Title: Pattern matching in sequences of rows (11)

Author: Fred Zemke (Oracle), Andrew Witkowski (Oracle), Mitch Cherniak (Streambase), Latha Colby (IBM)

Source: U.S.A.
Status: Change proposal
Date: March 2, 2007

Abstract

This paper addresses the need of finding patterns in a sequence of rows. We see this often in Complex Event Processing where business processes are driven from sequence of events, in Security applications where detection of unusual behavior definable with regular expressions is needed, in Financial applications where detecting stock patterns is critical, Fraud Detection applications, and for RFID processing where tracking of valid paths for RFID tags is needed.

This paper proposes new SQL functionality for finding patterns in sequences of rows. Patterns are defined using familiar syntax of Regular Expressions (RE). RE variables span sub-sequences of rows and are defined using conditions on individual rows and their aggregates. New pattern recognition clause, the MATCH_RECOGNIZE clause, can be applied either to a table expression or can be used in a window definition. In the former case we provide two modes of matching: one that issues a single row for each match of the pattern or one that issues all rows that matched it. In both cases, we can perform and emit calculations on the pattern variables. In the latter case, a window size is defined using a pattern, and then standard processing on the window of rows can take place.

History

This paper proposes syntax and semantics for recognizing patterns in rows of a table.

version 4: respond to comments from Xin Zhou (xin.zhou@ncr.com); change “lazy quantifier” to “reluctant quantifier” to match terminology in [WLG-027].

version 6: references to UCLA work

version 7: reference Oracle internal paper

version 8: add my understanding of IBM’s proposal

version 9: { ALL | DISJOINT | SUCCESSIVE | JOINED }; ideas from Streambase;

version 10: begin issues list

version 11: first try at rigorous definition of singleton vs. group variables. Resolutions to issues in version 10 per telecon 25 Jan 2007. Introduction of MEASURES clause.

References

[Foundation:2003] Jim Melton (ed), "ISO International Standard (IS) Database Language SQL - Part 2: SQL/Foundation", ISO/IEC 9075-2:2003

[Friedl] Jeffrey E.F. Friedl, "Mastering regular expressions", 1997, O'Reilley and Associates

[WLG-027] Fred Zemke, Weiran Zhang, "XQuery regular expression support in SQL/Foundation", ISO/IEC JTC1/SC32 WG3:WLG-027 = ANSI INCITS H2-2005-292r2

[Xin Zhou] Xin Zhou, "Comments on ANSI draft for Row Pattern Recognition", attached to email to Andy Witkowski dated 2 Oct 2006, containing comments on version 3 of this paper.

[Sadri Ph.D.] M. Reza Sadri, "Optimization of sequence queries in database systems", Ph.D. thesis, UCLA, 2001, http://wis.cs.ucla.edu/theses/thesis_sadri.pdf

[Sadri pods] Reza Sadri et.al., "Optimization of sequence queries in database systems", http://www.cs.ucla.edu/%7Ezaniolo/papers/pods2001.pdf

[Sadri tods] Reza Sadri et.al., "Expressing and optimizing sequence queries in database systems", ACM Transactions on database systems, Vol 29, No. 2, June 2004, pages 282-318, http://cs.ucla.edu/~zaniolo/papers/todsjune04.pdf

[Continuous query FS]
Andrew Witkowski, Hakan Jakobsson, Tolga Bozkaya, Srikanth Bellamkonda, Jim Stamos, John Ciminski, Bhaskar Ghosh, "Functional Specification for Continuous Queries", Oracle Internal Document. Email andrew.witkowski@orcle.com for a copy.

[Streambase patterns] Streambase Systems, "Pattern matching for StreamSQL", October 2006

1. Prior work

Prior work on this subject may be found in the references [Sadri Ph.D.], [Sadri tods] and [Sadri pods], and [Continuous Query FS].

2. Pattern Recognition Over Table Expression

This paper proposes syntax and semantics for recognizing patterns in rows of a table.

The syntax enhances the capability of the FROM clause with a MATCH_RECOGNIZE clause to specify the row pattern. There are two variants of the MATCH_RECOGNIZE clause:

1. ONE ROW PER MATCH, which creates a single summary row for each match of the pattern (the default).
2. ALL ROWS PER MATCH, which emits one row for each row of each match.

In addition we propose a window version of MATCH_RECOGNIZE. In this case the pattern defines the window size. The proposal for window clause extends its semantics and allows us to export additional columns calculated using pattern variables.

2.1 Example of ONE ROW PER MATCH

The following example will illustrate MATCH_RECOGNIZE with the ONE ROW PER MATCH option. Let Ticker (Symbol, Tstamp, Price) be a table with three columns representing historical stock prices. Symbol is a character column, Tstamp is a timestamp column (for simplicity shown as increasing integers) and Price is a numeric column. It is desired to partition the data by Symbol, sort it into increasing Tstamp order, and then detect the following pattern in Price: a falling price, followed by a rise in price that goes higher than the price was when the fall began. After finding such patterns, it is desired to report the starting time, starting price, inflection time (last time during the decline phase), low price, end time, and end price.

The query to run on this sample data is

```
SELECT a_symbol,
       a_tstamp,                   /* start time */
       a_price,                    /* start price */
       max_c_tstamp,               /* inflection time */
       last_c_price,               /* low price */
       max_f_tstamp,               /* end time */
       last_c_price,               /* end price */
       Matchno
FROM Ticker MATCH_RECOGNIZE (
            PARTITION BY Symbol
            ORDER BY Tstamp
            MEASURES A.Symbol       AS a_symbol,
                     A.Tstamp       AS a_tstamp,
                     A.Price        AS a_price,
                     MAX (C.Tstamp) AS max_c_tstamp,
                     LAST (C.Price) AS last_c_price
                     MAX (F.Tstamp) AS max_f_tstamp
                     MATCH_NUMBER   AS matchno
            ONE ROW PER MATCH
            AFTER MATCH SKIP PAST LAST ROW
            MAXIMAL MATCH
            PATTERN (A B C* D E* F+)
            DEFINE /* A defaults to True, matches any row */
                   B AS (B.price < PREV(B.price)),
                   C AS (C.price <= PREV(C.price)),
                   D AS (D.Price > PREV(D.price)),
                   E AS (E.Price >= PREV(E.Price)),
                   F AS (F.Price >= PREV(F.price)
                         AND F.price > A.price))
```

In this example, six variable names are defined (A, B, C, D, E, and F).

The processing of MATCH_RECOGNIZE is as follows:

1. The table is partitioned according to the PARTITION component.
2. The partitions are ordered according to the ORDER BY component.
3. The ordered partitions are searched for matches to the PATTERN. Semantics of the pattern are discussed later, but the basic idea is that it is a regular expression involving variable names. The SKIP TO clause determines where we will start the next match after the current one has been found. In the example above, we start the next match PAST LAST ROW of the current match, i.e., in the row after the current pattern. Other SKIP TO options allow us to start the next match at the next row after the first row of the current match (SKIP TO NEXT ROW) or at the first or last row matching a variable (SKIP TO FIRST/LAST <variable> correspondingly).
4. Variable names in the pattern are classified as either singleton (if they are not quantified) or group (if they have a quantifier, such as + or *). Singleton variables refer to exactly one row while group variables identify a set of rows.
5. Pattern matching operates in two matching modes MAXIMAL and INCREMENTAL. Roughly speaking, the MAXIMAL mode finds the longest matching sequence of rows (for exact definition of longest sequence see Section “Match recognition” on page 18.). After a match is found, we calculate measures (aggregates on grouped variables and expressions on singleton variables) on the matched rows as defined by the MEASURES and finally we emit a single row as defined by the SELECT list. We then attempt the next match by skipping the row designated by then SKIP TO clause and restarting matching again. The INCREMENTAL mode pretends that the partition is increased incrementally row by row (starting from empty partition) before each application of MAXIMAL match. We increase the partition by a single row and apply the MAXIMAL match until no more matches starting from a given row are found. Observe that, as opposed to the MAXIMAL match, we can output multiple matches rooted at the same starting row. After that SKIP TO clause is applied to find the next matching point.
6. Matches within a partition are numbered sequentially according to the order of their first rows. This number is assigned to the column designated by MATCH_NUMBER. This feature is not strictly necessary for ONE ROW PER MATCH (it can be computed using ROW_NUMBER). However, there are scenarios with the ALL ROWS PER MATCH option in which the MATCH_NUMBER cannot be computed later.
7. The MEASURES clause defines exported columns that contain expressions over the pattern variables. The expressions can reference partition columns, order by columns, singleton variables and aggregates on the group variables.

8. The MATCH_RECOGNIZE operates in two modes: ONE ROW PER MATCH and ALL ROWS PER MATCH. The former creates a single summary row for each match and the latter one row for each row of the match. For ONE ROW PER MATCH option, the visible columns of the table expression T with the MATCH_RECOGNIZE clause are the partition columns, order by columns and columns defined in the MEASURES clause. For ALL ROWS PER MATCH the visible columns include also all columns of T.

2.2 Example of ALL ROWS PER MATCH

The previous example can be modified slightly to illustrate ALL ROWS PER MATCH. In contrast to the previous option, ALL ROWS PER MATCH returns one row for each row of each match of the pattern. Observe that in addition to partitioning, ordering and measure columns we can reference other columns of the Ticker table. Inside the MEASURES clause we can use a CLASSIFIER component that may be used to declare a character result column whose contents on each row is the variable name that the row matched with.

```
SELECT T.Symbol,                  /* row’s symbol */
       T.Tstamp,                  /* row’s time */
       T.Price,                   /* row’s price */
       T.classy                   /* row’s classifier */
       T.a_tstamp,                /* start time */
       T.a_price,                 /* start price */
       T.max_c_tstamp,            /* inflection time */
       T.last_c_price,            /* low price */
       T.max_f_tstamp,            /* end time */
       T.last_c_price             /* end price */
FROM Ticker MATCH_RECOGNIZE (
            PARTITION BY Symbol
            ORDER BY Tstamp
            MEASURES A.Symbol       AS a_symbol,
                     A.Tstamp       AS a_tstamp,
                     A.Price        AS a_price,
                     MAX (C.Tstamp) AS max_c_tstamp,
                     LAST (C.Price) AS last_c_price
                     MAX (F.Tstamp) AS max_f_tstamp
                     MATCH_NUMBER   AS matchno,
                     CLASSIFIER     AS Classy
            ALL ROWS PER MATCH
            AFTER MATCH SKIP PAST LAST ROW
            MAXIMAL MATCH
            PATTERN (A B C* D E* F+)
            DEFINE /* A defaults to True, matches any row */
                   B AS (B.price < PREV(B.price)),
                   C AS (C.price <= PREV(C.price)),
                   D AS (D.Price > PREV(D.price)),
                   E AS (E.Price >= PREV(E.Price)),
                   F AS (F.Price >= PREV(F.price)
                         AND F.price > A.price) ) T
```

Note that the syntax provides access to both the all columns of the table T as well as the MEASURES columns. In a naive implementation, this means that MATCH_RECOGNIZE will not output any rows until it completely recognizes a match. However, if the user does not request aggregates from the end of the match, there are patterns in which it is possible to determine the beginning of a match even if the end of the match is not yet known. For example, PATTERN ((A B) *) — after recognizing A and B, MATCH_RECOGNIZE can output those two rows before continuing to look for a longer match. Thus an optimized implementation may be able to output rows sooner than recognition of an entire match.

When using skip options other than SKIP PAST LAST ROW or when using INCREMENTAL matching option, a row of the input table might occur in more than one match. In that case, MATCH_RECOGNIZE generates one row for each match that the row participates in. It can happen that the classifier column of the row will have different values in different matches, because the row is “hit” by different parts of the pattern.

Because a row can occur in more than one match, there are scenarios in which it will be impossible to determine which match a row belongs to. The MATCH_NUMBER feature allows the user to declare an exact numeric result column to receive the sequential number of the current match within the partition.

2.3 Syntax

The complete syntax for the MATCH_RECOGNIZE clause involves the following components:

PARTITION BY — optional

ORDER BY — optional, but in practice we expect this will always be used.

MEASURES - optional, but in practice we expect this will always be used

{ ONE ROW | ALL ROWS } PER MATCH — default to ONE ROW

AFTER MATCH SKIP { TO NEXT ROW | PAST LAST ROW | TO LAST <variable> | TO FIRST <variable> } - default AFTER MATCH SKIP PAST LAST ROW

{ MAXIMAL | INCREMENTAL } MATCH - defaults to MAXIMAL MATCH

PATTERN — mandatory

SUBSET — optional

DEFINE — mandatory

CLASSIFIER - optional (ALL ROWS PER MATCH only)

MATCH_NUMBER - optional

Now we consider each component of MATCH_RECOGNIZE in turn.

2.4 ONE ROW PER MATCH vs. ALL ROWS PER MATCH

ONE ROW PER MATCH indicates that the result has one row for each match. Columns of this row are referenced by variable names that are defined by partition, ordering and measure components.

ALL ROWS PER MATCH indicates that the result has one row for each row of each match. For example, using MATCH_RECOGNIZE (ALL ROWS PER MATCH...) AS T, the name T may be used to reference the current row of a match. This name has a column for each column of the input table, with the same name and data type as the corresponding input column. If the CLASSIFIER feature is specified, it also has a character column for the classifier. If the MATCH_NUMBER feature is specified, it also has an exact numeric scale zero column for the MATCH_NUMBER. In addition to the columns of T, all of the result columns provided by the ONE ROW PER MATCH are referenceable, using the variable names defined by MEASURES components.

2.5 PARTITION BY

PARTITION BY is used to specify that the rows of the table are to be partitioned by one or more columns. The syntax and semantics are the same as in a WINDOW clause. If there is no PARTITION BY, then all rows of the table constitute a single partition.

2.6 ORDER BY

ORDER BY is used to specify the order of rows within a partition. If the order of two rows in a partition is not determined by the ORDER BY, then the results of MATCH_RECOGNIZE is non-deterministic.

2.7 MEASURES

The MEASURES clause defines exported columns that contain expressions over the pattern variables. The expressions can reference partition columns, order by columns, singleton variables and aggregates on the group variables, and aggregates on the columns of the table that is a targed of MATCH_RECOGNIZE clause.

The MATCH_RECOGNIZE operates in two modes: ONE ROW PER MATCH and ALL ROWS PER MATCH. The former creates a single summary row for each match and the latter one row for each row of the match. For ONE ROW PER MATCH option, the visible columns of the table expression T with the MATCH_RECOGNIZE clause are the partition columns, order by columns and columns defined in the MEASURES clause. For ALL ROWS PER MATCH the visible columns include also all columns of T.

2.8 AFTER MATCH SKIP TO

The AFTER MATCH SKIP TO clause determines the resumption point of pattern matching after a match has been found. The syntax for resuming matching is:

```
AFTER MATCH SKIP { TO NEXT ROW | PAST LAST ROW |
                   TO FIRST <variable> | TO LAST <variable>|
                   <variable> }
```

where <variable> is a pattern variable. The default for the clause is default AFTER MATCH SKIP PAST LAST ROW. The options operate as follows:

SKIP TO <variable name>: used with a singleton pattern variable, to cause pattern matching to resume at the row that is classified by the pattern variable.

SKIP TO NEXT ROW: to resume pattern matching at the next row of the first row of the current match.

SKIP PAST LAST ROW: to resume pattern matching at the next row after the last row of the current match.

SKIP TO FIRST <variable name>: used with a group pattern variable, to resume pattern matching at the first row of the designated group

SKIP TO LAST <variable name>: used with a group pattern variable, to resume pattern matching at the last row of the designated group

Here are the usage rules of SKIP clause.

The <variable> may appear more than once in the pattern. In this case, SKIP TO <variable> is a syntax error, since the variable is not a singleton. SKIP TO FIRST will skip to the first row of the group of rows matched by any instance of the variable; SKIP TO LAST will skip to the last row that is classified by any instance of the variable.

The <variable> can be qualified zero times, for example, A in (A*), or A in (A | B). In this case SKIP TO FIRST and SKIP TO LAST can miss A as it has never been matched in the pattern. In this case, SKIP TO FIRST | LAST is equivalent to SKIP PAST LAST ROW.

SKIP may direct us to start next match at the same point that the last match started. For example, in pattern (X Y+ Z), SKIP TO X would reposition the next match at the start of the previous one resulting in an infinite loop. We consider it a user error, and will signal it at a run time.

In many cases use can usually work around the above syntax errors by using SUBSET. For example, if the pattern contains (A | B), then the user can define

```
SUBSET C = ( A, B )
```

and then write SKIP TO C if desired to skip to either A or B, whichever is matched.

2.9 ALL MATCHES vs. MAXIMAL vs. INCREMENTAL MATCH

We distinguish three modes of pattern matching: ALL, MAXIMAL, and INCREMENTAL. Only the last two have to be supported by conforming implementations.

> ALL mode finds all possible matches. Matches may overlap and may start at the same row. In this case, the distinction between greedy and reluctant quantifiers is ignored.
>
> MAXIMAL mode returns a subset of ALL matches. Each match obeys the rules of Section “Match recognition” on page 18. For example, greedy qualifiers match their longest sequence while reluctant qualifiers match the shortest one.
>
> INCREMENTAL mode returns the union of all MAXIMAL matches evaluated after every tuple in the sequence.

The difference between the options MAXIMAL and INCREMENTAL modes can be expressed in terms of awareness of the end-of-partition. In the INCREMENTAL mode, we perform the pattern match as if a particular row were the last row in the partition. Then we get another row and perform the pattern match again with the new end-of-partition. In the non-INCREMENTAL mode, we have the entire partition at the outset, and there is no process of adding another row. INCREMENTAL describes how a real-time streaming environment must work, because you never know the true end-of-partition; all we know is the set of rows that you have already received. When operating on historical data, we can simulate the real-time behavior by using INCREMENTAL, or we can adopt a global view that sees the entire partition.

In all matching modes, after finding a complete match in a partition, scanning for the next match begins at the row designated by the SKIP TO clause.

Examples: given the pattern over table T (row_id, attribute)

```
AFTER MATCH SKIP TO NEXT ROW
PATTERN ( (A* B*) || (B* A*) )
DEFINE A AS t.attribute = 'a',
       B AS t.attribute = 'b'
```

and data in T as follows:

row number	row id	attribute
1	a1	a
2	b1	b
3	b2	b
4	a2	a
5	a3	a
6	a4	a

row number	row id	attribute
7	b3	b
8	b4	b

Then the options will find the following matches (shown as sets of row_ids):

```
ALL: {a1} {a1,b1} {b1} {a1,b1,b2} {b1,b2} {b2} {b1,b2,a2}
{b2,a2} {a2} {b1,b2,a2,a3} {b2,a2,a3} {a2,a3} {a3}
{b1,b2,a2,a3,a4} {b2,a2,a3,a4} {a2,a3,a4} {a3,a4} {a4}
{a2,a3,a4,b3} {a3,a4,b3} {a4,b3} {b3}  {a2,a3,a4,b3,b4}
{a3,a4,b3,b4} {a4,b3,b4} {b3,b4} {b4}

MAXIMAL: {a1,b1,b2} {b1,b2,a2,a3,a4} {a2,a3,a4,b3,b4}

INCREMENTAL: {a1} {a1,b1} {a1,b1,b2}
             {b1,b2,a2} {b1,b2,a2,a3} {b1,b2,a2,a3,a4}
             {a2,a3,a4} {a2,a3,a4,b3} {a2,a3,a4,b3,b4}
```

The more complicated INCREMENTAL mode can be explained operationally as follows. We will maintain a sequence of visible rows in the partition. We start at first row, a1. The sequence of visible rows is [a1]. We match it and issue {a1}. We extend the visible sequence to [a1 ab1], match it and issue {a1,b1}. We extend the visible sequence to [a1 b1 b2], match it and issue {a1,b1,b2}. We extend the visible sequence to [a1,b1,b2,a2]. There is no match, hence we apply SKIP TO clause and skip to row b1. The visible sequence is [b1 b2 a2], we match it and issue {b1,b2,a2}. We extend the visible sequence to [b1 b2 a2 a3], match it and issue {b1,b2,a2,a3}. We extend the sequence to [b1 b2 a2 a3 a4], match it and issue {b1,b2,a2,a3,a4}. We extend the sequence to [b1 b2 a2 a3 a4 b3]. We cannot match it. We then apply SKIP TO clause, and the visible sequence is [b2 a2 a3 a4 b3]. There still is no match, and we SKIP TO next row, making the visible sequence [a2 a3 a4 b3]. We find match {a2,a3,a4,b3} and issue it. We then extend the visible sequence to [a2 a3 a4 b3 b4], match it and issue {a2,a3,a4,b3,b4}.

If, as shown below, we change the SKIP TO clause in the above example to SKIP PAST LAST ROW:

```
AFTER MATCH SKIP PAST LAST ROW
PATTERN ( (A* B*) || (B* A*) )
DEFINE A AS t.attribute = 'a',
       B AS t.attribute = 'b'
```

the matches will be as follows:

```
MAXIMAL: {a1,b1,b2} {a2,a3,a4,b3,b4}

INCREMENTAL: {a1} {a1,b1} {a1,b1,b2}
             {a2} {a2,a3} {a2,a3,a4} {a2,a3,a4,b3}
             {a2,a3,a4,b3,b4}
```

2.10 PATTERN

The PATTERN component is used to specify a regular expression. The regular expression is built from variable names, and may use the following operators:

concatenation: indicated by the absence of any operator sign between two successive items in a pattern

quantifiers: quantifiers are postfix operators with the following choices:

* — 0 or more matches

+ — 1 or more matches

? — 0 or 1 match

{ n } — exactly n matches

{ n, } — n or more matches

{ n, m } — between n and m (inclusive) matches

{ , m } — between 0 and m (inclusive) matches

“reluctant” quantifiers, indicated by an additional question mark (*?, +?, ??, { n, m }?). Reluctant quantifiers try to match as few rows as possible, whereas non-reluctant quantifiers are greedy and try to match as many rows as possible.

alternation: indicated by a vertical bar (|).

grouping: indicated by parentheses

^: indicates begining of partition

$: indicates end of partition

The PATTERN specifies the pattern to be recognized in the ordered sequence of rows in a partition. Each variable name in a pattern corresponds to a boolean condition, which is specified later using the DEFINE component of the syntax. Thus the PATTERN can be regarded as implicitly declaring one or more variable names; the definition of those variable names appears later in the syntax.

In the example

```
PATTERN (X+ Y+)
```

the pattern consists of two variable names, X and Y, each of which is quantified with +, indicating that they must be matched one or more times.

It is permitted for a variable name to occur more than once in a pattern, for example

```
PATTERN (X Y X)
```

2.10.1 PERMUTE

In many practical cases we need to detect patterns that consist of a permutation of variables. It is possible to express a permutation as alternations but it becomes clumsy when many variables are involved. For example a if all permutations of three variables A B C is needed we could express it as: (A B C | A C B | B A C | B C A | C A B | C B A).

To address that we introduce the PERMUTE clause to express a pattern that is a permutation of a list. PERMUTE is a sub-clause of the PATTERN clause and its syntax is:

```
PERMUTE {FACTORS | EXPAND FACTORS } <pattern list>
```

The default is FACTORS.

There are two variants of PERMUTE.

a) PERMUTE FACTORS (<pattern list>) where

```
<pattern list> ::=
    <pattern> { <comma> <pattern> }...
```

that is, <pattern list> is a comma-separated list of two or more <pattern>s. The syntax is transformed into an alternation of all possible concatenations of permutations of the <pattern>s. For example

```
PERMUTE FACTORS ( A{3), B C?, D)
```

is equivalent to

```
( ( A{3} B C? D )
| ( A{3} D B C? )
| ( B C? A{3} D )
| ( B C? D A{3} )
| ( D A{3} B C? )
| ( D B C? A{3} ) )
```

There are no restrictions on individual patterns.

b) PERMUTE EXPAND FACTORS (<pattern list>). The intention of the option is to first expand each pattern in the list, and then apply permutation on the expanded list. This variant assumes that each <pattern> in the <pattern list> is a regular expression with no parenthesis, alternations or variables with unbounded repetitions (i.e., we allow singleton variables or variables qualified by a known number of repetitions only). Each qualified is first expanded into an its equivalent alternation. For example A{,2} is expanded into (A A | A | ()). After expansion we form an alternation of all possible concatenations of the expanded patterns using PERMUTE FACTORS. For example,

```
PERMUTE EXPAND FACTORS ( A{2,3}, B{0,2} )
```

means any combination of three through two A’s and zero through two B’s. It translates to:

```
(   PERMUTE FACTORS ( A A )
  | PERMUTE FACTORS ( A A B )
  | PERMUTE FACTORS ( A A B B )
  | PERMUTE FACTORS ( A A A )
  | PERMUTE FACTORS ( A A A B )
  | PERMUTE FACTORS ( A A A B B ) )
```

The PERMUTE clause can occur in any place in the PATTERN clause. For example, to find a pattern consisting of any permutation of three A variables and two B variables followed by any permutation of two C variables and three D variables we can write:

```
PATTERN ( PERMUTE(A,A,A,B,B) PERMUTE (C,C,D,D,D) )
```

or

```
PATTERN ( PERMUTE EXPAND FACTORS (A{3},B{2})
          PERMUTE EXPAND FACTORS (c{2},D{3}) )
```

2.10.2 Excluding Portions of the pattern

For ALL ROWS PER MATCH option, rows matching a portion of the PATTERN may be excluded from the output. The excluded portion is bracketed by ‘{-’, ‘-}’ inside the PATTERN clause. In the following example we find the longest periods of increasing prices that start with a price no less than 10.

```
SELECT T.Symbol,                    /* row’s symbol */
       T.Tstamp,                    /* row’s time */
       T.Price,                     /* row’s price */
       T.gradient
       T.matchno                    /* row’s classifier */
FROM Ticker MATCH_RECOGNIZE (
            PARTITION BY Symbol
            ORDER BY Tstamp
            MEASURES (LAST(B.price) - A.price)/COUNT(B.*)
                                     AS gradient
                     MATCH_NUMBER    AS matchno
            ALL ROWS PER MATCH
            AFTER MATCH SKIP LAST B
            MAXIMAL MATCH
            PATTERN ( {-A-} B+ {-C-} )
            DEFINE A AS A.price >= 10
                   B AS B.price > PREV(B.price),
                   C AS (C.price <= PREV(C.price)) ) T
```

In the result we will find only rows that match B, i.e. where price was increasing above 10.

The exclusion works on any sub-expression. For example, PATTERN (A (B | {-NOTD-})* D) collects all B's that occur between A and D with NOTD defined to be a negation of D..

The excluded variables can be referenced in the definition of other variables as well as in the definition of exported expressions in the MEASURES clause, for example see gradient measure above.

The exclusion functionality has no effect on the ONE ROW PER MATCH clause since in this case we issue only a single aggregated row for each match.

2.11 SUBSET

The SUBSET component of the syntax is optional. It is used to declare additional variables and define them as unions of variable names in the PATTERN. For example

```
FROM Ticker MATCH_RECOGNIZE
       ( ORDER BY Tstamp
         MEASURES FIRST(X.time) x_firsttime,
                  LAST(Y.time)  y_lasttime,
                  AVG(t.price)  yx_avgprice
         PATTERN (X+ Y+)
         SUBSET T = (X, Y)
         DEFINE X AS X.price > PREV (X.price),
                Y AS Y.price < PREV (Y.price)
       )
```

This example declares an additional variable, T, and defines it as the union of the rows matched by X and the rows matched by Y.

There can be multiple variables defined using SUBSET clause. For example, the following is allowed:

```
PATTERN (W+ X+ Y+ Z+ )
SUBSET A = (X, Y)
       B = (W, Z)
```

The right hand side of a SUBSET item is a comma-separated list of distinct variable names that are found in the PATTERN clause. This defines the variable name on the left hand side as the union of the variable names on the right hand side. Note that defining of subset variables in terms of previously defined subset variables is not allowed since this adds no expressive power.

2.12 DEFINE

DEFINE is a mandatory component, used to specify the boolean condition that defines a variable name that is declared in the pattern. In the example,

```
DEFINE X AS X.price > PREV (X.price),
                    Y AS Y.price < PREV (Y.price)
```

X is defined by the condition X.price > PREV (X.price), and Y is defined by the condition Y.price < PREV (Y.price).

A variable name does not require a definition and if there is no definition, the default is a predicate that is always true. Such a variable name can be used to match any row.

No variable name that is defined by a SUBSET may be defined by PATTERN.

The definitions of variable names may reference the same or other variable names. For example

```
     FROM Ticker MATCH_RECOGNIZE
            ( PARTITION BY symbol
              ORDER BY tstamp
              MEASURES FIRST(a.time) a_firsttime,
                       LAST(d.time)  d_lasttime,
                       AVG(b.price)  b_avgprice,
                       AVG(d.price)  d_avgprice
              PATTERN ( A B+ C+ D )
              DEFINE A AS A.price > 100,
                     B AS B.price > A.price,
                     C AS C.price < AVG (B.price),
                     D AS D.price > PREV(D.price)
            )
```

This example illustrates how one variable name can be defined in terms of earlier variable name(s).

1. A defines a single row, so the definition of B can reference a column (A.price) of that single row.
2. B, on the other hand, defines one or more rows, so it is not possible in the definition of C to reference an individual column. Instead only aggregates of B are allowed in the definition of C.
3. The definition of D illustrates how a variable name might be defined in terms of itself, in which case references to columns rather than aggregates are allowed. The operator PREV gives access to the previous row in the partition.

2.12.1 Singletons and group variables

The above example motivates the following distinction between singleton and group variables.

> A variable is a singleton if it occurs exactly once in a pattern, is not defined by a SUBSET and is not in the scope of an alternation or quantification.
>
> Otherwise, a variable is a group.

Columns of singleton and group variables can be referenced in the DEFINED and MEASURES clauses using these rules.

> If a variable is a singleton, then only individual columns may be referenced, not aggregates.
>
> If a variable is a group, then:
>
> - if the variable is used to reference a column, then the last row to match the variable is used. If the variable is not yet matched, the column reference returns null. If the variable is being referenced in a definition of the same variable, then the column of the current row is returned.
> - if the variable is used in an aggregate, then the aggregate is performed over all rows that have matched the variable so far. If desired, we can construe this as providing running aggregates with no special syntax, when a variable is referenced in an aggregate in its own definition, or we can continue to require special syntax to highlight that a running aggregate is meant.

Examples:

1. Find a zigzag pattern in which a price alternately goes up and down, for as long a run as possible.

```
PATTERN (UP DOWN)+
DEFINE UP   AS COUNT (UP.*) = 1 OR UP.Price > DOWN.Price,
       DOWN AS DOWN.Price < UP.Price
```

   How this works: UP will match any row as the first row, because COUNT(UP.*) = 1 is always true for the first row of a candidate match. After picking a candidate first row, then DOWN will match the row if the Price has declined. After picking the second row, UP will match the third row if the Price has increased over the second row.

2. If you don’t care whether the zigzag starts by going from high to low, or ends by going from low to high, you could use

```
PATTERN DOWN? (UP DOWN)+ UP?
DEFINE UP   AS COUNT (UP.*) = 1   OR UP.Price > DOWN.Price,
       DOWN AS COUNT (DOWN.*) = 1 OR DOWN.Price < UP.Price
```

   With this pattern, the first row might match UP or DOWN. If it matches DOWN, we need the COUNT (DOWN.*) = 1 to watch for the first row.

3. A uniform way to recognize the first row is to define a subset for all rows:

```
PATTERN DOWN? (UP DOWN)+ UP?
SUBSET A = (UP, DOWN)
DEFINE UP   AS COUNT (A.*) = 1 OR UP.Price > DOWN.Price,
       DOWN AS COUNT (A.*) = 1 OR DOWN.Price < UP.Price
```

If we follow this path, then a variable's definition might contain both aggregate references and column references to the same variable. In the example above, we see both COUNT(UP.*) and UP.Price. The aggregate is computed over all rows that match the variable (including the current row if relevant). Individual columns refer to the last row that matched the variable (possibly the current row).

2.12.2 Aggregates and Running aggregates

Aggregates with multiple arguments must operate on the same variable, for example

```
MEASURES xy_corr AS CORR(X.price, Y.price)
PATTERN (X+,Y+)
DEFINE X AS x.price <= 10, Y AS y.price>10
```

is illegal since CORR operates on two variables X and Y. The group Y may have different number of rows that X, so there cannot be a pairing of Y with X. We could allow singleton variables to appear in such cases but for consistency with group variables we prohibit that. This applies not only to the MEASURES but DEFINE as well.

According to the Section "Singletons and group variables" on page 15., while defining a variable we can reference variables that have not been matched yet. Hence,

```
PATTERN (X+,Y+)
DEFINE X AS COUNT(y.*) >= 10,
       Y AS y.price > 10
```

legal, but will never be matched since it at the time that X is matched, Y has not yet been matched and COUNT(y.*) is 0.

Section "Singletons and group variables" on page 15. states that all aggregates on a variable X in the DEFINE clause are in effect running aggregates, i.e., they aggregate matches of X up to the current row. If the variable X have been completely matched so far, then the aggregate if final, otherwise it is running. Users should be aware of the running nature of the aggregates, since they can arise stealthily using the SUBSET clause, for example:

```
PATTERN (X+ Y+)
SUBSET Z = (X, Y)
DEFINE X AS X.Price > 100,
       Y AS SUM(Z.Price) < 1000
```

Observe that since variable Z involves Y hence, the definition of Y involves a running aggregate on Y.

Aggregates in the DEFINE clause can either reference variables or columns not qualified by the variables names. In the latter case, the aggregate is a running aggregate and includes all preceding rows and the current row. For example,

```
PATTERN (UP DOWN)+
DEFINE UP   AS COUNT (UP.*) = 1 OR UP.Price > DOWN.Price,
       DOWN AS DOWN.Price < UP.Price AND COUNT(*) > 1000
```

finds a zigzag pattern in which a price alternately goes up and down, for as long a run as possible but larger than 1000 rows.

Observe that aggregates in the MEASURES clause a final since the are a running aggregates computed over the the final match.

2.13 Match recognition

The rows of the input table are partitioned according to the PARTITION BY component and each partition is ordered by the ORDER BY component. Within an ordered partition, a match is a contiguous subsequence of rows S = { R_1, ..., R_n } and a function f which maps S to the variable names in the pattern, and meeting specific conditions to be enumerated. Note that the function f maps S to a sequence { $f(R_1)$, $f(R_2)$, ... , $f(R_n)$ } which is a sequence of variable names.

If C is a variable name, S is a contiguous subsequence and f is a function from S to variable names, then $f^{-1}(C)$ is defined as the set of rows R_i such that $f(R_i) = C$.

The conditions on S and f are as follows:

1. For all i, R_i satisfies the condition that defines the variable name $f(R_i)$. When testing whether R_i satisfies this condition, variable names are interpreted as follows:

   a) The variable name $f(R_i)$ references the current row, or, using PREV, a previous row.

   b) A variable name C that is a singleton

   c) A variable name C that is a group variable references, via an aggregate, the set of rows $f^{-1}(C)$ that are prior to the current row, i.e., all rows R_j prior to the current row (i.e., $j < i$) such that $f(R_j) = C$.

   d) A variable name C that is a group variable to $f(R_i)$ references the last row R_j prior to the current row (i.e., $j < i$) such that $f(R_j) = C$. If R_j does not exist, references to C result in null.

2. The sequence { $f(R_1)$, ... , $f(R_n)$ } is a member of the language generated by the pattern. The language of a pattern is defined as follows. Let P be a pattern; its language L(P) is defined recursively as follows:

   a) If P is a single variable name C, then L(P) = { { C } }

   b) If P is of the form (Q), then L(P) = L(Q).

   c) If P is of the form Q_1Q_2 then L(P) = { { A_1 A_2 } | A_1 is in $L(Q_1)$ and A_2 is in $L(Q_2)$ }.

   d) If P is of the form Q? or Q??, then L(P) = L (Q{0,1}).

e) If P is of the form Q* or Q*?, then L(P) = L (Q{0,}).

f) If P is of the form Q+ or Q+?, then L(P) = L (Q{1, }).

g) If P is of the form Q{n,} or Q{n,}?, then L(P) = { { A_1 A_2 ... A_k } | k >= n and for all i, 1 <= i <= k, A_i is in L(Q) }

h) If P is of the form Q{n,m} or Q{n,m}?, then L(P) = { { A_1 A_2 ... A_m } | n <= k <= m and for all i, 1 <= i <= k, A_i is in L(Q) }

If ALL MATCHES is specified, then all matches are retained as the result. Otherwise if MAXIMAL MATCH is specified, then the matches to a pattern in a partition are ordered by preferment. Preferment is given to matches based on the following priorities:

1. A match that begins at an earlier row is preferred over a match that begins at a later row.
2. Of two matches matching a greedy quantifier, the longer match is preferred.
3. Of two matches matching a reluctant quantifier, the shorter match is preferred.
4. Of two matches matching an alternation, we need to decide whether to prefer the longer match (POSIX-style), or the one that matches the lexically prior alternative (Perl-style, I think). My impression from [Friedl] is that Perl-style is easier to implement. I think there are tricky use cases in which POSIX alternation is desirable, and other tricky use cases in which Perl alternation is desirable. Conceivably we could decide between the two by looking at such use cases. Or we could let the user decide with syntax. Note that the character string regular expression operator defined in [WLG-027] used Perl alternation.

After ranking matches by preferment, matches are chosen as follows:

1. The first match by preferment is taken.
2. The pool of matches is reduced as follows based on the SKIP TO clause:

   a) If SKIP PAST LAST ROW is specified, all matches that overlap the first match are discarded from the pool.

   b) If SKIP TO NEXT ROW is specified, then all matches that overlap the first row of the first match are discarded.

   c) If SKIP TO FIRST <variable> is specified, then all matches that overlap any row except the first match to <variable> of the first match are discarded

   d) If SKIP TO LAST <variable> is specified, then all matches that overlap any row except the last match to <variable> of the first match are discarded
3. The first match by preferment of the ones remaining is taken.
4. Step 2 is repeated to remove more matches from the pool.
5. Steps 3 and 4 are repeated until there are no remaining matches in the pool.

2.14 CLASSIFIER

This component is only available with ALL ROWS PER MATCH. It is used to specify the name of a character string column, called the classifier column. In each row of output, the classifier column is set to the variable name in the PATTERN that the row matched.

2.15 MATCH_NUMBER

Matches within a partition are numbered sequentially starting with 1 in the order they are chosen in the previous section.

The MATCH_NUMBER component is used to specify a column name for an extra column of output from the MATCH_RECOGNIZE construct. The extra column is an exact numeric with scale 0, and provides the MATCH_NUMBER within a partition, starting with 1 for the first match, 2 for the second, etc.

2.16 Aggregate extensions for row pattern recognition

A number of extensions to aggregates will be useful with MATCH_RECOGNIZE:

2.16.1 FIRST and LAST

FIRST and LAST are aggregates that are only available for group variable names. FIRST returns the value of the first row of the group in the order defined by ORDER BY, and LAST returns the value of the last row of the group.

FIRST and LAST functions can accept and optional non-negative integer argument indicating the offset following the first and offset preceding the last row of the variable correspondingly. That argument must be a constant so during execution we know what offsets to track. If that offset does not fall within the match of the variable, or falls outside of the table, null value is returned.

2.16.2 Running aggregates in the MEASURES clause

All aggregates within the scope of MATCH_RECOGNIZE clause are running aggregates. Since the MEASURES clause is evaluated after the match has been found, the running aggregates there are in effect final.

However, in applications that use ALL ROWS PER MATCH, it can be convenient to return not only the final aggregates, but running aggregates over matched variables as well. We provide that using a notation of a cumulative aggregate borrowed from WINDOW functions:
<aggregate_name> (<arguments>) OVER (ORDER BY). The ORDER BY clause indicate that the cumulative aggregate is performed over a sequence of rows ordered by according to the ORDER BY clause of MATCH_RECOGNIZE. The aggregate can be over a variable and this will aggregate all matches of the variable up to the current point, or can be on a column of the source table and it will aggregate the column up to the current row. For example,

```
SELECT T.Symbol,                     /* row’s symbol */
       T.Tstamp,                     /* row’s time */
```

```
            T.Price,                     /* row's price */
            T.gradient
            T.matchno                    /* row's classifier */
     FROM Ticker MATCH_RECOGNIZE (
                 PARTITION BY Symbol
                 ORDER BY Tstamp
                 MEASURES (LAST(B.price) - A.price)/COUNT(B.*)
                                         AS gradient,
                          AVG(B.price)   AS final_avg_price,
                          AVG(B.price)   AS running_avg_price,
                          MATCH_NUMBER   AS matchno
                 ALL ROWS PER MATCH
                 AFTER MATCH SKIP LAST B
                 MAXIMAL MATCH
                 PATTERN ( {-A-} B+ {-C-} )
                 DEFINE A AS A.price >= 10
                        B AS B.price > PREV(B.price),
                        C AS (C.price <= PREV(C.price)) ) T
```

2.17 PREV function

The PREV function may be used to access columns of the previous row of a variable name. If there is no previous row, the null value is returned.

The PREV function can accept an optional non-negative integer argument N indicating the offset to the previous rows. That argument must be a constant. In this case, we return column from the N-th row preceding the current row, and if the row doesn't exist, we return NULL.

This function can access variables currently defined, for example

```
Y AS Y.price < PREV (Y.price, 2)
```

is legal, while

```
Y AS Y.price < PREV (X.price, 2)
```

is not since it while defining variable Y it references variable X inside the PREV function.

3. Pattern recognition in WINDOW clause

This section proposes to enhance the WINDOW clause to include pattern matching. In [Foundation:2003], a window is a data structure defined on the result of FROM...WHERE...GROUP BY... HAVING... clauses, which we will call the underlying query, producing a table T. This data structure does three things:

1. Partitions the rows of T according to zero or more columns
2. Within each partition, orders the rows of T according to zero or more expressions

3. For each row R in a partition, defines a “window frame”, WF(R), which is some contiguous subset of the ordered partition, whose endpoints may be the beginning or end of the partition, or may be defined relative to the row using either a physical offset (row count) or a logical offset (a value added to (or subtracted from) the only sort column).

We introduce pattern recognition to further reduce the window frame. Pattern recognition will be applicable only to the window frames that follow the current row R. Lets call the window frame coming out of step 3 the full window frame WF(R), and the window frame after reduction by pattern matching the reduced window frame WFP(R). Note that the full window frame might be as large as the entire partition, so this formulation permits reducing the window frame based on the pattern alone.

Observe that the pattern to be recognized can either be the match the first row of the full window frame, or can start at the first match within it. To require that the match begin with the first row of the full window frame, we use INITIAL PATTERN, i.e, WFP(R) is the match within WF(R) whose first row is R, if any. To permit a search for the first match within the window frame, we use SEEK PATTERN. Here WFP(R) is the first match to the pattern within WF(R), if any. The proposed keywords { INITIAL | SEEK } go before the PATTERN keyword. The default for the window frame is INITIAL PATTERN.

In the event that there is no match, the reduced window frame is empty.

The options { ONE ROW | ALL ROWS } PER MATCH do not apply to window patterns. This is because the design of the [Foundation:2003] windows is that all rows of the underlying table emerge from the WINDOW clause. In effect, the WINDOW clause tags the output rows with additional information (the window frame of each row of T). Unmatched rows are still in the output, though their window frame size is 0. The MATCH_RECOGNIZE proposal of Section “Pattern Recognition Over Table Expression” on page 2., on the other hand, completely suppresses unmatched rows.

Similarly, the INCREMENTAL MATCH option does not apply to windows with frames following the current row. This option finds multiple matches starting from the current row and outputs them. This would imply that a single row could have multiple window frames rooted at it, and hence for a single row we would output multiple rows as a result violating the semantics of the existing window clause. Suppose that we have some other window W (not necessary related to patterns) in addition to a window WP with a pattern defined in the query block. Then results can be different depending on their order of evaluation, hence we would have to define priority of their evaluation. This is a violation of current window function semantics.

The MATCH_NUMBER feature in Oracle’s proposal seems to have no place in the WINDOW clause.

The output of a WINDOW W clause can be used in the SELECT list to form window functions using OVER W. Window functions are aggregates that are computed using the window frame of the current row.

The MATCH_RECOGNIZE of Section “Pattern Recognition Over Table Expression” on page 2. introduced MEASURES clause that allows us to name and export expressions on singleton variables and aggregates on group variables. We propose that the measures be referencable in the window functions using <measure_name> OVER W notation. Hence we propose extending the visibility rules for a window W. The columns visible to W are all olumns defined in the query block as well asall columns defined in the MEASURES clause of W. The values for columns defined in the MEASURES clause are the same as for ALL ROWS PER MATCH option of MATCH_RECOGNIZE clause on a table expression.

We note that pattern variables are not directly available to the window functions on the SELECT list. For example,

```
SELECT AVG(Y.price) OVER W
```

or

```
SELECT X.price OVER W
```

is illegal. Instead, users have to define exported variables in the MEASURES clause and use them as arguments to the OVER W clause, for example:

```
SELECT sum_yprice OVER W, x_time OVEER W, AVG(Y.Price)
FROM T
WINDOW W AS (PARTITION BY .. ORDER BY..
             MEASURES SUM(Y.price) AS sum_yprice
                      x.time       AS x_time
             (PATTERN (X Y+ Z)...))
```

As for the CLASSIFIER feature, note that a full window frame does not necessarily start with the current row. It may be of interest to know how the current row is classified by the pattern recognition. In the event that the current row is not part of the match, the classifier column would be null.

Note that a match might occur but not include the current row. This situation can be detected by using COUNT(S.*) where S is some pattern variable that must have at least one row. If there is no variable in the PATTERN that is mandatory (for example, they are all quantified with *), one can still use SUBSET to define a variable that is the union of the entire match. By definition, a match must have at least one row. We note again that COUNT(S.*) should be defined in the MEASURES clause and the used in the window function.

An example illustrating the complete set of capabilities:

```
SELECT T.Symbol,                    /* row’s symbol */
       T.Tstamp,                    /* row’s time */
       T.Price,                     /* row’s price */
       a_tstamp       OVER W,       /* start time */
       a_price        OVER W,       /* start price */
       max_c_tstamp   OVER W,       /* inflection time */
       lasc_c_price   OVER W,       /* low price */
       max_f_tstamp   OVER W,       /* end time */
       last_f_price   OVER W        /* end price */
       classy         OWER W        /* classifier */
```

```
FROM Ticker AS T
WINDOW W AS ( PARTITION BY Symbol
              ORDER BY Tstamp
              MEASURES A.Tstamp       AS a_tstamp,
                       A.price        AS a_price,
                       MAX(C.Tstamp)  AS max_tstamp,
                       LAST(C.Price)  AS last_c_price,
                       MAX(F.Tstamp)  AS max_f_tstamp,
                       LAST(F.Price)  AS last_f_price,
                       CLASSIFIER     AS classy
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              INITIAL PATTERN (A B C* D E* F+)
              DEFINE /* A defaults to True */
                   B AS (B.price < PREV(B.price)),
                   C AS (C.price <= PREV(C.price)),
                   D AS (D.Price > PREV(D.price)),
                   E AS (E.Price >= PREV(E.Price)),
                   F AS (F.Price >= PREV(F.price)
                         AND F.price > A.price)
              )
```

The MEASURES clause is evaluated before any references to the window functions in the SELECT list and its values are the same as with MATCH_RECOGNIZE with ALL ROWS PER MATCH option.

To be complete we define the semantics of the SKIP TO when used for windows. Let R, WF(R) and WFP(R) be, as before, the current row and its full and reduced window frame correspondingly. Suppose that the SKIP TO specification identifies row RN as the salient point at which the next pattern match should start. Then the rows between R and RN will have empty windows in the result and the pattern matching resumes at row RN -- i.e., we start looking for a match within the window frame associated with row RN.

OPEN ISSUE. Should we make the pattern variables visible to the window functions in the SELECT list. Window clause is the last clause in a query block, thus all joins, (correlated) predicates and aggregates have been evaluated before it, hence the rows covered by the variables are established only by the matching rules and not by other operations in the query block (like joins or aggregations). If a window has a deterministic ORDER BY clause, then window functions over the variables are deterministic as well (our pattern match is deterministic), thus there is no semantic reason to exclude them. We disallowed that to be consistent with the previous mode of MATCH_RECOGNIZE as a qualifier of a table expression.

4. Extensions discussed but not yet agreed upon

4.1 IBM additional extensions to the window clause

4.1.1 INCREMENTAT_MATCH in window functions

IBM proposed to permit the INCREMENTAL MATCH in the window definition. In that case, a single row R could have multiple window frames WFP(R) attached to it: $\text{WFP(R)}_{1}..\text{WFP(R)}_{n}$. For each WFP(R)_{i} we would emit one row of the result.

We have not included this it in the main proposal since it violates the semantic of the window functions requiring that for each row of the input we calculate a single WPR(R) and emit a single row based on it. Further more, accepting INCREMENTAT_MATCH, we have a problem of ordering of window functions on different windows. Suppose there are two windows W1 and W2 defined in a query block, both with INCREMENTAL_MATCH. Then the results will depend on the ordering of the window frame computations. If W1 is evaluated first, then W2 will see the additional rows produced by W1 and will run its computation on these rows as well. If W2 is evaluated first, then W1 will see the additional rows produced by W1. Results in these two executions will be different. Another issue is usage of window functions with window definitions in-line on the SELECT list. Functions should not produce new rows, and INCREMENTAL_MATCH would in this case.

4.1.2 EXCLUDE/INCLUDE in the window functions

In order to make windows return similar results to MATCH_RECOGNIZE with ONE ROW PER MATCH option, IBM proposed to optionally exclude rows with empty widow frames. This is governed by a new clause [INCLUDE | EXCLUDE] EMPTY WINDOW. The clause can operate either on an individual window, or on the entire query. In the former case (clause on window level), EXCLUDE on a window W1 will exclude a row with an empty window frame even if there is another non-empty window frame rooted at R resulting from another window W2 != W1. In the latter case (the clause on query block level), EXCLUDE will exclude the row only if all window frames resulting from all window clauses are empty.

We have not included this in the proposal since it violates the existing semantic of windows.

Users can emulate the EXLUDE behavior but placing COUNT(*) OVER W in the SELECT list, and then in the outer query filter rows whose count is zero.

5. Proposal

5.1 Changes to 7.6 <table reference>

```
<table reference> ::=
        <table factor>
      | <joined table>
```

```
<table factor> ::=
     <table primary> [ <sample clause> ]

<sample clause> ::= . . .

<table primary> ::=
       <table or query name> [ ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
       [ <left paren> <derived column list> <right paren> ]]
     | <derived table> ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
       [ <left paren> <derived column list> <right paren> ]
     | <lateral derived table> ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
       [ <left paren> <derived column list> <right paren> ]
     | <collection derived table> ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
        [ <left paren> <derived column list> <right paren> ]
     | <table function derived table>
       ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
       [ <left paren> <derived column list> <right paren> ]
     | <only spec> [ ~~[ AS ] <variable name>~~
       **<as variable or recognition>**
       [ <left paren> <derived column list> <right paren> ]]
     | <parenthesized joined table>

**<as variable or recognition> ::=**
       **[ AS ] <variable name>**
     **| <row pattern recongition clause>**

**<row pattern recognition clause> ::=**
     **MATCH_RECOGNIZE <left paren>**
     **[ <row pattern partition by> ]**
     **[ <row pattern order by> ]**
     **[ <row pattern order by> ]**
     **[ <row pattern measures> ]**
     **[ <row pattern rows per match> ]**
     **[ <row pattern skip to> ]**
     **[ <row pattern match specification> ]**
     **PATTERN <left paren> <row pattern> <right paren>**
     **[ <row pattern subset clause> ]**
     **DEFINE <row pattern definition list>**
     **[ <row pattern classifier> ]**
     **[ <row pattern MATCH_NUMBER> ]**
     **[ <row pattern aggregates> ]**
     <right paren>

<row pattern partition by> ::=
     PARTITION BY <row pattern partition list>

<row pattern partition list> ::=
     <row pattern partition column>
     [ { <comma> <row pattern partition column> }... ]

<row pattern partition column> ::=
     <column reference> [ <collate clause> ]

<row pattern order by> ::=
     ORDER BY <sort specification list>

<row pattern measures> ::=
     MEASURES <row pattern measure list>

<row pattern measure list> ::=
     <row pattern measure column>
     [ { <comma> <row pattern measure column> }... ]

<row pattern measure column> ::=
     <column name> AS <expression>

<row pattern rows per match> ::=
       ONE ROW PER MATCH
     | ALL ROWS PER MATCH

<row pattern match qualifier> ::=
       SKIP TO NEXT ROW
     | SKIP PAST LAST ROW
     | SKIP TO FIRST <variable name>
     | SKIP TO LAST <variable name>
     | SKIP TO <variable name> }

<row pattern match specification> ::=
       INCREMENTAL MATCH
     | MAXIMAL MATCH

<row pattern> ::=
       <row pattern term>
     | <row pattern> <vertical bar> <row pattern term>

<row pattern term> ::=
       <row pattern factor>
     | <row pattern term> <row pattern factor>

<row pattern factor> ::=
     <row pattern primary> [ <row pattern quantifier> ]
```

```
<row pattern quantifier> ::=
         <asterisk> [ <question mark> ]
       | <plus sign> [ <question mark> ]
       | <question mark> [ <question mark> ]
       | <left brace> [ <unsigned integer> ]
         <comma> [ <unsigned integer> ] <right brace>
         [ <question mark> ]

<row pattern primary> ::=
         <variable name>
       | <left paren> <row pattern> <right paren>

<row pattern subset> ::=
     SUBSET <row pattern subset list>

<row pattern subset list> ::=
     <row pattern subset item>
     [ { <comma> <row pattern subset item> }... ]

<row pattern subset item> ::=
     <variable name> <equals operator> <left paren>
     <row pattern subset rhs> <right paren>

<row pattern subset rhs> ::=
     <variable name>
     [ { <comma> <variable name> }... ]

<row pattern definition list> ::=
     <row pattern definition>
     [ { <comma> <row pattern definition> }... ]

<row pattern definition> ::=
     <variable name> AS <search condition>

<row pattern classifier> ::= CLASSIFIER <column name>

<row pattern MATCH_NUMBER> ::= MATCH_NUMBER <column name>

<row pattern aggregates> ::=
     AGGREGATES <row pattern aggregate list>

<row pattern aggregate list>  ::=
     <row pattern aggregate definition>
     [ { <comma> <row pattern aggregate definition> } ... ]

<row pattern aggregate definition>  ::=
     <value expression> AS <column name>
```

Syntax Rules

General Rules

Conformance Rules

5.2 Changes to 7.11 <window clause> — based on IBM's ideas

1.

```
<window specification details> ::=
     [ <existing window name> ]
     [ <window partition clause> ]
     [ <window order clause> ]
     [ <window frame clause> ]
     [ <window pattern recognition clause> ]

<window pattern recognition clause>  ::=
     MATCH_RECOGNIZE <left paren>
     [ <row pattern partition by> ]
     [ <row pattern order by> ]
     [ <row pattern measures> ]
     <row pattern initial or seek>
     PATTERN <left paren> <row pattern> <right paren>
     [ <row pattern subset clause> ]
     DEFINE <row pattern definition list>
     [ <row pattern classifier> ]
     [ <row pattern aggregates> ]
     <right paren>
```

[NOTE to the proposal reader: <row pattern rows per match> is missing; <row pattern initial or seek> is new; <row pattern MATCH_NUMBER> is missing. These are the three syntactic differences from Oracle's proposal.]

```
<row pattern initial or seek> ::=
       INITIAL
     | SEEK
```

- End of paper -