情報処理学会論文誌 Vol.62 No.1 302–320 (Jan. 2021)

シーケンスデータに対する行パターンマッチングの効率化

中挾 晃介[1,a)] 北川 博之[2,b)]

受付日 2020年3月18日, 採録日 2020年10月6日

概要：近年，情報通信技術，センサ技術の発展にともない，シーケンスデータが日々大量に生成・処理されている．このシーケンスデータに対して，行パターンマッチングを行う操作の標準として，SQL/RPRがある．SQL/RPRは，行パターンマッチングを実現するためのSQLの拡張である．一方で，行パターンマッチングの処理コストは，SelectionやJoinといった処理と比べると大きく，効率的に処理する方法を考える必要がある．本研究では，RDB等に格納された膨大な量のシーケンスデータに対する行パターンマッチングの処理コストを削減するために，SelectionやJoinといった相対的に処理コストの小さい処理を組み合わせ，行パターンマッチング対象となる行を減らす前処理を事前に適用することにより，処理コストを削減する2つの効率化手法（Sequence FilteringとRow Filtering）を提案する．また，本研究では，PostgreSQLおよびSparkにSQL/RPRを実装し，これらの効率化手法が行パターンマッチングの処理時間を削減することを確認する．さらに，各効率化手法を適用した場合の処理時間を見積もるコストモデルを構築，検証し，適切な効率化手法を選択可能であることを示す．

キーワード：シーケンスデータ，行パターンマッチング，SQL/RPR，MATCH_RECOGNIZE

Efficient Row Pattern Matching over Sequence Data

Kosuke Nakabasami[1,a)] Hiroyuki Kitagawa[2,b)]

Received: March 18, 2020, Accepted: October 6, 2020

Abstract: Due to the advance of information, communications, and sensor technology, a large quantity of sequence data is generated and processed every day. Row pattern matching for the sequence data was standardized as SQL/RPR in 2016. SQL/RPR is an extension of SQL for realizing row pattern matching. However, computational cost of the row pattern matching process is large and it is needed to make this process efficient. In this paper, we propose two methods for this purpose: Sequence Filtering and Row Filtering which realize the reduction of processing time for row pattern matching by filtering input data in advance. We implement SQL/RPR for PostgreSQL and Spark and verify that our methods can reduce the processing time of queries including row pattern matching. Also, we construct a cost model for estimating processing time of queries when the proposed methods are applied and show that users can select a proper processing method.

Keywords: sequence data, row pattern recognition, SQL/RPR, MATCH_RECOGNIZE

1. はじめに

近年の情報通信技術，センサ技術の発展にともない，様々なデータが日々，生成・配信されており，それにともない大量のデータが処理・蓄積されている．これらのデータには，データの連続性が重要な意味を持つシーケンスデータが多く含まれる．これらのシーケンスデータに対しては，ある特定のパターンが含まれるかどうかを検出する行パ

[1] 公益財団法人鉄道総合技術研究所信号・情報技術研究部
Signalling & Transport Information Technology Division, Railway Technical Research Institute, Kokubunji, Tokyo 185–8540, Japan
[2] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[a)] nakabasami.kosuke.39@rtri.or.jp
[b)] kitagawa@cs.tsukuba.ac.jp

ターンマッチングが重要となる．ここで，行パターンマッチングとは，行のシーケンスからパターンオカレンスを検出する手法である．一般的には，シーケンスデータ内の各行は，株価 ID や個人 ID のようなシーケンス固有の属性（*sequence_id*）と，時刻のような行の順序を定める属性（*order_key*）と，株価や地点のような各イベント（行）に固有の属性からなる．sequence_id によってデータをパーティショニングし，order_key によってソートすることで，各 sequence_id の行のシーケンスが得られる．このようなシーケンスに対し，ユーザがある行のパターンを定義することで，そのパターンに合致するすべてのオカレンス，つまり行のサブシーケンスが抽出されることにより，行パターンマッチングが行われる．行パターンマッチングは，シーケンスデータに対する分析において本質的に重要な処理である．

RDB 等に蓄積された静的なシーケンスデータに対する行パターンマッチングは，SQL/RPR（Row Pattern Recognition）[8] として SQL2016 において標準化されている．SQL/RPR は，行パターンマッチングを実現するための SQL の拡張である．ユーザは，シーケンスから抽出するパターンを定義するために，FROM 句中に MATCH_RECOGNIZE 句を指定し，行パターンマッチングを行う．

一方で，この行パターンマッチングには，Selection や Join 等のクエリ処理に対し，その処理コストが相対的に大きい．行パターンマッチングの処理コストについては，4 章において詳説する．そのため，この行パターンマッチングを効率的に処理する方法を考える必要がある．

著者らの論文 [11] では，RDB 等に格納された膨大な時系列データに対する行パターンマッチングの処理コストを削減するために，行パターンマッチング処理の前に，Selection や Join といった相対的にコストの小さい処理を組み合わせ，行パターンマッチング対象となる行数を削減する 2 つの効率化手法を提案した．これは，MATCH_RECOGNIZE 句に含まれる行パターンマッチングのための条件を用いて入力データをフィルタリングする手法である．1 つ目は，MATCH_RECOGNIZE 句の条件を満たす行がシーケンス内に 1 行も存在しないシーケンスを事前にフィルタリングする *Sequence Filtering* である．2 つ目は，MATCH_RECOGNIZE 句の条件を満たさない行を事前にフィルタリングする *Row Filtering* である．これら 2 つの効率化手法の効果を，並列分散処理環境を用いて大量のデータを処理するフレームワークの 1 つである Spark [18] に SQL/RPR を実装し検証した．一方で，文献 [11] の Sequence Filtering では，パターンオカレンスが存在し得ない一部のシーケンスがフィルタリングされずに残ってしまうという課題がある．また，Row Filtering についても，Row Filtering を適用するための条件指定に制約が存在し，実用場面において課題が存在する．

そこで，本研究では，文献 [11] で示した Sequence Filtering よりも多くのシーケンスをフィルタリングすることで処理コストを削減する新しい Sequence Filtering と，パターンオカレンスの最大行数がクエリ内で指定したパターンから定まることを利用し，前述した制約を緩和する新しい Row Filtering を提案する．また，本研究では，Spark に加え，UDF により SQL/RPR を実装した PostgreSQL を用いた評価実験により，2 つの効率化手法による処理コスト削減の効果を検証する．さらに，本研究では，各効率化手法の処理時間を見積もるコストモデルを構築する．行数やシーケンス数の異なるデータ，パターンの異なるクエリに対して処理時間を見積り，実際の処理時間との誤差を比較することで，構築したコストモデルの妥当性を検証する．これにより，適切な効率化手法の選択が可能となることを示す．

以降では，2 章において関連研究を示す．3 章では，SQL/RPR で標準化されている MATCH_RECOGNIZE 句について説明する．4 章では，行パターンマッチングの処理コストを示す．5 章では，本研究で新しく提案する Sequence Filtering と Row Filtering の 2 つの効率化手法について説明する．6 章では，提案する効率化手法のコストモデルを示す．7 章では，UDF を用いた PostgreSQL への SQL/RPR の実装方法を示す．8 章では，Spark への SQL/RPR の実装方法を示す．9 章では，効率化手法の効果を検証するための評価実験と 6 章において示したコストモデルの検証について述べる．最後に 10 章において本研究のまとめと今後の課題を示す．

2. 関連研究

シーケンスデータに対するパターンマッチングとしては，文字列に対するパターンマッチングが存在し，この処理を高速化する研究が広く行われている．Wang らは，文字列検索に用いる部分パターンと，繰返しを表す正規表現を含む部分パターンに分割し，決定性有限オートマトン（DFA）により高速に処理が可能な，文字列検索に用いる部分パターンからパターンマッチングを行うことにより高速化する手法を提案している [14]．また，文字列の最初の数文字がパターンにマッチするかを判定する前処理を行う手法 [5]，Bloom Filtering による前処理を用いる手法 [4]，文字列を分割し MPI を用いて複数スレッドでパターンマッチングを行うことで高速化する手法 [13] 等が提案されている．

一方で，文字列に対するパターンマッチングでは，そのオートマトンにおける状態の遷移の条件が単一かつ特定の文字との等価条件が前提であり，上記の文献はこの前提の上での手法を提案している．本研究で対象とする行パターンマッチングは，オートマトンにおいて状態の遷移の条件が複数かつ非等価条件も含む点，遷移の条件として今まで

マッチしてきた行の値を参照することができる点において，文字列に対するパターンマッチングとは異なり，上記の文献の手法を直接適用できない．また，本研究は，行パターンマッチングのようなコストの大きい処理が含まれるクエリの処理コストを削減することを目的に，Selection 等の既存の相対的に小さいコストの処理を組み合わせたクエリを用いて，全体の処理コストを削減する手法を提案するものである．一方で，上記の文献の手法は，パターンマッチングの処理の高速化のために，そのアルゴリズムを独自実装したエンジンを構築するという点において，本研究とは目的が異なる．

ストリームデータに対してパターンマッチングを行う手法として最も知られているものの 1 つに Complex Event Processing（CEP）がある．CEP の例としては，Cayuga [6]，SASE [15]，ZStream [10] 等があげられる．また，本研究のように，前処理によって行パターンマッチングの処理コストを削減する手法として，Cadonna らは，前処理フェーズとパターンマッチングフェーズからなる行パターンマッチング手法を提案している [2]．一方で，本研究では，静的なシーケンスデータに対する行パターンマッチング処理における効率化を対象としている．

MATCH_RECOGNIZE 句を実装した RDBMS における，行パターンマッチングの効率化も提案されている [9]．この効率化は，ソート手法の選択，クエリの実行計画の変更，パターンマッチングにおける DFA と NFA の選択の 3 つの要素からなる．特に，2 つ目の，クエリの実行計画の変更は，クエリ中の WHERE 句において sequence_id に対する条件が存在する場合，MATCH_RECOGNIZE 句の前にプッシュダウンし，シーケンスをフィルタリングするというものである．本研究において提案する効率化手法は，文献 [9] で提案されている効率化に対して次の 2 点において異なる．1 点目は，シーケンスのフィルタリングに用いる属性である．既存手法では，sequence_id の条件のみによってシーケンスがフィルタリングされるが，提案手法では，属性に対する制約なく，シーケンスのフィルタリングを行う．2 点目は，シーケンスによらない個々の行のフィルタリングである．既存手法では，シーケンス単位でのみフィルタリングを行うが，本研究では，行単位でフィルタリングを行い，パターンマッチング対象となる行数をさらに削減する．

著者らの研究 [11] において，シーケンス単位でフィルタリングを行う Sequence Filtering と行単位でフィルタリングを行う Row Filtering を提案している．ただし，Sequence Filtering については，パターンオカレンスが存在し得ない一部のシーケンスがフィルタリングされずに残ってしまうという課題がある．また，Row Filtering については，適用するための条件指定に制約が存在する．この制約については 5.2 節において述べる．本研究では，文献 [11] で示した Sequence Filtering よりも多くのシーケンスをフィルタリング可能な新しい Sequence Filtering を提案する．また，適用するための条件指定の制約を緩和した新しい Row Filtering を提案する．

3. SQL/RPR における MATCH_RECOGNIZE 句

SQL/RPR における MATCH_RECOGNIZE 句を説明する．これは RDB に格納されたシーケンスデータに対する行パターンマッチングを行い，出力としてパターンオカレンスのテーブル（パターンオカレンステーブル）を得るものである．**図 1** に，MATCH_RECOGNIZE 句の文法を示す．また，**図 2** に MATCH_RECOGNIZE 句を用いたクエリの例（Example Query）を示す．

図 2 の Example Query は，person_id（sequence_id に対応），time（order_key に対応），location（各行に固有の属性に対応）の 3 つの属性を持つ，人の移動の時系列データ（moving_table）に対して，地点 A-> 任意の地点 -> 地点 C の順に移動するようなパターンの行パターンマッチングを行い，最終的に出力されるパターンオカレンステーブルとして地点 A と地点 C の時刻を含むテーブル（result_table）が得られるクエリである．

3.1 MATCH_RECOGNIZE 句の構文

MATCH_RECOGNIZE 句は FROM 句内に指定する．図 1 における `<table_name>` に行パターンマッチ

```
SELECT …
FROM <table_name> <pattern_recognition_clause> AS <result_table> , …
WHERE …
```

```
<pattern_recognition_clause> ::= “MATCH_RECOGNIZE (”
                                 “PARTITION BY” <sequence_id_list>
                                 “ORDER BY” <order_key_list>
                                 “MEASURES” <measure_list>
                                 “ONE ROW PER MATCH” | “ALL ROW PER MATCH”
                                 “AFTER MATCH SKIP” { “PAST LAST ROW”
                                                    | “TO NEXT ROW”
                                                    | “TO FISRT” <correlation_name>
                                                    | “TO LAST” <correlation_name>
                                                    | “TO” <correlation_name> }
                                 “PATTERN (” <regexp> “)”
                                 “SUBSET” <subset_definition_list>
                                 “DEFINE” <corrname_definition_list> “)”
<sequence_id_list> ::= <sequence_id> { “,” <sequence_id_list> }
<order_key_list> ::= <order_key> { “,” <order_key_list> }
<regexp> ::= { <correlation_name> [<pattern_quantifier>]
           | “(” <regexp> “)” [<pattern_quantifier>]
           | <regexp> “|” <regexp> }
<pattern_quantifier> ::= { “*” | “+” | “?” | “{n}” | “{n,}” | “{n,m}” | “{,m}”
                         | “*?” | “+?” | “??” | “{n}?” | “{n,}?” | “{n,m}?” | “{,m}?” }
<corrname_definition_list> ::= <correlation_name_definition>
                               { “,” <corrname_definition_list> }
<correlation_name_definition> ::= <correlation_name> “AS” <cond_list>
<cond_list> ::= <cond_list> { AND | OR | XOR } <cond_list>
              | NOT <cond_list>
              | “(” <cond_list> “)”
              | <condition>
              | <between_condition>
<condition> ::= <one_variable_cond> | <count_cond> | <other_cond>
<one_variable_cond> ::= <correlation_name>“.”<column_name>
                        { “<” | “<=” | “>” | “>=” | “=” | “<>” } <non_attr_arith_expr>
<count_cond> ::= “COUNT (” <correlation_name> “.*)” { “<” | “<=” | “=” } <non_attr_arith_expr>
<non_attr_arith_expr> ::= <const_value>
                        | <non_attr_arith_expr> { “+” | “-” | “*” | “/” } <non_attr_arith_expr>
                        | “(” <non_attr_arith_expr> “)”
<correlation_name> ::= <パターン変数>
<column_name> ::= <<table_name>の属性名>
```

図 1 MATCH_RECOGNIZE 句の構文
Fig. 1 Syntax of MATCH_RECOGNIZE clause.

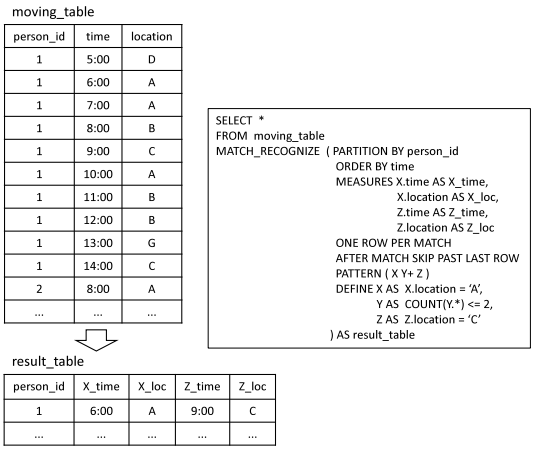


図 2 Example Query
Fig. 2 Example Query.

ング対象となるシーケンスデータのテーブルを指定する．図 2 の Example Query では，moving_table を指定している．本研究では，FROM 句に指定する MATCH_RECOGNIZE 句に関連する一連の部分，‘‘<table_name> <pattern_recognition_clause> AS <result_table>’’ を，*MATCH_RECOGNIZE specification* と呼ぶ．MATCH_RECOGNIZE 句内の構文は <pattern_recognition_clause> に示している．

<table_name> で指定した時系列データを個々のシーケンスにパーティショニングするための sequence_id は PARTITION BY 句で指定する．Example Query では person_id を指定することにより，各個人が経由した地点のシーケンスが形成される．

シーケンス内の行の順序を定める order_key は ORDER BY 句で指定する．Example Query では time を指定することにより，各個人が経由した地点のシーケンスが時刻順に並べられ，移動履歴のシーケンスが形成される．

パターンは PATTERN 句で指定する．任意の文字列を用いた ‘‘パターン変数’’ でマッチする行を表し，正規表現を用いてパターンを指定する．指定できる正規表現は図 1 における <regexp> に示している．Example Query ではパターン変数 X，Y，Z を用いて，地点 A と地点 C の間に任意の 2 地点以下を経由するパターンを指定している．なお，SQL/RPR における行パターンマッチングは，シーケンス内の互いに隣接する行のなかでパターンオカレンスを検出する strict contiguity によるマッチング方式をとっている [19]．そのため，パターン変数にマッチする行は互いに隣接している必要がある．

パターンの条件は DEFINE 句で指定する．パターン内の行の属性は，‘‘<パターン変数>.<属性名>’’ で表され，これを用いた条件式により指定する．Example Query ではパターン変数 X の地点を A，パターン変数 Z の地点を C と指定している．また，パターン変数 Y に対する条件のように，集約関数 COUNT を用いて 2 地点以下を経由するというように指定できる．本研究では，DEFINE 句における条件 <condition> を，*one variable condition*，*count condition*，*other condition* の 3 種類に分類する．one variable condition の定義を図 1 の <one_variable_cond> に示す．one variable condition は，式中に ‘‘<パターン変数>.<属性名>’’ が 1 つのみ存在し，集約関数等を含まない．count condition は，集約関数 COUNT を用いてあるパターン変数にマッチする行数あるいは最大行数を定める．count condition は，1 つのパターン変数に 1 つ指定するものとする．other condition は，one variable condition と count condition 以外のすべての条件指定である．Example Query ではパターン変数 X，Z に対する条件が one variable condition であり，パターン変数 Y に対する条件が count condition である．

SUBSET 句では，一部のパターン変数を結合し，1 つのパターン変数として再定義できる．

出力するパターンオカレンステーブルの属性（メジャー）は MEASURES 句で指定する．メジャーは，‘‘<パターン変数>.<属性名>’’，集約関数，およびそれらを組み合わせた式で定義できる．Example Query では，パターン変数 X にマッチした行の地点および時刻とパターン変数 Z にマッチした行の地点および時刻をメジャーとして定義している．person_id は自動的にメジャーとして含まれる．出力するパターンオカレンステーブルの相関名は，図 1 の <result_table> に指定する．Example Query では result_table と指定している．

SQL/RPR では，パターンオカレンステーブル中の 1 行（パターンオカレンステーブル行）を出力するタイミングとして，パターンに完全に一致した時点で出力する ONE ROW PER MATCH と，パターンの一部にマッチする度に出力する ALL ROWS PER MATCH のいずれかを選択できる．Example Query では ONE ROW PER MATCH を指定している．

また，パターンに一致後，行パターンマッチングを再開する行を AFTER MATCH SKIP 句により指定できる．パターンに一致した最後の行の次の行から行パターンマッチングを再開する AFTER MATCH SKIP PAST LAST ROW や，最後に行パターンマッチングを開始した行の次の行から再開する AFTER MATCH SKIP TO NEXT ROW 等が指定可能である．Example Query では AFTER MATCH SKIP PAST LAST ROW を指定している．

MATCH_RECOGNIZE 句の処理の流れは次のようになる．まず，FROM 句で指定したテーブルが sequence_id と，order_key を元にシーケンスとして形成される．次に，PATTERN 句，DEFINE 句，SUBSET 句で指定した条件から DFA あるいは NFA が構築され，行パターンマッチングが実行され，MEASURES 句で定義したメジャーを属性

として持つパターンオカレンステーブル行が ONE ROW PER MATCH あるいは ALL ROWS PER MATCH のいずれかのタイミングで出力される．そして，パターン一致後，AFTER MATCH SKIP 句により指定した行から行パターンマッチングが再開される．

3.2 本研究で対象とする MATCH_RECOGNIZE 句の前提条件

本研究では，SQL/RPR の前提条件として下記の 2 点を置く．1 点目として，パターンオカレンステーブル行を出力するタイミングを制御するオプションとして ONE ROW PER MATCH のみを対象とし，ALL ROWS PER MATCH は対象外とする．

2 点目として，PATTERN 句で指定するパターンについて，そのパターンに一致するパターンオカレンスの最大行数が定まるものとする．たとえば，$PATTERN\ (X\ Y\ Z)$ のような 3 地点間を順に経由するシーケンシャルなパターンの場合は，X，Y，Z のそれぞれに 1 行ずつマッチするため，このパターンにマッチするパターンオカレンスの行数は 3 行に定まる．一方で，Example Query のように，パターンに * や + 等の繰返しを表す正規表現を用いる場合，行数は一意に定まらないが，実用上，無限に長いパターンオカレンスを取得したいという要求はほとんどないと考えられるため，本研究では，パターンオカレンスの最大行数がユーザによって指定されるものとする．最大行数の指定は関数 COUNT を用いて指定する．Example Query の場合，パターン変数 Y にマッチする最大行数は 2 行と指定している．そのため，パターン全体でマッチする最大行数は 4 行に定まる．また，$PATTERN\ (X+Y)+$ のように複数のパターン変数から構成されているような繰返しを表す正規表現の場合，SUBSET 句を用いてサブパターンを定義し，サブパターンに対して関数 COUNT を用いて最大行数を指定する．前述の例では，$SUBSET\ S\ =\ (X,Y)$ のようにサブパターン S を定義し，S に対して最大行数を指定する．

4. 行パターンマッチングの処理コスト

本章では，シーケンスデータに対する行パターンマッチングの処理コストの大きさを予備実験により示す．ここでは，Selection を行うクエリ（Selection Query），Join を行うクエリ（Join Query），行パターンマッチングを行うクエリ（MR Query）の 3 種類のクエリの処理時間をデータの行数を変えて比較する．**図 3** に 3 種類のクエリを示す．なお，行パターンマッチングを行うクエリの実行には，8 章において説明する SQL/RPR を実装した Spark を用いる．実験環境としては，5 ノードからなるクラスタマシンを用いる．クラスタ構成として，1 ノードをマスターノード，残りの 4 ノードをワーカーノードとする．また，分散ファイルシステムとして HDFS を用い，リソースマネージャとして YARN を用いる．なお，各ノードには，Ubuntu 14.04 LTS，AMD Opteron™Processor 2435 @2.60 GHz CPU，8 GB RAM の PC を使用する．

Selection Query

```
SELECT *
FROM test_table
WHERE test_table.c3 = 'A'
```

Join Query

```
SELECT *
FROM test_table
JOIN test_table2
  ON test_table.c2 < test_table2.c2
   AND test_table.c3 = test_table2.c3
```

MR Query

```
SELECT *
FROM test_table
MATCH_RECOGNIZE (
  PARTITION BY c1
  ORDER BY c2
  MEASURES X.c2 AS X_c2, Y.c2 AS Y_c2, Z.c2 AS Z_c2
  ONE ROW PER MATCH
  PATTERN ( X Y Z )
  DEFINE X AS X.c3 = 'A', Y AS Y.c3 = 'B', Z AS Z.c3 = 'C' )
```

図 3 処理コストの比較に用いる 3 種類のクエリ
Fig. 3 Three queries for cost comparison.

表 1 3 種類のクエリの処理時間
Table 1 Execution time of three queries.

n（行）	Selection Query (s)	Join Query (s)	MR Query (s)
10,000	10.39	13.89	34.39
100,000	9.97	30.34	41.99
1,000,000	13.05	81.53	79.50
10,000,000	15.31	285.12	434.13

予備実験に用いるデータは，c1，c2，c3 の 3 つの属性を持つテーブル（test_table）である．c1 は 1 から始まり 10,000 行ごとにインクリメントされる属性であり，sequence_id に対応する．c2 は 1 から 1 行ごとにインクリメントされ，10,001 行目で 1 に戻り，再びインクリメントされる属性であり，order_key に対応する．c3 はアルファベット 1 文字が入る文字列の属性であり，各行に固有の属性に対応する．

表 1 に，test_table の行数を変えたときの各クエリの処理時間を示す．値は 5 回の試行の平均値である．結果として，Selection Query は処理時間がほぼ一定であるのに対し，MR Query は行数が増えていくに従って処理時間が大きく増加している．この結果から，行パターンマッチングの処理コストを削減することは重要な課題であるといえる．

5. シーケンスデータに対する行パターンマッチングの効率化

本研究では，行パターンマッチングの処理の前に，入力テーブルに対してフィルタリングを行い，入力テーブルの行数を小さくすることで，行パターンマッチングにかかる処理コストを小さくする．入力データの行数を削減する最適化手法として，2 種類示す．1 つ目は対象シーケンス数を削減する Sequence Filtering，2 つ目は対象行数を削減する Row Filtering である．これらの最適化手法は文献 [11] の手法をさらに改良したものである．以降において，それぞれの最適化手法について説明する．

5.1 Sequence Filtering

sequence_id によって形成された各シーケンスに着目し，

```
SELECT *
FROM ( SELECT <sequence_id_list>
        FROM ( SELECT <sequence_id_list>,
                  MAX ( CASE WHEN <one_variable_cond> THEN 1 ELSE 0 END ) AS <flag>
                  ...
                  MAX ( CASE WHEN <one_variable_cond> THEN 1 ELSE 0 END ) AS <flag>
               FROM <table_name>
               WHERE <filt_cond_list>
               GROUP BY <sequence_id_list> ) AS <sequence_id_list_table>
       WHERE <flag> > 0 AND ... AND <flag> > 0 ) AS <tmp_table>
JOIN <table_name> ON <table_name>.<sequence_id_list> = <tmp_table>.<sequence_id_list> ;

<filt_cond_list> ::= <one_variable_cond_list>
<one_variable_cond_list> ::= <one_variable_cond_list> "OR" <one_variable_cond>
                           | <one_variable_cond>
```

図 4 Sequence Filtering Query の構文
Fig. 4 Syntax of Sequence Filtering Query.

```
SELECT *
FROM ( SELECT person_id
        FROM ( SELECT person_id,
                  MAX ( CASE WHEN location = 'A' THEN 1 ELSE 0 END ) AS flag1,
                  MAX ( CASE WHEN location = 'C' THEN 1 ELSE 0 END ) AS flag2
               FROM moving_table
               WHERE location = 'A' OR location = 'C'
               GROUP BY person_id ) AS moving_table2
       WHERE flag1 > 0 AND flag2 > 0 ) AS moving_table3
JOIN moving_table ON moving_table.person_id = moving_table3.person_id ;
```

図 5 Example Query に対応する Sequence Filtering Query
Fig. 5 Sequence Filtering Query corresponding to Example Query.

あるシーケンス内に DEFINE 句において指定されている one variable condition を満たす行が 1 行もない場合，そのシーケンス全体を行パターンマッチングの対象外とできる．これらのシーケンスをフィルタリングすることにより，処理コストが削減できる．Sequence Filtering の適用条件を以下に示す．

- DEFINE 句において，少なくとも 1 つのパターン変数の条件に one variable condition が存在する．

Sequence Filtering は *Sequence Filtering Query* によってなされる．Sequence Filtering Query は MATCH_RECOGNIZE 句を含む行パターンマッチングを行うクエリから one variable condition を抽出し，それをフィルタリングの条件としたクエリを構築することにより生成される．**図 4** に構文を示す．また，**図 5** に，Example Query に対応する Sequence Filtering Query を示す．Sequence Filtering では，まず，集約関数により，各シーケンスについて，各 one variable condition を満たす行が存在する場合に 1 を立てる属性 `<flag>` を付与する．そして，すべての `<flag>` が 1 であるシーケンスの sequence_id のテーブルを作成する．その後，`<table_name>` のテーブルと Join する．これにより，すべての one variable condition を満たす行が存在するシーケンス以外のシーケンスがフィルタリングされる．図 5 の Example Query の例では，まず，location の値として，‘‘A’’ と ‘‘C’’ のいずれも含むシーケンスの person_id のテーブルを作成する．その後，この person_id のテーブルと moving_table を person_id について Join することで，‘‘A’’ と ‘‘C’’ のいずれも含むシーケンス以外のシーケンスがフィルタリングされる．

Algorithm 1 Sequence Filtering Query の作成
Input: Original Query q
Output: Sequence Filtering Query sfq
1: $mr \leftarrow q$ 内の MATCH_RECOGNIZE specification
2: $clause[] \leftarrow mr$ を句ごとに分割
3: $OVCond[] \leftarrow clause[define]$ から one variable condition を抽出
4: $sfq \leftarrow OVCond[]$ を元に Sequence Filtering Query を作成
5: **return** sfq

MATCH_RECOGNIZE 句を含む行パターンマッチングを行うクエリから，Sequence Filtering Query を作成する流れを Algorithm 1 に示す．

- 1 行目において，クエリ q から MATCH_RECOGNIZE specification を抽出．
- 2 行目において，MATCH_RECOGNIZE specification 内の各句を分割．
- 3 行目において，DEFINE 句に指定した条件から one variable condition を抽出．
- 4 行目において，one variable condition と図 4 に示した構文を元に Sequence Filtering Query を作成．
- 5 行目において，sfq を返す．

5.2 Row Filtering

Sequence Filtering では，パターンオカレンスを構成する行が含まれる可能性のあるシーケンスを残し，そのほかのシーケンスを除去した．つまり，シーケンスがフィルタリングにおける単位であった．Row Filtering は，行パターンマッチングの結果としてパターンオカレンスを構成する行に含まれ得ない行を除去する．つまり，Row Filtering では行がフィルタリングにおける単位となる．Row Filtering は，3.2 節において説明した，指定したパターンからパターンオカレンスの最大行数が定まることを利用し，パターン変数に対する条件に合致する行に加え，パターンオカレンスの最大行数分，前後の行を残し，それ以外の行をフィルタリングする．

Row Filtering は，*Row Filtering Query* によってなされる．Row Filtering Query は MATCH_RECOGNIZE 句を含む行パターンマッチングを行うクエリ内の one variable condition と count condition を抽出し，それをフィルタリングの条件としたクエリを構築することにより生成される．**図 6** に，Row Filtering の構文を示す．また，**図 7** に，Example Query に対応する Row Filtering Query を示す．まず，パターンオカレンスの最大行数 n から 1 を引いた行数分，前後の行を対象とする Window により，`<table_name>` のテーブルに対して Window 関数を適用する．Window 関数では，Window 内に `<filt_cond_list>` を満たす行が存在する場合，フラグを立てる処理を行う．図 7 では，$n = 4$

であるため，対象行に加え前後 3 行分の Window の中に，location の値として ‘‘A’’ あるいは ‘‘C’’ を含む行が存在する場合，フラグを立てる．このようにフラグを立てたテーブルを作成し，最後に，このテーブルに対してフラグが立っていない行をフィルタリングする．これにより，パターンオカレンスに含まれ得ない行がフィルタリングされる．

Row Filtering Query の作成の流れを Algorithm 2 に示す．Sequence Filtering Query の作成の流れである Algorithm 1 とほぼ同じであるが，Row Filtering では，4 行目において，DEFINE 句に指定した条件から count condition を抽出し，Row Filtering Query における Window 関数の Window 幅の指定に利用する．

以上から，効率化手法の適用の種類としては，(1) 効率化を行わない（No Filtering），(2) Sequence Filtering を行う，(3) Row Filtering を行う，(4) Sequence Filtering の後，Row Filtering を行う（Sequence Filtering + Row Filtering），の 4 種類の手法が考えられる．(4) を行う理由としては，全入力テーブルに対し Row Filtering を行うよりも，1 度 Sequence Filtering により行を削減した後に Row Filtering を行う方が，Sequence Filtering のオーバヘッド以上に処理時間が削減されることが期待されるためである．

```
SELECT *
FROM ( SELECT <sequence_id_list>, <order_key_list>, <column_name_list>,
              MAX ( CASE WHEN <filt_cond_list> THEN 1 ELSE 0 END )
                 OVER ( PARTITION BY <sequence_id_list>
                        ORDER BY <order_key_list>
                        ROWS BETWEEN (n – 1) PRECEDING AND (n – 1) FOLLOWING ) AS <flag>
       FROM <table_name>
     ) AS <flag_table>
WHERE <flag> > 0 ;
```

```
<column_name_list> ::= <column_name> { “,” <column_name_list> }
```

図 6 Row Filtering Query の構文
Fig. 6 Syntax of Row Filtering Query.

```
SELECT *
FROM ( SELECT person_id, time, location,
              MAX( CASE WHEN location = ‘A’ OR location = ‘C’
                     THEN 1 ELSE 0 END )
                 OVER ( PARTITION BY person_id
                        ORDER BY time
                        ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING ) AS flag
       FROM moving_table ) AS flag_table
WHERE flag > 0 ;
```

図 7 Example Query に対応する Row Filtering Query
Fig. 7 Row Filtering Query corresponding to Example Query.

Algorithm 2 Row Filtering Query の作成
Input: Original Query q
Output: Row Filtering Query rfq
1: $mr \leftarrow q$ 内の MATCH_RECOGNIZE specification
2: $clause[] \leftarrow mr$ を句ごとに分割
3: $OVCond[] \leftarrow clause[define]$ から one variable condition を抽出
4: $CCond[] \leftarrow clause[define]$ から count condition を抽出
5: $rfq \leftarrow OVCond[]$ と $CCond[]$ を元に Row Filtering Query を作成
6: **return** rfq

6. 各効率化手法のコストモデル

6.1 RDBMS を対象としたコストモデル

前章の最後に示した 4 種類の効率化手法について，それぞれの処理時間を見積もるコストモデルを構築する．まずは RDBMS における処理を対象としたコストモデルを構築する．このコストモデルにより各効率化手法を適用した場合の処理時間を見積もることが可能となれば，最も処理時間が短くなる適切な効率化手法を選択することが可能となる．**表 2** に記号の定義を行う．

(1) No Filtering

N 行のテーブルに対して行パターンマッチングを行うため，処理コストは 1 行あたりの行パターンマッチングのコスト r に N 行を乗じたものとなる．そのため，処理コストは以下となる．

$$rN \tag{1}$$

なお，上記のコストは行パターンマッチング対象となるテーブルがメモリに乗り切る場合の CPU コストを示している．行パターンマッチング対象となるテーブルがメモリに乗り切らない場合は I/O コストを追加する必要がある．行パターンマッチングではパーティショニング（集約処理に相当），ソート，オートマトンでの処理において I/O が発生する．パーティショニングとソートでは，テーブルを sequence_id，order_key でソートする．よってこの処理の I/O ページ数は，$2M \log_2 M$ となる．オートマトンでの処理はソートされたテーブルを先頭から処理するため，I/O ページ数は M となる．よって，I/O コストを含めた No Filtering における全体の処理コストは以下となる．

$$rN + m(2M \log_2 M + M) \tag{2}$$

(2) Sequence Filtering

Sequence Filtering では，主に，サブクエリ内の，WHERE

表 2 RDBMS を対象としたコストモデルにおいて用いる記号の定義
Table 2 Definition of symbols in cost model on RDBMS.

記号	意味
N	テーブルの行数
S	テーブルに含まれるシーケンス数
M	テーブルのページ数
α	Sequence Filtering によって残るシーケンスの割合
β	Sequence Filtering によって残ったシーケンスの中で，さらに Row Filtering によって残る行の割合
γ	Row Filtering によって残る行の割合
δ	Row Filtering によって残るシーケンスの割合
c	シーケンシャルスキャンにかかる 1 行あたりのコスト
r	行パターンマッチングにかかる 1 行あたりのコスト
w	Window 関数にかかる 1 行あたりのコスト
m	1 ページあたりの I/O コスト

句における <filt_cond_list> を満たす行の探索コスト，GROUP BY 句および集約関数 MAX におけるコスト，<sequence_id_list_table> のテーブルにおけるすべての <flag> が 1 となっている sequence_id を探索するためのスキャンコスト，sequence_id のテーブルと <table_name> のテーブルの Join コストがかかる．<table_name> テーブルにはインデックスが張られているものとし，最初の探索コストは無視できるほど小さいとする．GROUP BY 句における Aggregation には Hash Aggregate を用いると仮定する．ハッシュ表の作成コストは，その作成対象のテーブルのスキャンコストとほぼ等しい．このテーブルの行数は，パターン変数に対する条件に合致する行の数であるため γN に等しい．集約関数 MAX の計算はハッシュ表の作成におけるテーブルのスキャンと同時になされるとする．そのため，GROUP BY 句および集約関数 MAX のコストは $c\gamma N$ となる．すべての <flag> が 1 となっている sequence_id を探索するための対象行数は，パターン変数に対する条件に合致する行を含むシーケンス数であるため δS とほぼ等しい．そのため，すべての <flag> が 1 となっている sequence_id を探索するためのスキャンコストは $c\delta S$ となる．Join には Hash Join を用いると仮定する．<tmp_table> の行数は αS であるため，ハッシュ表の作成コストは $c\alpha S$ となる．<table_name> テーブルの行数は N であるため，Join のコストは cN となる．行パターンマッチングの対象行数は αN となるため，行パターンマッチングにかかるコストは $r\alpha N$ となる．以上から，処理コストは以下となる．

$$c\gamma N + c\delta S + c\alpha S + cN + r\alpha N \tag{3}$$

I/O コストについては，まず WHERE 句において <filt_cond_list> を満たす行の探索するために M ページ読み込む．ここで，Selection，Aggregation や最適化手法を適用した結果の中間テーブルはメモリに乗り切る前提とする．そのため，その後の <tmp_table> を作成するサブクエリ内の処理はメモリ上で行われる．<table_name> テーブル（行パターンマッチング対象となる元テーブル）と <tmp_table> の Join では，<table_name> テーブルの M ページの読み込みが I/O コストとしてかかる．Sequence Filtering 後のテーブルはメモリ上に乗り切るため，行パターンマッチングの処理はメモリ上で行われる．なお，中間テーブルがメモリ上に乗り切らない場合も考えられるが，その場合はさらに追加の I/O コストがかかる．この場合の I/O コストは各処理の読み込みと書き込みにかかるページ数を考慮することで見積もることができるため，ここでは省略する．以上から，I/O コストを含めた Sequence Filtering における全体の処理コストは以下となる．

$$c\gamma N + c\delta S + c\alpha s + cN + r\alpha N + 2mM \tag{4}$$

(3) Row Filtering

Row Filtering では，主に，サブクエリ内における，Window 関数にかかるコストと，Window 関数の結果としてフラグが付与されたテーブルに対し，フラグが立っていない行をフィルタリングするスキャンコストが存在する．Window 関数にかかるコストは wN となる．また，フラグが付与されたテーブルのスキャンコストは cN となる．よって，これらのコストの和は $(w+c)N$ となる．行パターンマッチングの対象行数は γN となるため，行パターンマッチングにかかるコストは $r\gamma N$ となる．以上から，処理コストは以下となる．

$$(w + c)N + r\gamma N \tag{5}$$

I/O コストについて，Row Filtering では Window 関数のためにパーティショニング，ソートを行うため，No Filtering における行パターンマッチングの処理と同じ I/O コストがかかる．Row Filtering 後のテーブルはメモリに乗り切るとする．以上から，I/O コストを含めた Row Filtering における全体の処理コストは以下となる．

$$(w + c)N + r\gamma N + m(2M \log_2 M + M) \tag{6}$$

(4) Sequence Filtering + Row Filtering

Sequence Filtering を行う処理コストは，上記に示した Sequence Filtering の処理コストと同一である．Row Filtering における，Window 関数の対象行数は αN となるため，Row Filtering を行う処理コストは $(w+c)\alpha N$ となる．以上から，処理コストは以下となる．

$$c\gamma N + c\delta S + c\alpha S + cN + (w + c)\alpha N + r\alpha\beta N \tag{7}$$

I/O コストについて，まず Sequence Filtering を行うため，Sequence Filtering 単独の場合と同じ I/O コストがかかる．フィルタリングされたテーブルはメモリ上に乗り切るとし，以降の処理はメモリ上で行われる．以上から，I/O コストを含めた Sequence Filtering + Row Filtering における全体の処理コストは以下となる．

$$c\gamma N + c\delta S + c\alpha S + cN + (w + c)\alpha N + r\alpha\beta N + 2mM \tag{8}$$

上記に示した処理コストの妥当性を，9.3 節において検証する．

6.2 Spark を対象としたコストモデル

Spark SQL を対象としたコストモデルは，Lorenzo らが I/O コストとノード間のデータ転送のコストを主要コストとしたコストモデルを提案している [3]．そのため，各最適化手法のコストモデルには，これらのコストを含める必要がある．本研究では，文献 [3] のコストに行パターンマッチングの CPU コストを追加するように拡張した，ある処

表 3 Spark を対象としたコストモデルにおいて用いる記号の定義
Table 3 Definition of symbols in cost model on Spark.

記号	意味
T	テーブルサイズ（MB）
d_r^L	単位時間あたりのローカルノードのディスクから読み込み可能なデータサイズ（MB/s）
d_r^R	単位時間あたりのリモートノードのディスクから読み込み可能なデータサイズ（MB/s）
d_r^S	シャッフリングのために複数ノードで並列にディスクから読み込む際の単位時間あたりの読み込み可能なデータサイズ（MB/s）
d_w	単位時間あたりのディスクへの書き込み可能なデータサイズ（MB/s）
ρ	単位時間あたりにノード間で転送可能なデータサイズ（MB/s）
s	単位時間あたりに行パターンマッチング可能なデータサイズ（MB/s）

理を複数ノードで並列に実行したときにかかる全体の処理時間（秒）を推定するコストモデルを示す．表 2 の定義に加え，**表 3** に記号の定義を行う．なお，ここではすべてのノードは同一ラック内に存在するとし，ノード数，各ノードのコア数，1 コアあたりのプロセス数，HDFS のレプリカ数は固定とし，1 つのコアが 1 つの HDD partition を処理すると仮定する．

文献 [3] では，Selection や Join や GROUP BY といった基本演算にかかるコストモデルを示しているが，ここではその中で，最適化手法のコストモデルに必要な 3 つの演算のコストを示す．1 つ目は，Selection のようなデータの単純なスキャンコストである．ここではこのコストは，ディスクにあるデータの読み込みのコスト，スキャン後のデータのシャッフルのためのバケットの書き込みのコストを含む．スキャンコストでは，実際にデータ処理を行う executor ノードのローカルノードのディスクに処理対象のデータがある場合のデータの読み込みのコストと，ローカルノードのディスクに処理対象のデータがなく，リモートノードのディスクからデータを読み込むコストの期待値とする．ローカルノードのディスクに処理対象のデータがある場合，ディスクからの読み込みとバケットへの書き込みはパイプライン処理であるため，コストはそれらの最大値となり，$MAX(T/d_r^L, T/d_w)$ となる．リモートノードからデータを読み込む場合も，ディスクからの読み込み，ノード間の転送，バケットへの書き込みはパイプライン処理であるため，コストはそれらの最大値となり，$MAX(T/d_r^R, T/\rho, T/d_w)$ となる．ローカルノードのディスクに処理対象のデータがある確率を P_L，ローカルのディスクに処理対象のデータがなく，リモートノードからデータを読み込む確率を $(1-P_L)$ とすると，テーブルのスキャンコストは以下となる．このコストを $SC()$ とおく．

$$\begin{aligned} &SC(T, d_r^L, d_r^R, d_w, \rho) \\ &= P_L \cdot MAX\left(\frac{T}{d_r^L}, \frac{T}{d_w}\right) \\ &\quad + (1-P_L) \cdot MAX\left(\frac{T}{d_r^R}, \frac{T}{\rho}, \frac{T}{d_w}\right) \end{aligned} \tag{9}$$

2 つ目は Join のコストである．ここでは Join のコストは，バケットを読み込み，Join のキーとなる属性値が同じものを同じノードへ転送するシャッフル処理をし，Join の後，再度データのシャッフルのためにバケットを書き込むコストを含む．読み込みのコストは，T_1，T_2 を Join するテーブルサイズとすると $(T_1+T_2)/d_r^S$，転送のコストは $(T_1+T_2)/\rho$ となる．また，両処理はパイプライン処理であるため，コストはそれらの最大値となり，$MAX((T_1+T_2)/d_r^S, (T_1+T_2)/\rho)$ となる．Join 自体の CPU コストは無視できるほど小さいとする．以上から，Join のコストは以下となる．このコストを $SJ()$ とおく．なお，jT は Join の結果のテーブルサイズである．

$$\begin{aligned} &SJ(T_1, T_2, jT, d_r^S, d_w, \rho) \\ &= MAX\left(\frac{T_1+T_2}{d_r^S}, \frac{T_1+T_2}{\rho}\right) + \frac{jT}{d_w} \end{aligned} \tag{10}$$

3 つ目は GROUP BY のコストである．ここでは GROUP BY のコストは，シャッフル処理と，GROUP BY 後，再度データのシャッフルのためにバケットを書き込むコストを含む．GROUP BY 自体の CPU コストは無視できるほど小さいとする．以上から，GROUP BY のコストは以下となる．このコストを $GB()$ とおく．なお，gT は GROUP BY の結果のテーブルサイズである．

$$\begin{aligned} &GB(T, gT, d_r^S, d_w, \rho) \\ &= MAX\left(\frac{T}{d_r^S}, \frac{T}{\rho}\right) + \frac{gT}{d_w} \end{aligned} \tag{11}$$

本研究ではこれらに加え，行パターンマッチングのコストを追加する．行パターンマッチングのコストは，行パターンマッチング処理自体の CPU コストと結果を書き込むコストを含む．以上から，行パターンマッチングのコストは以下となる．このコストを $RPR()$ とおく．なお，pT は行パターンマッチングの結果のテーブルサイズである．

$$RPR(T, pT, d_w, s) = \frac{T}{s} + \frac{pT}{d_w} \tag{12}$$

以上から，各最適化手法のコストを以下に示す．

(1) No Filtering

行パターンマッチングの処理では，データの読み込み，sequence_id をキーとしたシャッフル，行パターンマッチングの処理，結果の書き込みが含まれる．データの読み込みはデータのスキャンコスト $SC()$ と等価，シャッフルのコストは GROUP BY のコスト $GB()$ と等価と考え，処理コストは以下となる．

$$SC(T, d_r^L, d_r^R, d_w, \rho) + GB(T, T, d_r^S, d_w, \rho)$$

$$+ RPR(T, \alpha\beta T, d_w, s) \quad (13)$$

(2) Sequence Filtering

Sequence Filtering では，データの読み込み，サブクエリの GROUP BY の処理におけるシャッフル，Join の処理におけるシャッフル，行パターンマッチングのためのシャッフル，行パターンマッチングのコストが含まれる．GROUP BY の処理の対象となるデータサイズは γT，Join における <tmp_table> のサイズは $(\delta S/N)T$，行パターンマッチング対象となるテーブルのサイズは αT となるため，処理コストは以下となる．

$$SC(T, d_r^L, d_r^R, d_w, \rho) + GB\left(\gamma T, \frac{\delta S}{N} T, d_r^S, d_w, \rho\right) + SJ\left(T, \frac{\delta S}{N} T, \alpha T, d_r^S, d_w, \rho\right) + GB(\alpha T, \alpha T, d_r^S, d_w, \rho) + RPR(\alpha T, \alpha\beta T, d_w, s) \quad (14)$$

(3) Row Filtering

Row Filtering では，データの読み込み，Window 関数のためのシャッフル，行パターンマッチングのためのシャッフル，行パターンマッチングのコストが含まれる．Window 関数後にフラグが付与されたテーブルに対し，フラグが立っていない行をフィルタリングすることでデータサイズが γT となるため，処理コストは以下となる．

$$SC(T, d_r^L, d_r^R, d_w, \rho) + GB(T, \gamma T, d_r^S, d_w, \rho) + GB(\gamma T, \gamma T, d_r^S, d_w, \rho) + RPR(\gamma T, \alpha\beta T, d_w, s) \quad (15)$$

(4) Sequence Filtering + Row Filtering

最初の Sequence Filtering については，データの読み込み，サブクエリの GROUP BY の処理におけるシャッフル，Join の処理におけるシャッフルが含まれる．その次の Row Filtering については，Window 関数のためのシャッフルが含まれる．そしてその後，行パターンマッチングのためのシャッフルと行パターンマッチングのコストが含まれる．Row Filtering における Window 関数の対象となるデータサイズが αT，行パターンマッチング対象となるデータサイズが $\alpha\beta T$ であるため，処理コストは以下となる．

$$SC(T, d_r^L, d_r^R, d_w, \rho) + GB\left(\gamma T, \frac{\delta S}{N} T, d_r^S, d_w, \rho\right) + SJ\left(T, \frac{\delta S}{N} T, \alpha, d_r^S, d_w, \rho\right) + GB(\alpha T, \alpha\beta T, d_r^S, d_w, \rho) + GB(\alpha\beta T, \alpha\beta T, d_r^S, d_w, \rho) + RPR(\alpha\beta T, \alpha\beta T, d_w, s) \quad (16)$$

7. PostgreSQL における行パターンマッチング

PostgreSQL 上の SQL/RPR の実装は，那須らの実装 [12] をベースとし，UDF によっている．本研究では，行パターンマッチングを行う関数 match_recognize() を C 言語を用いて作成している．関数 match_recognize() は，入力テーブルを受け取り，行パターンマッチングを行い，その結果のパターンオカレンステーブルを返す関数である．関数 match_recognize() の引数と戻り値を図 8 に示す．AS 句以前のカッコ内が引数，AS 句以降のカッコ内が戻り値である．関数 match_recognize() は，4 つの引数（入力テーブルを取得するための SQL 文，メジャー，パターン，パターン変数に対する条件）と複数の戻り値（パターンオカレンステーブルの各属性と型）を有する．引数に指定した情報から NFA を構築し，行パターンマッチングを実行する．NFA はパターン変数が各状態に対応し，正規表現により状態間を遷移するエッジが定まる．状態間を遷移する条件はパターン変数に対する条件が対応する．また，各状態に，マッチした行を保持するバッファが存在する．受理状態まで達したとき，メジャーとして指定した属性の値を各バッファから参照し，パターンオカレンステーブル行を生成する．この NFA の構築方法は 8 章における Spark に対する行パターンマッチングの実装においても同様である．なお，一般にパターンマッチングでは NFA よりも DFA の方が高速に処理できるが，すでに行パターンマッチングを実装している RDBMS では，シーケンシャルなパターンのような，受理状態まで次に遷移する状態が一意に定まるパターン以外，すべて NFA で実装されている [9]．これは，NFA の方が DFA よりもメモリ消費量が少なくなるためである．本研究ではこれに準じ，NFA で実装を行う．DFA での行パターンマッチングの実装は今後の課題とする．

```
SELECT *
FROM match_recognize (
  ' SELECT row_number() OVER (ORDER BY <sequence_id>, <order_key>) AS rid,
          <sequence_id>, <order_key>, <column_name_list>
   FROM <table_name> ' ,
  ' <measure_list> ' ,
  ' ( <regexp> ) ' ,
  ' <corrname_definition_list> ' )
  AS ( <sequence_id> <type>, <measure_and_type_list> ) ;

<measure_and_type_list> ::= <measure> <type> { "," < measure_and_type_list > }
```

図 8 関数 match_recognize() の引数と戻り値
Fig. 8 Arguments and return value of function match_recognize().

本研究における効率化手法を適用する場合は，図 8 に示した関数 match_recognize() を含むクエリの前に VIEW を用いて Filtering Query を入力テーブルに対して適用するように書き換える．本研究では，8 章において説明する Spark も含め，上記に述べた Filtering Query の生成手順にのっとり，Filtering Query を手動で作成している．なお，効率化のコストには，Filtering Query の実行にかかるコストに加え，Filtering Query の作成処理も含まれるが，この処理は，MATCH_RECOGNIZE 句を含むクエリをパースし，DEFINE 句に指定した one variable condition,

count condition を抽出し，Filtering Query の文字列を作成する処理である．そのため，この処理自体の処理コストは Filtering Query の実行にかかるコストと比較し小さいと考え，手動作成による Filtering Query であっても本研究において提案する手法の効果はあると考える．

8. Spark における行パターンマッチング

8.1 SQL/RPR を実装する Spark のアーキテクチャ

Spark 上で SQL ライクなクエリを用いてデータ分析を行うコードを記述するモジュールとして Spark SQL [1] がある．ここで，MATCH_RECOGNIZE 句を含んだ Spark SQL のクエリを実行できるシステムアーキテクチャを考える．このアプローチには，**図 9** (a) に示すような，現状の Spark SQL の内部のコードを SQL/RPR が実行できるように直接書き換える方法（*Direct Extension Approach*）と，図 9 (b) に示すような，ユーザが記述した Spark のコードを解析し，MATCH_RECOGNIZE 句を含むクエリのコードを通常の Spark でも処理が可能なように書き換える，フロントエンドのモジュールを用意する方法（*Front-end Approach*）の 2 種類の方法が考えられる．

本研究では，現状の Spark SQL の内部コードを変更することなく容易に実装が可能な，(b) の Front-end Approach で実装する．なお，本研究において提案する行パターンマッチングの効率化手法は，いずれのアプローチにも適用可能なものである．また，本研究では SQL/RPR を Java 言語により実装している．

8.2 MATCH_RECOGNIZE 句を含む Spark SQL のクエリ書き換え

フロントエンドモジュールによる，Spark のコードの書き換えの流れを説明する．Spark ではデータを *dataframe* と呼ばれるコレクションの形で保持し，ユーザはこの dataframe に対する処理を記述する．本研究では前提として，この dataframe を Spark の内部関数 createOrReplaceTempView() によりクエリ内で用いるテーブル名を登録し，クエリ文字列を変数 *query* に格納し，*query* を Spark の内部関数 sql() により実行する，3 つの処理が含まれたコードを対象に書き換えを行う．これらのコードを以下に示す．

- <入力 `dataframe` 名>.createOrReplaceTempView(‘‘<入力テーブル名>’’)
- query = ‘‘<`MATCH_RECOGNIZE` 句を含むクエリ文字列>’’
- <結果 `dataframe` 名> = spark.sql(query)

上記の 3 つの処理は，コード内の同一クラス，メソッドに含まれているものとする．また，*query* 内の FROM 句には関数 createOrReplaceTempView() により登録した入力テーブル名を指定するものとする．また，*query* には変数がバインドされておらず，固定の文字列が指定されているものとする．なお，ここでは説明の簡単化のため，クエリ文字列を変数 *query* に格納し，*query* を関数 sql() により実行しているが，関数 sql() の引数として直接クエリ文字列を指定する場合にも対応は可能である．

書き換えの流れを Algorithm 3 に示す．この Algorithm を実装した変換器を作成することで，Spark 上で MATCH_RECOGNIZE 句を含むクエリの処理が実行可能なコードに変換できる．

- 1 行目から 8 行目にかけ，フロントエンドモジュールが，Spark の Application Code をスキャン．
- 2 行目において，MATCH_RECOGNIZE 句を含むクエリを見つけた場合，そのクエリを変数 q に格納．
- 3 行目において，q から MATCH_RECOGNIZE specification を抽出．
- 4 行目において，効率化を行う場合は Algorithm 1 あるいは 2 により Filtering Query を作成し，*code* に追加．効率化を行わない場合は *code* には何も追加しない．
- 5 行目において，MATCH_RECOGNIZE specification に対応する行パターンマッチングを行う NFA を構築し，行パターンマッチングを実行するコードを *code* に追加．
- 6 行目において，q から MATCH_RECOGNIZE specification を除いたクエリを *code* に追加．

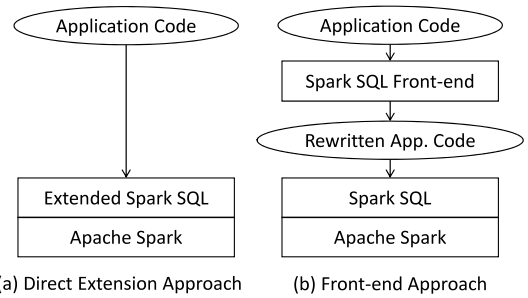


図 9 MATCH_RECOGNIZE 句を実装するアーキテクチャ
Fig. 9 Architecture for implementation of MATCH_RECOGNIZE Clause.

Algorithm 3 コードの書き換えの流れ

Input: Application Code c
Output: Rewritten App. Code c
1: **while** *scan* c **do**
2: $q \leftarrow$MATCH_RECOGNIZE 句を含むクエリ文字列
3: $mr \leftarrow q$ 内の MATCH_RECOGNIZE specification
4: $code \leftarrow mr$ を元に作成された Filtering Query
5: $code \leftarrow code + mr$ を元に NFA を構築し行パターンマッチングを行うコード
6: $code \leftarrow code + q$ から MATCH_RECOGNIZE specification を除いたクエリ
7: $code$ を c における抽出したクエリの部分に挿入
8: **end while**
9: **return** c

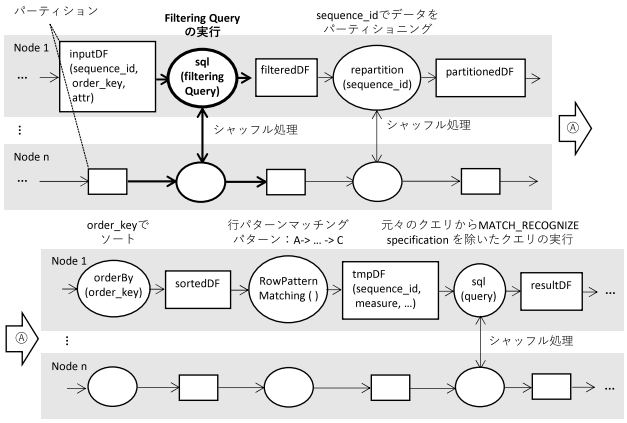


図 10 Spark 実行時のデータフロー
Fig. 10 Dataflow of executing by Spark.

- 7 行目において，生成したコードを元コードの MATCH_RECOGNIZE 句を含むクエリの部分に挿入．

8.3 MATCH_RECOGNIZE 句の処理の Spark 実行時におけるデータフロー

図 **10** に，MATCH_RECOGNIZE 句を含むクエリを Spark 上で実行する際のデータフローを示す．dataframe は各ノード上にパーティション単位で分割され配置されている．ユーザが特に指定をしない限り，Spark がデータを等サイズのパーティションに分割する．図 10 では，inputDF という dataframe に対し，行パターンマッチングを行い，最終的に resultDF という dataframe を作成する．MATCH_RECOGNIZE 句を含むクエリの実行においては，行パターンマッチングの処理と MATCH_RECOGNIZE 句を除いた部分のクエリの処理からなる．まず，効率化の処理を行う場合は行パターンマッチングの前に Filtering Query を実行する（太字で示している）．次に，行パターンマッチングの処理を行う．まず，sequence_id でパーティショニングする．この際，ノード間でシャッフル処理が行われる．次に，order_key によってソートする．そして，ソートされたデータに対し，行パターンマッチングを行う．行パターンマッチングの実行は，本研究において独自実装する関数 RowPatternMatching() により行う．最後に，MATCH_RECOGNIZE specification を除いたクエリを実行し，その結果として resultDF を得る．

9. 評価実験

9.1 人工データを用いた実験

本実験では，5.2 節の最後に示した 4 通りの効率化について，人工データを用いて検証する．本実験では，Q1 から Q7 の 7 種類のクエリを用いる．これらのクエリを，図 **11**，図 **12**，図 **13**，図 **14**，図 **15**，図 **16**，図 **17** に示す．なお，図 12 以降は PATTERN 句と DEFINE 句のみ示す．Q1 はシーケンシャルなパターン，Q2 は繰返しを表す正規表現を含むパターン，Q3 および Q4 は選択を表す正規表現を含むパターン，Q5 および Q6 は繰返しや選択を表す正規表現が複合的に含まれているパターン，Q7 は繰返し

```
SELECT *
FROM test_table
MATCH_RECOGNIZE (
  PARTITION BY c1
  ORDER BY c2
  MEASURES X.c2 AS X_c2, Y.c2 AS Y_c2, Z.c2 AS Z_c2
  ONE ROW PER MATCH
  PATTERN ( X Y Z )
  DEFINE X AS X.c3 = 'A', Z AS Z.c3 = 'C' ) ;
```

図 11 問合せ Q1
Fig. 11 Query Q1.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( X Y* Z )
  DEFINE X AS X.c3 = 'A',
         Y AS COUNT(Y.*) <= 2,
         Z AS Z.c3 = 'C' ) ;
```

図 12 問合せ Q2
Fig. 12 Query Q2.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( ( X | Y ) Z )
  DEFINE X AS X.c3 = 'A', Y AS Y.c3 = 'B',
         Z AS Z.c3 = 'C' ) ;
```

図 13 問合せ Q3
Fig. 13 Query Q3.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( ( X | Y ) ( Z | W ) )
  DEFINE X AS X.c3 = 'A', Y AS Y.c3 = 'B',
         Z AS Z.c3 = 'C', W AS W.c3 = 'D' ) ;
```

図 14 問合せ Q4
Fig. 14 Query Q4.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( X+ ( Y | Z ) W )
  DEFINE X AS COUNT(X.*) <= 3, Y AS Y.c3 = 'B',
         Z AS Z.c3 = 'C', W AS W.c3 = 'D' ) ;
```

図 15 問合せ Q5
Fig. 15 Query Q5.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( X ( Y | Z W )* )
  DEFINE X AS X.c3 = 'A',
         Y AS Y.c3 = 'B' AND COUNT(Y.*) <= 2,
         Z AS Z.c3 = 'C' AND COUNT(Z.*) <= 2,
         W AS COUNT(W.*) <= 2 ) ;
```

図 16 問合せ Q6
Fig. 16 Query Q6.

```
...
MATCH_RECOGNIZE (
  ...
  PATTERN ( X Y* Z )
  DEFINE X AS X.c3 = 'A',
         Y AS COUNT(Y.*) <= 98,
         Z AS Z.c3 = 'C' ) ;
```

図 17 問合せ Q7
Fig. 17 Query Q7.

を表す正規表現を含み，結果のパターンオカレンスが長くなるパターンである．

本実験において入力テーブルとして用いる人工データの属性，行数，シーケンス数は，4 章で用いたものと同じである．本実験で用いる人工データは 3 種類のシーケンスを含む．1 つ目はパターンオカレンスが含まれるシーケンス，2 つ目はパターンオカレンスは含まれないが，one variable condition の一部の条件を満たす行が含まれるシーケンス，3 つ目は one variable condition の条件を満たす行が含まれないシーケンスである．このうえで人工データの生成手順を以下に示す．

(1) S 個のシーケンスのうち，$\alpha \times S$ 個のシーケンスをパターンオカレンスが含まれるシーケンス，$(\delta - \alpha) \times S$ 個のシーケンスをパターンオカレンスは含まれないが，one variable condition の一部の条件を満たす行が含まれるシーケンス，$(1 - \delta) \times S$ 個のシーケンスを，one variable condition の条件を満たす行が含まれないシーケンスとする．

(2) パターンオカレンスが含まれる $\alpha \times S$ 個のシーケンスについて，その各シーケンス内の N/S 行のうち，$\beta \times N/S$ 行にあらかじめ用意した複数種類のパターンオカレンスを挿入する．具体的には Q1 の場合，パターンオカレンスとなる {A, B, C} の行の繰返しで埋める．Q2 の場合，パターンオカレンスとして {A, C}，{A, B, C}，{A, B, B, C} が考えられるため，これらを繰り返し挿入する．$\beta \times N/S$ 行は固定値であるため，最後のパターンオカレンスについては $\beta \times N/S$ 行が埋まった時点で挿入をやめる．残りの $(1 - \beta) \times N/S$ 行は，one variable condition の条件を満たさない行で埋める．具体的には，{E, F, G} の行の繰返しで埋め，$(1 - \beta) \times N/S$ 行が埋まった時点で終了する．設定した行数で挿入を取りやめることについては，以降の (3)，(4) の手順においても同様であり，Q1 から Q7 のすべてのクエリで共通である．

(3) パターンオカレンスは含まれないが，one variable condition の一部の条件を満たす行が含まれる $(\delta - \alpha) \times S$ 個のシーケンスについても (2) の手順と同様に，$\beta \times N/S$ 行に one variable condition の一部の条件を満たす行で埋める．たとえば Q1 の場合，one variable condition として X.c3=''A''，Z.c3=''C'' の 2 つがあるが，このうちの 1 つの条件だけを満たす行で埋める．具体的には，{A, B, A} の繰返しで埋める．残りの $(1 - \beta) \times N/S$ 行は，one variable condition の条件を満たさない行で埋める．具体的には，{E, F, G} の行の繰返しで埋める．

(4) one variable condition の条件を満たす行が含まれない $(1 - \delta) \times S$ 個のシーケンスについては，その各シーケンス内の全行を one variable condition の条件を満たさない行で埋める．具体的には，{E, F, G} の行の繰返しで埋める．

(5) 最後に，(2)，(3)，(4) で作成したそれぞれのシーケンスをマージする．

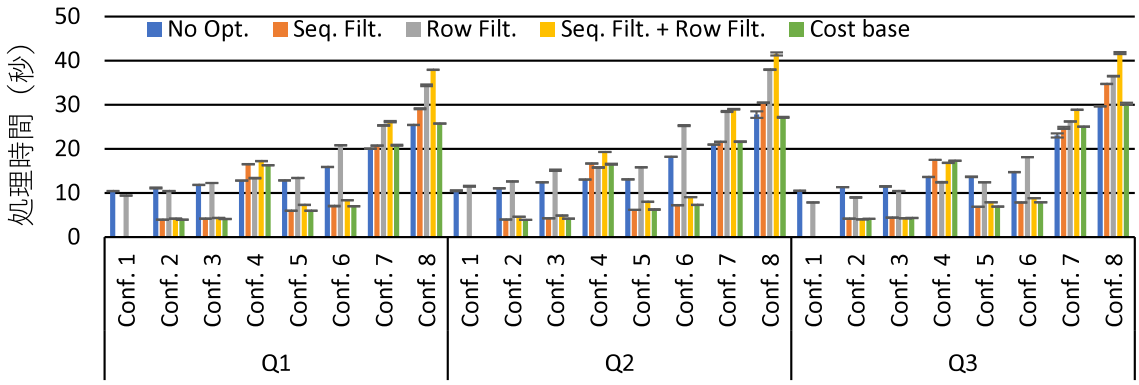
表 4 Q1 から Q7 における α，β，γ，δ の値
Table 4 Values of α, β, γ, and δ from Q1 to Q7.

	Configuration							
	1	2	3	4	5	6	7	8
α	0.00	0.20	0.20	0.80	0.20	0.20	0.80	1.00
β	0.00	0.20	0.20	0.20	0.80	0.80	0.80	1.00
γ	0.00	0.04	0.16	0.16	0.16	0.64	0.64	1.00
δ	0.00	0.20	0.80	0.80	0.20	0.80	0.80	1.00

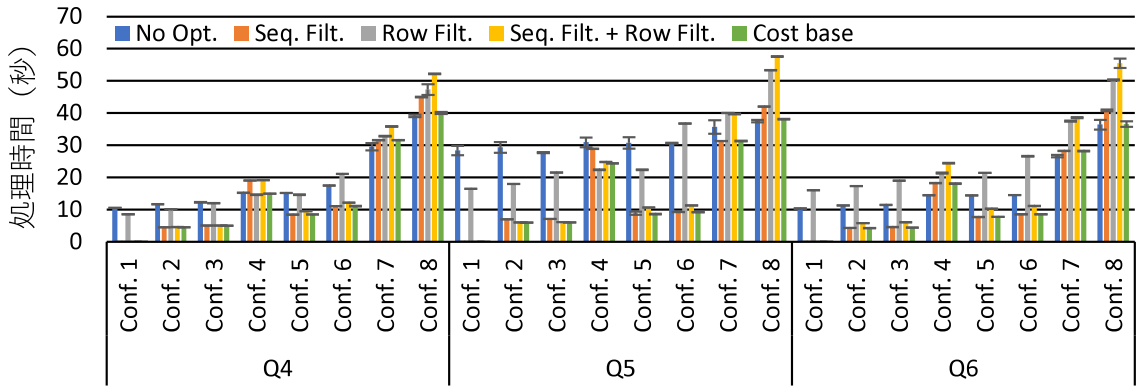
本実験では，α，β，γ，δ の値のパラメータの組合せとして，**表 4** に示す 8 種類を用意する．

本実験では，PostgreSQL においては，データはすべてメモリ上に乗っている状態から処理を行う．そのため，本章では 6.1 節で示したコストモデルについては CPU コストのみを考慮する．なお，PostgreSQL の実験環境として，Ubuntu 16.04 LTS，Intel®Core™i9-7920X CPU @2.90 GHz，128 GB RAM の PC を使用する．Spark については，4 章で示した実験環境と同一である．

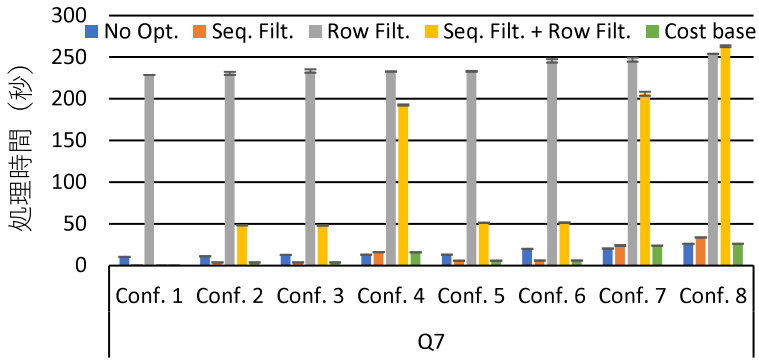
まず，PostgreSQL を用いた人工データに対する実験結果を**図 18** に示す．No Filt.，Seq. Filt.，Row Filt.，Seq. Filt. + Row Filt. は，それぞれの最適化手法を適用した場合の処理時間である．Cost Base は，コストモデルにより各最適化手法の処理時間を計算し，最短となる最適化手法を選択する処理を含んだ場合の処理時間である．実験結果はいずれも，処理時間は 5 回の試行の平均処理時間である．また，各棒グラフには，5 回の試行の処理時間の標準偏差を平均処理時間から正負の方向にエラーバーで示している．Sequence Filtering によって削減されるシーケンスが多い，Configuration 1 から 3 と Configuration 5，6 については，Sequence Filtering による処理時間の削減効果が得られた．Row Filtering については，Q2，Q6，Q7 を除き，削減される行数が多い Configuration 1 から 5 において，Row Filtering による処理時間の削減効果が得られた．特に，Sequence Filtering によって削減されるシーケンスが少ない一方で Row Filtering によって削減される行数が多い Configuration 4 においては，Q1，2，6，7 を除



(a) Q1 から Q3



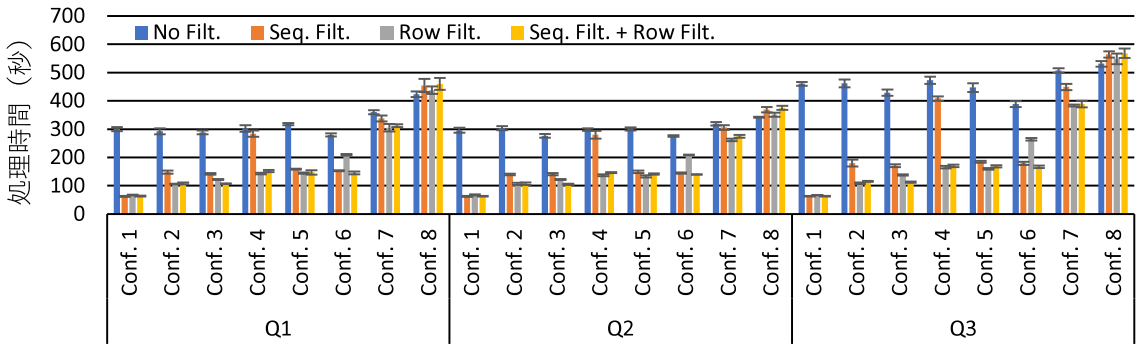
(b) Q4 から Q6



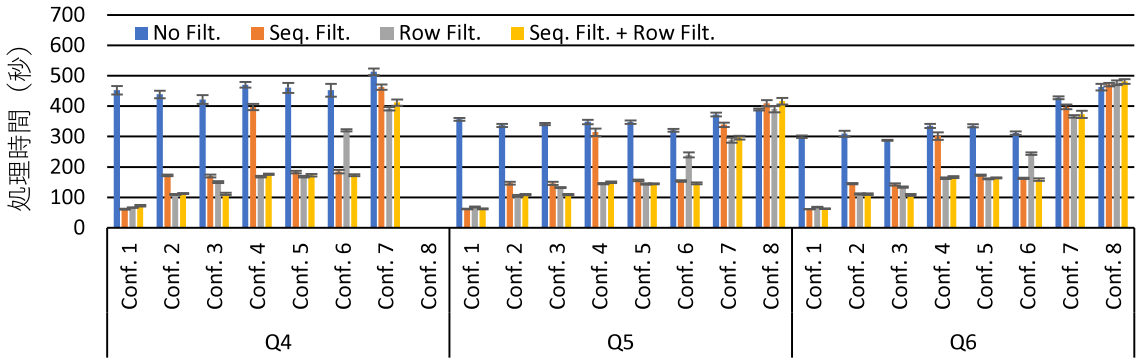
(c) Q7

図 18 PostgreSQL における人工データを用いた実験結果
Fig. 18 Experiment result of synthetic data in PostgreSQL.

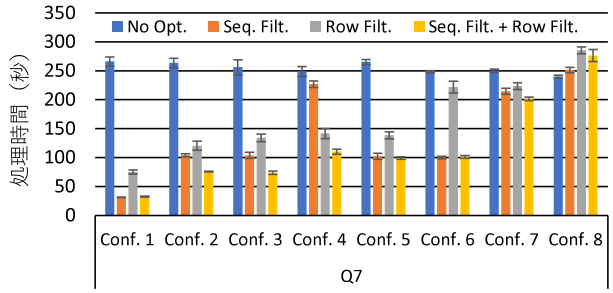
き，Row Filtering の方が Sequence Filtering よりも処理時間が削減された．なお，Q2，Q6，Q7 については，Window 関数における Window 幅が広く設定されているため，Row Filtering の処理コストが大きくなり，No Filtering に対し，処理時間の削減効果が得られなかったと考えられる．特に，Q7 については Window 幅が 99 と他のクエリよりも大幅に広く設定されているため，Row Filtering の処理コストが非常に大きくなっているが，後述するように，コストモデルにより各最適化手法を適用した場合にかかる処理時間を適切に見積もることができているため，この場合の最適化手法として Row Filtering が選ばれることはない．Q5 についても Window 幅が広く設定されているが，Q5 については Row Filtering に限らず，全体的に処理時間が長くなっている．これは，パターンの最初のパターン変数である X が繰返しを含む正規表現で，かつそのパターン変数に一致する値に関する条件が存在しないため，次のパターン変数である (Y|Z) に一致しない場合，パターンに一致するかを探索し始めた行の次の行から行パターンマッチングを再開するため，全体として行パターンマッチングにおけるオートマトンの遷移回数が増えるためである．そのため，Row Filtering のオーバヘッドが影響せず，処理時間の削減効果が得られたと考えられる．Sequence Filtering + Row Filtering については，Q3 と Q5 の Configuration 2 において処理時間が最も短くなったが，Sequence Filtering によ



(a) Q1 から Q3



(b) Q4 から Q6



(c) Q7

図 19 Spark における人工データを用いた実験結果
Fig. 19 Experiment result of synthetic data in Spark.

る処理時間の削減効果とほぼ同じとなった．

コスト計算自体の処理コストを含める Cost Base と，コスト計算自体の処理コストを含めない，コスト式により処理時間が最短となる最適化手法を比較すると，その誤差は 2.5%程度となり，ほぼ等しい結果となった．また，Configuration 4 と 7 を除き，コスト式により処理時間が最短となる最適化手法を選択した場合の処理時間が実際に最短となった．Configuration 4 と 7 についても，コスト式により処理時間が最短となる最適化手法と実際に処理時間が最短となる最適化手法の処理時間の誤差は 11%程度で，大きな差はなかった．以上から，コストモデルによって適切に処理時間を見積もることができている．

クエリごとの処理時間を比較すると，Q5 のみ他のクエリよりも全体的に処理時間が長くなっているが，この理由は前述のとおりである．

次に，Spark を用いた人工データに対する実験結果を**図 19** に示す．なお，Q4 の Configuration 8 については，メモリ不足により処理が完了できないため結果を示していない．フィルタリングされる行が 1 行も存在しない Configuration 8 を除き，すべての効率化手法において，処理時間の削減効果が得られている．すべての行がフィルタリングされる Configuration 1 については，Sequence Filtering が最も処理時間が短くなった．Sequence Filtering によって残る行の割合（$\alpha \times \beta$）と Row Filtering によって残る行の割合（γ）が等しい Configuration 2，4，5，7 については，Q1 と 6 を除き Row Filtering が最も処理時間が短くなった．これは，Spark においては，Row Filtering における Window 関数のコストが小さいためと思われる．一方で，Sequence Filtering によって残る行の割合の方が Row Filtering によって残る行の割合よりも小さい，Configuration 3 と 6 については，Sequence Filtering + Row Filtering が最も処理時間が短くなった．

クエリごとの処理時間について，シーケンシャルなパターンである Q1 よりも繰返しの正規表現を含むパターンである Q2 の方が処理時間が短くなっているが，これは，Q1 で用いている人工データでは受理状態までの異なる状態間の遷移が 2 回（X → Y → Z, ... の繰返し）であるのに対し，Q2 で用いている人工データでは受理状態までの異なる状態間の遷移が平均で約 1.67 回になる（X → Z，X → Y → Z，X → Y → Z, ... の繰返し）．本研究において実装した Spark の NFA の実装上，異なる状態間の遷移に時間を要するため，遷移回数が少ない Q2 の方が処理時間が短くなっている．Q7 は Q2 よりもさらに処理時間が短いが，これは，パターン変数 Y に，Q2 では最大 2 行マッチして Z に遷移するのに対し，Q7 では 98 行マッチして Z に遷移するため，異なる状態間の遷移回数の総数が Q7 の方が少なくなるためである．また，Q3 および Q4 の処理時間が他のクエリよりも長くなっている．これは，Q3 および Q4 のパターンの最初に選択条件が含まれているためである．Spark での NFA の実装上，2 つの状態のいずれかに遷移しうる選択条件の場合，2 つの状態に対して遷移可能かどうかについて条件を 2 回参照する．そのため，ある 1 つの状態に遷移可能かどうかについて条件を 1 回参照するだけでよい他のクエリと比較し，処理時間が長くなる．Q6 は，パターンは複雑である一方で，パターンの最初のパターン変数である X が繰返しを含まず，かつそのパターン変数に one variable condition が指定されているため，最初のパターン変数 X でパターンに一致するかどうかがおおむね決まるものが多く，Q1 や Q2 と同様の処理時間となっている．

9.2 実データを用いた実験

次に，実データとして Foursquare Dataset [7], [16], [17] を用い，効率化の効果を検証する．本実験では，dataset_TIST2015_Checkins および dataset_TIST2015_POIs を用いる．dataset_TIST2015_Checkins は，属性として (User ID, Venue ID, UTC time, Timezone offset in minutes) の 4 つを持ち，行数は，33,263,633 行，シーケンス数は，266,909 である．dataset_TIST2015_POIs は，Venue のマスタデータである．

本実験では Q8 から Q13 の 6 種類のクエリを用いる．これらのクエリを**図 20**，**図 21**，**図 22**，**図 23**，**図 24**，**図 25** に示す．なお，図 21 以降は PATTERN 句と DEFINE 句のみ示す．Q8 はシーケンシャルなパターン，Q9 および Q10 は繰返しを表す正規表現を含むパターン，Q11 は選択を表す正規表現を含むパターン，Q12 および Q13 は繰返しや選択を表す正規表現が複合的に含まれているパターンである．各クエリに指定した条件から，α，β，γ，δ の値は**表 5** のようになる．Q10 と Q13 は α の値が低いクエリとなっている．また，Q11 と Q12 は γ の値が低いクエリとなっている．

```
SELECT *
FROM dataset_TIST2015_Checkins JOIN dataset_TIST2015_POIs
        ON dataset_TIST2015_Checkins.Venue_ID = dataset_TIST2015_POIs.Venue_ID
MATCH_RECOGNIZE ( PARTITION BY User_ID
                  ORDER BY UTC_time
                  MEASURES X.UTC_time AS X_UTC_time,
                           Y.UTC_time AS Y_UTC_time
                  ONE ROW PER MATCH
                  PATTERN ( X Y )
                  DEFINE X AS X.Venue_category_name = 'Home (private)',
                         Y AS Y.Venue_category_name = 'Office' )
```

図 20 問合せ Q8
Fig. 20 Query Q8.

```
...
MATCH_RECOGNIZE ( ...
                  PATTERN ( X Y+ Z )
                  DEFINE X AS X.Venue_category_name = 'Home (private)',
                         Y AS COUNT(Y.*) <= 3,
                         Z AS Z.Venue_category_name = 'Office' )
```

図 21 問合せ Q9
Fig. 21 Query Q9.

```
...
MATCH_RECOGNIZE ( ...
                  PATTERN ( X Y* Z )
                  DEFINE X AS X.Venue_category_name = 'Hospital'
                             AND X.Country_code = 'FR',
                         Y AS COUNT(Y.*) <= 3,
                         Z AS Z.Venue_category_name = 'Restaurant' )
```

図 22 問合せ Q10
Fig. 22 Query Q10.

```
...
MATCH_RECOGNIZE ( ...
                  PATTERN ( X ( Y | Z ) )
                  DEFINE Y AS Y.Venue_category_name = 'Shoe Store'
                         Z AS Z.Venue_category_name = 'Liquor Store' )
```

図 23 問合せ Q11
Fig. 23 Query Q11.

```
...
MATCH_RECOGNIZE ( ...
                  PATTERN ( ( X | Y )+ Z W )
                  DEFINE X AS X.Country_code = 'ES' AND COUNT(X.*) <= 2
                         Y AS Y.Country_code = 'PT' AND COUNT(Y.*) <= 2)
```

図 24 問合せ Q12
Fig. 24 Query Q12.

```
...
MATCH_RECOGNIZE ( ...
                  PATTERN ( X Y * ( Z | W ) )
                  DEFINE X AS X.Venue_category_name = 'Airport',
                         Y AS Y.Country_code = 'DE' AND COUNT(Y.*) <= 3,
                         Z AS Z.Country_code = 'CH',
                         W AS W.Country_code = 'FR' )
```

図 25 問合せ Q13
Fig. 25 Query Q13.

まず，PostgreSQL を用いた実験結果を**図 26** に示す．Sequence Filtering については，すべてのクエリにおいて処理時間の削減効果が得られた．Row Filtering については，Q8，Q11，Q12 において処理時間の削減効果が得ら

表 5 Q8 から Q13 における α, β, γ, δ の値
Table 5 Values of α, β, γ, and δ from Q8 to Q13.

	Q8	Q9	Q10	Q11	Q12	Q13
α	0.263	0.263	0.004	0.079	0.033	0.006
β	0.367	0.614	0.291	0.025	0.313	0.430
γ	0.261	0.443	0.149	0.004	0.010	0.076
δ	0.703	0.703	0.622	0.079	0.033	0.411

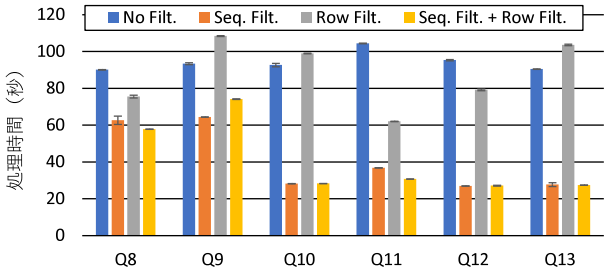


図 26 PostgreSQL における Foursquare Dataset を用いた実験結果
Fig. 26 Experiment result of Foursquare Dataset in PostgreSQL.

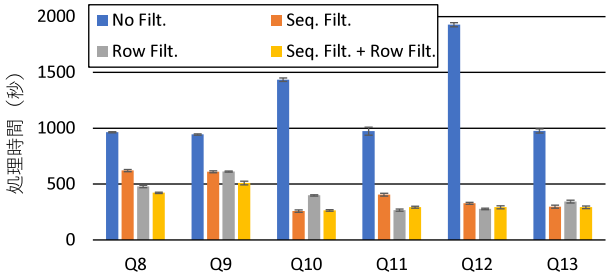


図 27 Spark における Foursquare Dataset を用いた実験結果
Fig. 27 Experiment result of Foursquare Dataset in Spark.

れた．削減される行数が少ない Q9 は処理時間の削減効果が得られていないが，削減される行数が多い Q10 と Q13 においても処理時間の削減効果が得られなかった．これは，Window 関数における Window 幅が大きく，その分，Row Filtering の処理コストが大きくなってしまったためと考えられる．Q8，Q11 については Row Filtering における Window 演算の Window 幅も小さいため，Sequence Filtering + Row Filtering が最も処理時間が短くなった．

次に，Spark を用いた実験結果を**図 27** に示す．すべてのクエリにおいて，各効率化手法の処理時間が No Filtering よりも短くなっており，処理時間の削減効果が得られている．α の値が低い Q10 と Q13 は Sequence Filtering による処理時間の削減効果が大きく，γ の値が低い Q11 と Q12 は Row Filtering による処理時間の削減効果が大きく得られている．Q8 と Q9 は Sequence Filtering + Row Filtering が最も処理時間の削減効果があった．これは，Q10 から Q13 については，Sequence Filtering あるいは Row Filtering のいずれかを適用することにより行数が 1%未満まで削減されるのに対し，Q8 と Q9 は両手法を適用することで行数を 25%未満まで削減できるため，いずれかの手法を適用するよりも処理時間を削減することができると考えられる．

表 6 見積値と実測値の RMSPE（%）
Table 6 RMSPE between measured and estimated values (%).

効率化手法	N = 10,000,000, S = 10,000	N = 5,000,000, S = 1,000
No Filt.	4.13	3.63
Seq. Filt.	24.11	23.18
Row Filt.	11.73	10.64
Seq. Filt. + Row Filt.	18.42	16.35

9.3 コストモデルの妥当性の検証

6.1 節において構築したコストモデルを PostgreSQL を対象に検証する．具体的には，コストモデルから算出した処理時間（見積値）と実際の処理時間（実測値）の比較を，2 通りの行数 N とシーケンス数 S の組合せ（N = 10,000,000，S = 10,000 と，N = 5,000,000，S = 1,000）について 9.1 節に示した 7 つのクエリを対象に行う．比較には，各効率化手法ごとに見積値と実測値の平均平方二乗誤差率（RMSPE）を用いる．RMSPE の算出式は以下のとおりである．下記の式において，クエリ数は Q1 から Q7 の 7 種類，Configuration 数は Configuration 2 から 8 の 7 種類，試行回数は 5 回である．なお，Configuration 1 については，Sequence Filtering および Sequence Filtering + Row Filtering において，実測値が非常に小さな値（0.01 秒未満）となるため，式 (5) の分子における相対誤差が非常に大きな値となる．一方で，処理時間が非常に短い処理のコストを正確に見積りたいというニーズはないと考えられるため，ここでは評価対象から除外する．

$$RMSPE(\%) = \sqrt{\frac{\sum\left(\frac{(\text{見積値})-(\text{実測値})}{(\text{実測値})}\right)^2}{(\text{クエリ数})\times(\text{Conf. 数})\times(\text{試行回数})}} \times 100 \quad (17)$$

コストの導出に際しては，c，r，w の値が必要となる．そこで，本節で検証対象とする行数 N とシーケンス数 S の組合せとは別の組合せである，N = 10,000,000，S = 1,000 のときの c，r，w の値を実測し，それを利用して処理時間を見積もる．

表 6 に結果を示す．いずれの効率化手法においても 10%から 20%程度の誤差で見積もることができている．No Filt. は行パターンマッチングの処理時間のみを見積もっているため，他の効率化手法よりも誤差が小さい．次に，**表 7** にクエリごとの見積値と実測値の RMSPE を示す．上記の式 (17) における分母を (Conf. 数) × (試行回数) で計算した値となる．各クエリとも，10%から 20%程度の誤差で見積もることができている．Q4 について，Sequence Filter-

表 7 クエリごとの見積値と実測値の RMSPE（%）
Table 7 RMSPE between Measured and Estimated Values for 6 Queries (%).

クエリ	効率化手法	$N = 10{,}000{,}000$, $S = 10{,}000$	$N = 5{,}000{,}000$, $S = 1{,}000$
Q1	No Filt.	1.68	0.83
	Seq. Filt.	15.36	14.50
	Row Filt.	9.70	9.37
	Seq. Filt. + Row Filt.	13.67	13.85
Q2	No Filt.	5.31	5.19
	Seq. Filt.	16.88	16.02
	Row Filt.	13.99	13.84
	Seq. Filt. + Row Filt.	13.77	14.30
Q3	No Filt.	1.64	2.20
	Seq. Filt.	20.80	19.79
	Row Filt.	15.44	15.21
	Seq. Filt. + Row Filt.	14.86	15.08
Q4	No Filt.	2.73	2.26
	Seq. Filt.	34.39	33.35
	Row Filt.	9.66	9.66
	Seq. Filt. + Row Filt.	25.31	24.30
Q5	No Filt.	7.20	5.64
	Seq. Filt.	7.38	8.16
	Row Filt.	6.70	5.06
	Seq. Filt. + Row Filt.	10.49	10.74
Q6	No Filt.	2.88	2.57
	Seq. Filt.	30.44	28.90
	Row Filt.	7.00	6.92
	Seq. Filt. + Row Filt.	17.25	16.42
Q7	No Filt.	1.10	1.70
	Seq. Filt.	19.29	18.77
	Row Filt.	10.53	1.36
	Seq. Filt. + Row Filt.	19.80	2.38

ing および Sequence Filtering + Row Filtering の誤差が大きいが，これは Sequence Filtering において one variable condition の数だけ集約関数 MAX を実行し，属性 `<flag>` を付与することが影響していると思われる．Q4 においては one variable condition が 4 つ存在し，Q1 から Q7 のなかで最も多いため，誤差が最も大きくなっていると思われる．

以上から，提案するコストモデルにより処理時間を見積もることができていることから，c，r，w の値を適切に設定することで，最も処理時間が短くなる効率化手法を適切に選択することが可能となる．

10. まとめ

本研究では，RDB 等に格納された大規模なシーケンスデータに対する行パターンマッチングの処理コストを削減するために，行パターンマッチングの前に，パターンオカレンスになり得ない行を事前にフィルタリングする効率化を行うことで処理を効率化する，Sequence Filtering と Row Filtering の 2 種類の効率化手法を提案した．Sequence Filtering については，パターン変数に対するすべての one variable condition について，満たす行が存在するシーケンス以外をフィルタリングする手法を提案した．Row Filtering については，一部のパターン変数に対してのみ one variavle condition が指定されているクエリにも適用可能な手法を提案した．また，PostgreSQL と Spark を対象に SQL/RPR を実装し，評価実験により提案する効率化手法の有効性を示した．さらに，各効率化手法のコストモデルを構築し，その妥当性を示すことで，コストモデルを用いて適切な効率化手法の選択が可能であることを示した．

今後の課題として行パターンマッチングだけでなく，クエリ全体の詳細なコストモデルを構築することがあげられる．また，one variable condition 以外の条件も利用し，より多くの行をフィルタリング可能な効率化手法を検討することがあげられる．また，行パターンマッチング対象となるデータがメモリに乗り切らない場合のコストモデルの検証，および Spark のコストモデルの検証があげられる．

謝辞 本研究の一部は，日本学術振興会科学研究費・基盤研究（B）（#19H04114）による．

参考文献

[1] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A. and Zaharia, M.: Spark SQL: Relational Data Processing in Spark, *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pp.1383–1394 (2015).
[2] Cadonna, B., Gamper, J. and Bohlen, M.H.: Efficient Event Pattern Matching with Match Windows, *Proc. 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD2012*), pp.471–479 (2012).
[3] Baldacci, L. and Golfarelli, M.: A Cost Model for SPARK SQL, *Proc. 2019 IEEE Transactions on Knowledge and Data Engineering*, Vol.31, No.5, pp.819–832 (2019).
[4] Cha, S.K., Moraru, I., Jang, J., Truelove, J., Brumley, D. and Anderson, D.G.: SplitScreen: Enabling Efficient, Distributed Malware Detection, *Journal of Communications and Networks*, Vol.13, No.2, pp.187–200 (2011).
[5] Choi, B., Chae, J., Jamshed, M., Park, K. and Han, D.: DFC: Accelerating String Pattern Matching for Network Applications, *Proc. 13th USENIX Symposium on Networked Systems Design and Implementation*

(*NSDI2016*), pp.551–565 (2016).
[6] Demers, A., Gehrke, J., Panda, B., Riedewald, M., Sharma, V. and White, W.: Cayuga: A General Purpose Event Monitoring System, *CIDR 2007*, pp.412–422 (2007).
[7] Foursquare: FOURSQUARE, Foursquare (online), available from ⟨https://foursquare.com⟩ (accessed 2020-03-18).
[8] ISO/IEC TR 19075-5:2016(E): Information technology - database languages - sql technical reports - part 5: Row pattern recognition in SQL, Technical Report, ISO Copyright Office (2016).
[9] Laker, K.: A Technical Deep Dive into Pattern Matching using MATCH_RECOGNIZE, HrOUG 2016, Business Intelligence & Analytics, Hall C (2016) (online), available from ⟨http://www.oracle.com/technetwork/database/bi-datawarehousing/mr-deep-dive-3769287.pdf⟩ (accessed 2020-03-18).
[10] Mei, Y. and Madden, S.: ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events, *Proc. 2009 ACM SIGMOD International Conference on Management of Data*, pp.193–206 (2009).
[11] Nakabasami, K., Kitagawa, H. and Nasu, Y.: Optimization of Row Pattern Matching over Sequence Data in Spark SQL, *Proc. 30th International Conference on Database and Expert Systems Applications* (*DEXA2019*), pp.3–17 (2019).
[12] Nasu, Y., Kitagawa, H. and Nakabasami, K.: Efficient Pattern Matching using Pattern Hierarchies for Sequence OLAP, *Proc. 21st International Conference on Big Data Analytics and Knowledge Discovery* (*DaWaK2019*), pp.89–104 (2019).
[13] Rashmi, C. and Hemantha Kumar, G.: Parallel Processing Approach for Pattern Matching using MPI, *International Journal of Computer Applications*, Vol.180, No.11, pp.31–34 (2018).
[14] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J. and Zhu, H.: Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs, *Proc. 16th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI2019*), pp.631–648 (2019).
[15] Wu, E., Diao, Y. and Rizvi, S.: High-Performance Complex Event Processing over Streams, *SIGMOD 2006*, pp.407–418 (2006).
[16] Yang, D., Zhang, D., Chen, L. and Qu, B.: NationTelescope: Monitoring and Visualizing Large-Scale Collective Behavior in LBSNs, *Journal of Network and Computer Applications* (*JNCA*), Vol.55, pp.170–180 (2015).
[17] Yang, D., Zhang, D. and Qu, B.: Participatory Cultural Mapping Based on Collective Behavior Data in Location Based Social Networks, *ACM Trans. Intelligent Systems and Technology* (*TIST*) (2015).
[18] Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S. and Stonica, I.: Spark: Cluster Computing with Working Sets, *Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing* (*HotCloud2010*), Vol.55, p.10 (2010).
[19] Zhang, H., Diao, Y. and Immerman, N.: On Complexity and Optimization of Expensive Queries in Complex Event Processing, *Proc. 2014 ACM SIGMOD International Conference on Management of Data* (*SIGMOD2014*), pp.217–228 (2014).

中挾 晃介 （正会員）

2015年3月筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了．同年4月（公財）鉄道総合技術研究所入所．2020年3月筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士後期課程修了．現在，（公財）鉄道総合技術研究所信号・情報技術研究部運転システム研究室研究員．博士（工学）．輸送計画，運行管理に関する研究開発に従事．日本データベース学会会員．

北川 博之 （正会員）

1978年東京大学理学部物理学科卒業．1980年同大学理学系研究科修士課程修了．日本電気（株）勤務の後，筑波大学電子・情報工学系講師，同助教授を経て，現在，筑波大学計算科学研究センター教授．理学博士（東京大学）．データベース，情報統合，データマイニング，情報検索等の研究に従事．電子情報通信学会フェロー，日本データベース学会監事，ACM，IEEE，日本ソフトウェア科学会各会員．本会フェロー．