}$  を作る可能性を排除し, 冗長な照合処理を避ける. []

上述した方式では, 1 レコードあたりの行き先 reduce の数を  $\binom{q_1+\tau}{\tau} (\ll N_r)$  としており, しかも, 正解を脱落しない.  $q_1$  が小さく, 補助情報も不要なため, shuffle コス

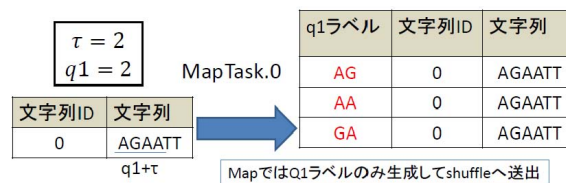


図 2: Q1/Q2 分割法の map 処理

トを大幅に削減できる. しかし, その分, reduce 内で行う  $q = q_2$  による処理が重くなる. (そもそも, 各 reduce 内では 1 ノード向けの別の計算アルゴリズムを使うこともできる.) そこで, 本稿では, reduce 内で 1 レコード  $x$  から  $L_{q_2}$  ラベルを生成してハッシュ表  $T$  を検索する際,  $x$  の付加情報  $pos(L_{q_2}, x)$  を null にして無視し, 相異なる  $L_{q_2}$  ラベルのリストだけに射影してから  $T$  を検索して reduce1() 関数を実行するオプションをつけた. これにより  $x$  から生じた  $L_{q_2}$  が同じで  $pos()$  情報が異なるだけによるループ回数を削減できる. 他にも, Q1/Q2 分割法は,  $q_1$  に依存してデータ分配するため負荷の偏りが生じやすい, mapreduce モデルのソートマージ機能を積極的に使っていない, という課題がある.

これに対して, ラベル prefix 分割法は, 先に map 側で  $q = q_2$  のラベル  $L_{q_2}$  を作るが, その先頭  $q_1$  文字の prefix をラベル  $L_{q_1}$  として用い,  $L_{q_1}.L_{q_2}.rid$  を key, 4 つ組み [rid,  $L_{q_2}$ ,  $x$  の文字列,  $pos(L_{q_2}, x)$ ] を value とした  $\langle key, value \rangle$  レコードを shuffle に送り出して, reduce 側で  $L_{q_2}$  の一致するレコード集合内で reduce1() の照合処理を行う方式である. shuffle 量を減らすため, reducer 1 つには同一レコードは高々一回しか送らないようにするが, 補助情報  $pos(L_{q_2}, x)$  は省略できず, reduce 側で使用する. また,  $L_{q_1}$  が高々  $\binom{q_1+\tau}{\tau}$  個しかないので, reducer 間の冗長な照合処理を削減できる.

## 4. 評価実験

### 4.1 実験条件

修正版の landmark join (一段階方式と呼ぶ) と, 2 段階ハッシュ分割法である Q1/Q2 分割法, ラベル prefix 分割法の合計 3 つを実装して, 遺伝子データを使った編集距離結合時の動作特性を調べた.

使用したデータは [3] と同じ遺伝子データ 300,000 件である. (1 文字を A, C, G, T, N のどれか 1 つとして, 1 レコードは長さ 100 文字前後の文字列). 使用環境は, 5 ノード構成の Hadoop クラスタ (Xeon 3.2GHz/4 コアを 4 ノード, 2.6GHz/4 コアを 1 ノード, ノードあたりメモリ 12GB, ノードあたりの同時並行 map 数, reduce 数は各 2. [3] の場合よりも速いノード構成) である. Hadoop Streaming ライブラリを使用して各方式を実行し, 起動する map タ

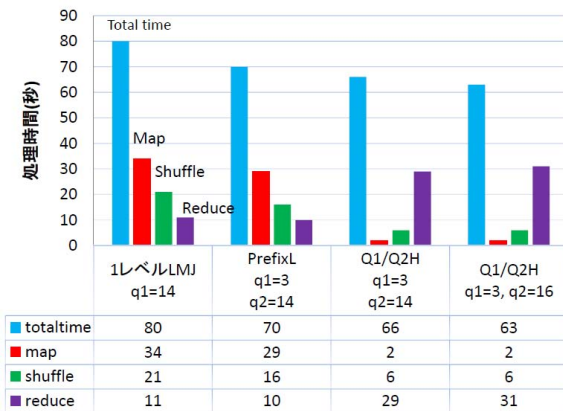


図 3: 各方式の map/shuffle/reduce 処理時間 (秒)

表 1: 各方式の shuffle 量, reduce あたり平均出力数

	1-level LMJ	prefix-L q1=3, q2=14	Q1/Q2-H q1=3, q2=14
shuffle 量	5.1GB	1.99GB	0.19GB
平均出力数	14203 組	7345 組	6187 組

スケル総数と reduce タスク総数は各 10 とした。

実装では, ラベル prefix 分割法と一段階方式の 2 つは直接 C 言語で記述した。Q1/Q2 分割法については, 編集距離計算関数は同じ C 版を使い, 全体の制御を python で記述して Cython で C に変換したものを用いた。[3] と同じく, 編集距離結合の閾値は  $\tau = 2$  とし, Q1/Q2 分割法とラベル prefix 分割法については, [3] の評価で最も処理時間が良かったパラメタ値  $q1 = 3, q2 = 14$  を基準にした。

#### 4.2 処理時間, Shuffle 量, reduce 間の重複計算の度合い

図 3 に, 各方式について編集距離結合の総処理時間 (total time), map タスク 1 つあたりの平均処理時間, shuffle 1 つあたりの平均処理時間, reduce タスク 1 つあたりの平均処理時間, を示した。図中の横軸は左から順に, 一段階方式 (1 レベル LMJ), ラベル prefix 分割法 (prefix-L), Q1/Q2 分割法 (Q1/Q2H) の  $q1 = 3, q2 = 14$ , Q1/Q2 分割法 (Q1/Q2H) の  $q1 = 3, q2 = 16$ , である。Q1/Q2 分割法は,  $pos()$  情報を無視するオプションの場合である ( $pos()$  情報を使う場合は, reduce 処理時間は図の値より 10 秒程度遅かった。) また, 表 1 に  $q2 = 14$  に相当する場合の各方式の shuffle データ総量と reduce タスクあたりの平均正解組の数を示す。

図 3 において,  $q2 = 14$  に相当する場合について見ると, 総処理時間は一段階方式 ( $q = 14$ ) の 80 秒に比べラベル prefix 型分割法の方が 70 秒で短い。これは前回と同じ傾向である。さらに Q1/Q2 分割法の処理時間は 66 秒で最速となった。これは一段階方式とラベル prefix 分割法が共に map/shuffle 時間が支配項になるのに対して, Q1/Q2 分

割法の設計方針が奏功したためである。すなわち, shuffle データ量 (表 1) が一段階方式の 5.1GB, ラベル prefix 分割法の 1.99GB から, Q1/Q2 分割法では 0.19GB まで低下したことが反映されている。

一方, Q1/Q2 分割法では, reduce 処理時間が支配項になるため, コピーしたレコード負荷の偏りとノードの速度差の影響を受けて, reduce の処理時間は 12 秒から 45 秒の間に大きくばらつき, total time が下らない原因になった。

reduce タスク間の重複計算の排除の度合いについては, 表 1 を見ると, reduce 1 つあたりの正解出力数の平均値は, 一段階方式では 14203 組に対し, ラベル prefix 分割法では 7344 組, Q1/Q2 分割法では 6188 組であった。(ユニークな正解総数は 15958 組)

最後に, Q1/Q2 分割法について,  $q2 = 14$  の場合から  $q2 = 16$  に変えたところ, 処理時間は 63 秒まで漸減した。これは  $q2$  が大きい方が reduce 内のデータ偏りによる CPU 時間の差を軽減できたためである。

## 5. まとめ

mapreduce モデル上の類似結合の算法は多いが, データ分布やパラメタによって map/shuffle/reduce コストが大きく異なることが知られている。本稿では, 編集距離結合を対象に, 部分列取り出し戦略による既出技法の一つ landmark join[2] を例にとり, map/shuffle コストを削減して reduce 側の設計自由度を上げる戦略として, 2 段階ハッシュ分割技法の一つ Q1/Q2 分割法を述べ, その実装と動作特性を述べた。遺伝子データ上の試験から, Q1/Q2 分割法の「reduce 総数よりレコードコピー量を下げて, shuffle 量削減と reduce 間の重複計算を排除する」という戦略は, landmark join のような典型的な prefix filtering 技法の算法では効率化に有効であった。一方で, 部分列取り出し戦略の下で Q1/Q2 分割法を使うだけでは, shuffle 量削減の他の効率化は難しい。reduce 側の処理アルゴリズムを別の手法に変えて Q1/Q2 分割のような戦略を適用することなどが課題である。

謝辞 本研究の一部は科研費 (課題番号 24500109) による。

## 参考文献

- [1] F.N.Afrati, et al., "Fuzzy Joins Using MapReduce," ICDE'12, pp.498-509, 2012.
- [2] K.Narita, et al., "Landmark-Join: Hash-Join Based String Similarity Joins with Edit Distance Constraints," DaWaK 2012, pp.180-191, 2012.
- [3] 今野, 大森, 新谷, "MapReduce における編集距離結合の負荷分散・効率化方式," DEIM 2015, E7-6, 2015.