TECHNICAL REPORT

ISO/IEC TR 19075-5

First edition
2016-12-15

Information technology — Database languages — SQL Technical Reports —

Part 5: Row Pattern Recognition in SQL

Technologies de l'information — Langages de base de données — SQL Rapport techniques —

Partie 5: Reconnaissance de formes de lignes dans SQL

Reference number
ISO/IEC TR 19075-5:2016(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Tables

Table

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 32, *Data management and interchange*.

A list of all the parts in the ISO 19075 series, can be found on the ISO website.

Introduction

This Technical Report discusses the syntax and semantics for recognizing patterns in rows of a table, as defined in [ISO9075-2].

[ISO9075-2] defines two features regarding row pattern recognition:

— Feature R010, “Row pattern recognition: FROM clause”

— Feature R020, “Row pattern recognition: WINDOW clause”

These two features have considerable syntax and semantics in common, the principle difference being whether the syntax is placed in the FROM clause or in the WINDOW clause.

The organization of this Technical Report is as follows:

1) Clause 1, “Scope”, specifies the scope of this Technical Report.

2) Clause 2, “Normative references”, identifies standards that are referenced by this Technical Report.

3) Clause 3, “Row pattern recognition: FROM clause”, discusses Feature R010, “Row pattern recognition: FROM clause”.

4) Clause 4, “Expressions in MEASURES and DEFINE”, discusses scalar expression syntax in row pattern matching.

5) Clause 5, “Row pattern recognition: WINDOW clause”, discusses Feature R020, “Row pattern recognition: WINDOW clause”. Clause 5, “Row pattern recognition: WINDOW clause”, does not duplicate material already presented in Clause 3, “Row pattern recognition: FROM clause” and Clause 4, “Expressions in MEASURES and DEFINE”, which should be read even if the reader is only interested in Feature R020, “Row pattern recognition: WINDOW clause”.

6) Clause 6, “Pattern matching rules”, discusses the formal rules of pattern matching.

Information technology — Database languages — SQL Technical Reports —

Part 5:
Row Pattern Recognition in SQL

1 Scope

This Technical Report discusses the syntax and semantics for recognizing patterns in rows of a table, as defined in [ISO9075-2].

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 ISO and IEC standards

[ISO9075-2] ISO/IEC 9075-2:2016, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*.

(Blank page)

3 Row pattern recognition: FROM clause

Feature R010, “Row pattern recognition: FROM clause” of [ISO9075-2] enhances the capability of the FROM clause with a MATCH_RECOGNIZE clause to specify a row pattern. The syntax and semantics of a row pattern is discussed through examples presented throughout this Clause of this Technical Report.

There are two principal variants of the MATCH_RECOGNIZE clause:

1) ONE ROW PER MATCH, which returns a single summary row for each match of the pattern (the default).

2) ALL ROWS PER MATCH, which returns one row for each row of each match. There are three suboptions, to control whether to also return empty matches or unmatched rows.

3.1 Example of ONE ROW PER MATCH

The following example illustrates MATCH_RECOGNIZE with the ONE ROW PER MATCH option. Let Ticker (Symbol, Tradeday, Price) be a table with three columns representing historical stock prices. Symbol is a character column, Tradeday is a date column, and Price is a numeric column.

NOTE 2 — All examples in this Technical Report use mixed-case identifiers for the names of tables, columns, *etc.*, whereas SQL key words are shown in uppercase. Unquoted identifiers are actually equivalent to uppercase, so the column headings of sample results will be shown with the identifiers converted to uppercase.

It is desired to partition the data by Symbol, sort it into increasing Tradeday order, and then detect maximal “V” patterns in Price: a strictly falling price, followed by a strictly increasing price. For each match to a V pattern, it is desired to report the starting price, the price at the bottom of the V, the ending price, and the average price across the entire pattern.

The following query may be used to solve this pattern matching problem:

```
SELECT M.Symbol, /* ticker symbol */
       M.Matchno, /* sequential match number */
       M.Startp, /* starting price */
       M.Bottomp, /* bottom price */
       M.Endp, /* ending price */
       M.Avgp /* average price */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  A.Price AS Startp,
                  LAST (B.Price) AS Bottomp,
                  LAST (C.Price) AS Endp,
                  AVG (U.Price) AS Avgp
         ONE ROW PER MATCH
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A B+ C+)
         SUBSET U = (A, B, C)
```

```
        DEFINE /* A defaults to True, matches any row */
               B AS B.Price < PREV (B.Price),
               C AS C.Price > PREV (C.Price)
    ) AS M
```

In the example above, the principal syntactic elements of MATCH_RECOGNIZE are presented on separate lines. In this example:

— Ticker is the name of the row pattern input table. In this example, the row pattern input table is a table or view. The row pattern input table may also be a derived table (in-line view).

— MATCH_RECOGNIZE introduces the syntax for row pattern recognition.

— PARTITION BY specifies how to partition the row pattern input table. The PARTITION BY clause is a list of columns of the row pattern input table. This clause is optional; if omitted, there are no row pattern partitioning columns, and the entire row pattern input table constitutes a single row pattern partition.

— ORDER BY specifies how to order the rows within row pattern partitions. The ORDER BY clause is a list of columns of the row pattern input table. This clause is optional; if omitted, the order of rows in row pattern partitions is completely non-determinstic. However, since non-deterministic ordering will defeat the purpose of most row pattern recognition, the ORDER BY clause will usually be specified.

— MEASURES specifies row pattern measure columns, whose values are calculated by evaluating expressions related to the match. The first row pattern measure column in this example uses the special nullary function MATCH_NUMBER(), whose value is the sequential number of a match within a row pattern partition. The third and fourth row pattern measure columns in this example use the LAST operation, which obtains the value of an expression in the last row that is mapped by a row pattern match to a row pattern variable. LAST is one of the row pattern navigation operations introduced by [ISO9075-2], discussed in Subclause 4.5, “Row pattern navigation operations”.

  The result of the MATCH_RECOGNIZE clause is called the row pattern output table. When ONE ROW PER MATCH is specified, as in this example, the row pattern output table has one column for each row pattern partitioning column and one column for each row pattern measure column.

— ONE ROW PER MATCH specifies that the row pattern output table will have a single row for each match that is found in the row pattern input table.

— AFTER MATCH SKIP clause specifies where to resume looking for the next row pattern match after successfully finding a match. In this example, AFTER MATCH SKIP PAST LAST ROW specifies that pattern matching will resume after the last row of a successful match.

— PATTERN specifies the row pattern that is sought in the row pattern input table. A row pattern is a regular expression using primary row pattern variables. In this example, the row pattern has three primary row pattern variables (A, B, and C).

— SUBSET defines the union row pattern variable U as the union of the primary row pattern variables A, B, and C.

— DEFINE specifies the Boolean condition that defines a primary row pattern variable; a row must satisfy the Boolean condition in order to be mapped to a particular primary row pattern variable. This example uses PREV, a row pattern navigation operation that evaluates an expression in the previous row. If a primary row pattern variable is not defined in the DEFINE clause, then the definition defaults to a condition that is always true, meaning that any row can be mapped to the primary row pattern variable.

— AS M defines the range variable M to associate with the row pattern output table. This clause is optional; if omitted, then an implementation-dependent range variable is used. Since an implementation-dependent

range variable is unknowable to the query writer, the AS clause should not be omitted if there are any other tables in the FROM clause aside from the MATCH_RECOGNIZE.

The processing of MATCH_RECOGNIZE is as follows:

1) The row pattern input table is partitioned according to the PARTITION BY clause. Each row pattern partition consists of the set of rows of the row pattern input table that are equal (more precisely, not distinct) on the row pattern partitioning columns.

2) Each row pattern partition is ordered according to the ORDER BY clause.

3) Each ordered row pattern partition is searched for matches to the PATTERN.

4) Pattern matching operates by seeking the match at the earliest row, considering the rows in a row pattern partition in the order specified by the ORDER BY. When there is more than one match at a row, then the most preferred match is taken. The precise rules of pattern matching are discussed in Clause 6, “Pattern matching rules”.

5) After a match is found, row pattern matching calculates the row pattern measure columns, which are expressions defined by the MEASURES clause.

6) Using ONE ROW PER MATCH, as shown in the example, row pattern recognition generates one row for each match that is found.

7) The AFTER MATCH SKIP clause determines where row pattern matching resumes within a row pattern partition after a non-empty match has been found. In the example above, row pattern matching resumes at the next row after the rows mapped by a match (AFTER MATCH SKIP PAST LAST ROW).

Here is sample data for one row pattern partition of Ticker, shown sorted according to the ORDER BY clause. The sample data contains two matches to the pattern, indicated by arrows showing the mapping to primary row pattern variables in each match.

Table 1 — Sample Data

SYMBOL	TRADEDAY	PRICE		
XYZ	2009-06-08	50		
XYZ	2009-06-09	60	→ A	⎫
XYZ	2009-06-10	49	→ B	⎪
XYZ	2009-06-11	40	→ B	⎬ first match
XYZ	2009-06-12	35	→ B	⎪
XYZ	2009-06-15	45	→ C	⎭
XYZ	2009-06-16	45		
XYZ	2009-06-17	45	→ A	⎫
XYZ	2009-06-18	43	→ B	⎪
XYZ	2009-06-19	47	→ C	⎬ second match
XYZ	2009-06-22	52	→ C	⎪
XYZ	2009-06-23	70	→ C	⎭
XYZ	2009-06-24	60		

The result of the example for this row pattern partition is:

Table 2 — Results of ONE ROW PER MATCH

SYMBOL	MATCHNO	STARTP	BOTTOMP	ENDP	AVGP
XYZ	1	60	35	45	45.8
XYZ	2	45	43	70	51.4

3.2 Example of ALL ROWS PER MATCH

The previous example can be modified slightly to illustrate ALL ROWS PER MATCH, as follows:

```
SELECT M.Symbol, /* ticker symbol */
       M.Matchno, /* sequential match number */
       M.Tradeday, /* day of trading */
```



```
       M.Price, /* price on day of trading */
       M.Classy, /* classifier */
       M.Startp, /* starting price */
       M.Bottomp, /* bottom price */
       M.Endp, /* ending price */
       M.Avgp /* average price */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  CLASSIFIER() AS Classy,
                  A.Price AS Startp,
                  FINAL LAST (B.Price) AS Bottomp,
                  FINAL LAST (C.Price) AS Endp,
                  FINAL AVG (U.Price) AS Avgp
         ALL ROWS PER MATCH
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A B+ C+)
         SUBSET U = (A, B, C)
         DEFINE /* A defaults to True, matches any row */
                B AS B.Price < PREV (B.Price),
                C AS C.Price > PREV (C.Price)
       ) AS M
```

Note that the second row pattern measure column in this example shows the use of the special function CLASSIFIER(), which returns the name of the row pattern variable to which a row is mapped. CLASSIFIER is discussed in Subclause 4.8, “CLASSIFIER function”.

Here is the result of this query on the sample data:

Table 3 — Results of ALL ROWS PER MATCH

SYMBOL	MATCHNO	TRADEDAY	PRICE	CLASSY	STARTP	BOTTOMP	ENDP	AVGP
XYZ	1	2009-06-09	60	A	60	35	45	45.8
XYZ	1	2009-06-10	49	B	60	35	45	45.8
XYZ	1	2009-06-11	40	B	60	35	45	45.8
XYZ	1	2009-06-12	35	B	60	35	45	45.8
XYZ	1	2009-06-15	45	C	60	35	45	45.8
XYZ	2	2009-06-17	45	A	45	43	70	51.4
XYZ	2	2009-06-18	43	B	45	43	70	51.4
XYZ	2	2009-06-19	47	C	45	43	70	51.4
XYZ	2	2009-06-22	52	C	45	43	70	51.4
XYZ	2	2009-06-23	70	C	45	43	70	51.4

ALL ROWS PER MATCH differs from ONE ROW PER MATCH in the following respects:

1) ALL ROWS PER MATCH returns one row for each row of each match of the pattern.

2) The row pattern output table has a column corresponding to every column of the row pattern input table, not just the row pattern partitioning columns. (Note the column M.Price in the SELECT list. This is a column of the row pattern input table, not a row pattern measure column.)

3) The MEASURES clause supports two semantics for expression evaluation, running semantics and final semantics, indicated by the keywords RUNNING and FINAL.

4) ALL ROWS PER MATCH provides three suboptions for handling empty matches and unmatched rows. These options are not illustrated in this example; see Subclause 3.10.1, “Handling empty matches”, and Subclause 3.10.2, “Handling unmatched rows”, for examples of these options.

3.3 Summary of the syntax

The complete syntax for row pattern recognition in the FROM clause involves the following components:

Table 4 — Row pattern recognition syntax summary

Syntactic component	Optional?	Default	Cross reference
row pattern input table	no	—	Subclause 3.4, “The row pattern input table”
row pattern input name	yes	implementation-dependent	Subclause 3.4.1, “The row pattern input name”
row pattern input declared column list	yes	none	Subclause 3.4.2, “The row pattern input declared column list”
MATCH_RECOGNIZE	no	—	Subclause 3.5, “MATCH_RECOGNIZE”
PARTITION BY	yes	row pattern input table constitutes one row pattern partition	Subclause 3.6, “PARTITION BY”
ORDER BY	yes	non-deterministic ordering in each row pattern partition	Subclause 3.7, “ORDER BY”
MEASURES	yes	none	Subclause 3.9, “MEASURES”
ONE ROW PER MATCH or ALL ROWS PER MATCH	yes	ONE ROW PER MATCH	Subclause 3.10, “ONE ROW PER MATCH *vs.* ALL ROWS PER MATCH”
AFTER MATCH SKIP	yes	AFTER MATCH SKIP PAST LAST ROW	Subclause 3.11, “AFTER MATCH SKIP”

Syntactic component	Optional?	Default	Cross reference
PATTERN	no	—	Subclause 3.12, “PATTERN”
SUBSET	yes	no explicit union row pattern variables	Subclause 3.13, “SUBSET”
DEFINE	no	—	Subclause 3.14, “DEFINE”
row pattern output name	yes	implementation-dependent	Subclause 3.15.1, “Row pattern output name”
row pattern output declared column list	yes	none	Subclause 3.15.2, “Row pattern output declared column list”

3.4 The row pattern input table

The row pattern input table is the input argument to MATCH_RECOGNIZE. In the examples above, the row pattern input table was Ticker, which is a table or view, or perhaps a named query (defined in a WITH clause). The row pattern input table can also be a derived table (also known as in-line view). For example:

```
FROM ( SELECT S.Name, T.Tradeday, T.Price
       FROM Ticker T, SymbolNames S
       WHERE T.Symbol = S.Symbol )
     MATCH_RECOGNIZE ( ... ) AS M
```

The row pattern input table may not be a <joined table>. The work-around is to use a derived table, such as:

```
FROM ( SELECT * FROM A LEFT OUTER JOIN B ON (A.X = B.Y) )
     MATCH_RECOGNIZE (...) AS M
```

Note that column names in the row pattern input table must be unambiguous, since it is impossible to use range variables within the MATCH_RECOGNIZE clause to disambiguate. If the row pattern input table is a base table or a view, this is not a problem, since SQL does not allow ambiguous column names in a base table or view. This is only an issue when the row pattern input table is a derived table.

For example, consider a join of two tables, Emp and Dept, each of which has a column called Name. The following is a syntax error:

```
FROM ( SELECT D.Name, E.Name, E.Empno, E.Salary
       FROM Dept D, Emp E
       WHERE D.Deptno = E.Deptno )
     MATCH_RECOGNIZE (
       PARTITION BY D.Name
       ... )
```

The preceding example is an error because the range variable D is not visible within the MATCH_RECOGNIZE (the scope of D is just the derived table). Rewriting like this is no help:

```
FROM ( SELECT D.Name, E.Name, E.Empno, E.Salary
       FROM Dept D, Emp E
```

```
      WHERE D.Deptno = E.Deptno )
    MATCH_RECOGNIZE (
      PARTITION BY Name
      ... )
```

This rewrite eliminates the use of the range variable D within the MATCH_RECOGNIZE. However, now the error is that Name is ambiguous, because there are two columns of the derived table called Name. The way to handle this is to disambiguate the column names within the derived table itself, like this:

```
FROM ( SELECT D.Name AS DName, E.Name AS EName,
              E.Empno, E.Salary
       FROM Dept D, Emp E
       WHERE D.Deptno = E.Deptno )
     MATCH_RECOGNIZE (
       PARTITION BY DName
       ... )
```

3.4.1 The row pattern input name

Optionally, a correlation name for the row pattern input table may be declared, as in this example (equivalent to the example in Subclause 3.1, “Example of ONE ROW PER MATCH”):

```
SELECT M.Symbol,      /* ticker symbol */
       M.Matchno,     /* sequential match number */
       M.Startp,      /* starting price */
       M.Bottomp,     /* bottom price */
       M.Endp,        /* ending price */
       M.Avgp         /* average price */
FROM Ticker AS T
       MATCH_RECOGNIZE (
         PARTITION BY T.Symbol
         ORDER BY T.Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  A.Price AS Startp,
                  LAST (B.Price) AS Bottomp,
                  LAST (C.Price) AS Endp,
                  AVG (U.Price) AS Avgp
         ONE ROW PER MATCH
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A B+ C+)
         SUBSET U = (A, B, C)
         DEFINE /* A defaults to True, matches any row */
                B AS B.Price < PREV (B.Price),
                C AS C.Price > PREV (C.Price)
       ) AS M
```

The row pattern input name in this example is T, as defined by the syntax “Ticker AS T”. It is also possible to omit the noise word AS, like this: “Ticker T”.

Specifying the row pattern input name is optional. The examples in Subclause 3.1, “Example of ONE ROW PER MATCH”, and Subclause 3.2, “Example of ALL ROWS PER MATCH”, do not show an explicit row pattern input name.

When the row pattern input name is not specified, the following defaults apply:

1) If the row pattern input table is a base table, view, or query name (the name of a query defined in a WITH clause), then the table name, view name or query name is the default row pattern input name.

2) Otherwise, an implementation-dependent row pattern input name, different from any other range variable in the query, is implicit. In practice, this means that the row pattern input name is unknowable and cannot be referenced elsewhere in the query.

The scope of the row pattern input name is the PARTITION BY and ORDER BY clauses of the MATCH_RECOGNIZE clause. This means that the row pattern input name can be used in the following contexts:

1) To qualify column names in the PARTITION BY clause.

2) To qualify column names in the ORDER BY clause.

The example above illustrates both of these uses.

The row pattern input name cannot be referenced in the MEASURES or DEFINE clauses, nor elsewhere in the query, such as the WHERE clause or the SELECT list.

3.4.2 The row pattern input declared column list

If an explicit row pattern input name is specified, it may be followed by a parenthesized list of column names, as in this example:

```
SELECT M.Sym,        /* ticker symbol */
       M.Matchno,    /* sequential match number */
       M.Startp,     /* starting price */
       M.Bottomp,    /* bottom price */
       M.Endp,       /* ending price */
       M.Avgp        /* average price */
FROM Ticker AS T (Sym, Td, Pr)
       MATCH_RECOGNIZE (
         PARTITION BY T.Sym
         ORDER BY T.Td
         MEASURES MATCH_NUMBER() AS Matchno,
                  A.Pr AS Startp,
                  LAST (B.Pr) AS Bottomp,
                  LAST (C.Pr) AS Endp,
                  AVG (U.Pr) AS Avgp
         ONE ROW PER MATCH
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A B+ C+)
         SUBSET U = (A, B, C)
         DEFINE /* A defaults to True, matches any row */
                B AS B.Pr < PREV (B.Pr),
                C AS C.Pr > PREV (C.Pr)
       ) AS M
```

The parenthesized list of column names (Sym, Td, Pr) is called the row pattern input declared column list. The row pattern input declared column list may be used to change the names of the columns of the row pattern input table. There must be exactly the same number of column names in the list as there are columns in the row pattern input table. In this example, Symbol has been renamed to Sym, Tradeday has been renamed to Td, and Price has been renamed to Pr. Consequently, the columns cannot be referenced as Symbol, Tradeday, or Price within the MATCH_RECOGNIZE; instead, they must be referenced by their new names, Sym, Td, and Pr. Note that

this also changes the default names of the columns in the row pattern output table. Thus, in the SELECT list, the first item must be M.Sym, because the input column names Symbol was renamed to Sym, which becomes the name of the corresponding output column.

3.5 MATCH_RECOGNIZE

MATCH_RECOGNIZE is the keyword that introduces the syntax for row pattern recognition in the FROM clause. Syntactically, MATCH_RECOGNIZE is a postfix operator following the row pattern input table. The MATCH_RECOGNIZE keyword is followed by a parenthesized list of syntactic components that collectively describe the row pattern recognition operation.

3.6 PARTITION BY

PARTITION BY is used to specify that the rows of the row pattern input table are to be partitioned by one or more columns. Note that the column names in the PARTITION BY may be unqualified, or they may be qualified by the row pattern input name. See the examples in Subclause 3.4.1, “The row pattern input name”.

If there is no PARTITION BY, then all rows of the row pattern input table constitute a single row pattern partition.

3.7 ORDER BY

ORDER BY is used to specify the order of rows within a row pattern partition. The ORDER BY clause of a MATCH_RECOGNIZE is similar to the ORDER BY clause of a cursor. As with the PARTITION BY clause, column names may be unqualified, or they may be qualified by the row pattern input name. See the examples in Subclause 3.4.1, “The row pattern input name”.

If the order of two rows in a row pattern partition is not determined by the ORDER BY, then the result of MATCH_RECOGNIZE is non-deterministic.

> NOTE 3 — Syntactically, the row pattern output table is always regarded as non-deterministic because there is no way for the query engine to deduce at compile time whether the ordering is total. This means that MATCH_RECOGNIZE cannot be used in contexts that must be deterministic, such as check constraints and assertions. However, the query author can use ORDER BY to insure that the query is sufficiently deterministic for the author's intended purpose.

3.8 Row pattern variables

Row pattern variables are range variables whose scope is limited to a MATCH_RECOGNIZE clause. As range variables, row pattern variables are used to qualify column references, in either the scalar expression of a row pattern measure column, or the Boolean condition of a DEFINE.

There are two kinds of row pattern variables:

1) Primary row pattern variables, which are declared in the PATTERN and defined by an associated Boolean condition specified in the DEFINE clause.

2) Union row pattern variables, which are declared in the SUBSET clause as a union of a list of primary row pattern variables. The primary row pattern variables are called components of the union row pattern variable.

A row pattern variable may not be both a primary row pattern variable and a union row pattern variable. This means that a row pattern variable that is declared in PATTERN may not also be declared on the left hand side of a SUBSET.

Informally, a match consists of a set of contiguous rows in a row pattern partition of the row pattern input table. (For a more formal treatment, see Clause 6, "Pattern matching rules".) Each row of the match is mapped to a primary row pattern variable. The mapping of rows to primary row pattern variables must conform to the regular expression in the PATTERN clause, and is further constrained to insure that all Boolean conditions in the DEFINE clause are true.

Thus rows are mapped to row pattern variables. Conversely, each row pattern variable *RPV* has a set of rows that are mapped to *RPV*. For example, given:

```
PATTERN (A+ (B+ | C+) D)
SUBSET S = (B, D)
```

Suppose that consecutive rows R_3, R_4, R_5, R_6, and R_7 are mapped as follows:

$R_3 \rightarrow$ A

$R_4 \rightarrow$ A

$R_5 \rightarrow$ B

$R_6 \rightarrow$ B

$R_7 \rightarrow$ D

Then:

— the set of rows mapped to A is { R_3, R_4 },

— the set of rows mapped to B is { R_5, R_6 },

— the set or rows mapped to C is empty, and

— the set of rows mapped to D is { R_7 }.

The set of rows mapped to a union row pattern variable *URPV* can be obtained as the set union of rows mapped to each component of *URPV*. In this example:

— the set of rows mapped to S is { R_5, R_6 } ∪ { R_7 } = { R_5, R_6, R_7 }.

There is always one implicit union row pattern variable, called the universal row pattern variable, defined as the union of all primary row pattern variables. Thus, every row of a match is mapped to the universal row pattern variable. The universal row pattern variable is used to implicitly qualify unqualified column reference within the MEASURES or DEFINE clauses. There is no syntax available to the user to denote the universal row pattern variable. The query writer may, of course, define an explicit union row pattern variable that is the union of all primary row pattern variables. (The example in Subclause 3.1, "Example of ONE ROW PER MATCH", illustrates this technique.)

3.9 MEASURES

The MEASURES clause defines row pattern measure columns, which are columns of the row pattern output table whose value is computed by evaluating an expression related to a particular match. Note that this facility extends the scalar expression syntax of [ISO9075-2], and provides special semantics for evaluating scalar expressions in the context of a row pattern match. This is discussed in Clause 4, "Expressions in MEASURES and DEFINE".

NOTE 4 — The MEASURES clause in a window definition does not define columns; instead, it defines named expressions which are accessed using a variant of the window function syntax, called row pattern measure functions. "Row pattern measure" is the generic term for row pattern measure columns and row pattern measure functions, whose values are computed using the same rules.

3.10 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 defined by the PARTITION and MEASURES clauses. This is the default.

ALL ROWS PER MATCH indicates that the result has one row for each row of each match. (It is possible to exclude some rows using the exclusion syntax `{- -}` in the PATTERN; see Subclause 3.12.2, "Excluding portions of the pattern".)

ALL ROWS PER MATCH has three suboptions:

— ALL ROWS PER MATCH SHOW EMPTY MATCHES

— ALL ROWS PER MATCH OMIT EMPTY MATCHES

— ALL ROWS PER MATCH WITH UNMATCHED ROWS

These options are explained in the following subsections.

3.10.1 Handling empty matches

Some patterns permit empty matches. For example:

```
PATTERN (A*)
```

can be matched by zero or more rows that are mapped to A.

An empty match does not map any rows to primary row pattern variables; nevertheless, an empty match has a starting row. For example, there can be an empty match at the first row of a row pattern partition, an empty match at the second row of a row pattern partition, *etc*. An empty match is assigned a sequential match number, based on the ordinal position of its starting row, the same as any other match.

When using ONE ROW PER MATCH, an empty match results in one row of the row pattern output table. The row pattern measures for an empty match are computed as follows:

— The value of MATCH_NUMBER() is the sequential match number of the empty match.

— Any COUNT is 0.

— Any other aggregate, row pattern navigation operation, or ordinary row pattern column reference is null.

For example, the example in Subclause 3.1, "Example of ONE ROW PER MATCH", can be modified to permit empty matches, as follows:

```
SELECT M.Symbol, /* ticker symbol */
       M.Matchno, /* sequential match number */
       M.Firstp,  /* starting price */
       M.Lastp    /* ending price */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  FIRST A.Price AS Firstp,
                  LAST (A.Price) AS Lastp
         ONE ROW PER MATCH
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A*)
         DEFINE
                A AS A.Price > PREV (A.Price)
       ) AS M
```

Here the pattern has been changed to A*, and is used to detect runs of increasing prices. The sample data is now analyzed as follows:

Table 5 — Analysis of sample data permitting empty matches

SYMBOL	TRADEDAY	PRICE		
XYZ	2009-06-08	50		match #1 (empty)
XYZ	2009-06-09	60	→ A	match #2
XYZ	2009-06-10	49		match #3 (empty)
XYZ	2009-06-11	40		match #4 (empty)
XYZ	2009-06-12	35		match #5 (empty)
XYZ	2009-06-15	45	→ A	match #6
XYZ	2009-06-16	45		match #7 (empty)
XYZ	2009-06-17	45		match #8 (empty)
XYZ	2009-06-18	43		match #9 (empty)
XYZ	2009-06-19	47	→ A	⎫
XYZ	2009-06-22	52	→ A	⎬ match #10
XYZ	2009-06-23	70	→ A	⎭

SYMBOL	TRADEDAY	PRICE		
XYZ	2009-06-24	60		match #11 (empty)

The result of the preceding query on the sample row pattern partition is:

Table 6 — Result of query permitting empty matches

SYMBOL	MATCHNO	FIRSTP	LASTP
XYZ	1		
XYZ	2	60	60
XYZ	3		
XYZ	4		
XYZ	5		
XYZ	6	45	45
XYZ	7		
XYZ	8		
XYZ	9		
XYZ	10	47	70
XYZ	11		

In the preceding result, note how the row pattern measures other than the match number are null for empty matches.

As for ALL ROWS PER MATCH, the question arises of whether to generate a row of output for an empty match, seeing that there are no rows in the empty match. To govern this, there are two options:

1) ALL ROWS PER MATCH SHOW EMPTY MATCHES: with this option, any empty match generates a single row in the row pattern output table.

2) ALL ROWS PER MATCH OMIT EMPTY MATCHES: with this option, an empty match is omitted from the row pattern output table. (This may cause gaps in the sequential match numbering.)

ALL ROWS PER MATCH defaults to SHOW EMPTY MATCHES. Using this option, an empty match generates one row in the row pattern output table. In this row:

— The value of a classifier function is null.

— The value of MATCH_NUMBER() is the sequential match number of the empty match.

— The value of any ordinary row pattern column reference is null.

— The value of any aggregate or row pattern navigation operation is computed using an empty set of rows (so any COUNT is 0, and all other aggregates and row pattern navigation operations are null).

— The value of any column corresponding to a column of the row pattern input table is the same as the corresponding column in the starting row of the empty match.

The following example alters the preceding example slightly, to use ALL ROWS PER MATCH SHOW EMPTY MATCHES:

```
SELECT M.Symbol,   /* ticker symbol */
       M.Matchno,   /* sequential match number */
       M.Tradeday,  /* day of trading */
       M.Price,     /* price on day of trading */
       M.Classy,    /* classifier */
       M.Firstp,    /* starting price */
       M.Lastp      /* ending price */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  CLASSIFIER AS Classy,
                  FINAL FIRST (A.Price) AS Firstp,
                  FINAL LAST (A.Price) AS Lastp
         ALL ROWS PER MATCH SHOW EMPTY MATCHES
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A*)
         DEFINE A AS A.Price > PREV (A.Price)
       ) AS M
```

The result of the preceding query on the sample row pattern partition is:

Table 7 — Results of query using SHOW EMPTY ROWS

SYMBOL	MATCH NO	TRADEDAY	PRICE	CLASSY	FIRSTP	LASTP
XYZ	1	2009-06-08	50			
XYZ	2	2009-06-09	60	A	60	60
XYZ	3	2009-06-10	49			
XYZ	4	2009-06-11	40			
XYZ	5	2009-06-12	35			
XYZ	6	2009-06-15	45	A	45	45
XYZ	7	2009-06-16	45			
XYZ	8	2009-06-17	45			
XYZ	9	2009-06-18	43			

SYMBOL	MATCH NO	TRADEDAY	PRICE	CLASSY	FIRSTP	LASTP
XYZ	10	2009-06-19	47	A	47	70
XYZ	10	2009-06-22	52	A	47	70
XYZ	10	2009-06-23	70	A	47	70
XYZ	11	2009-06-24	60			

If, instead, ALL ROWS PER MATCH OMIT EMPTY MATCHES were used, the result would lack the rows for the empty matches, like this:

Table 8 — Results of query using OMIT EMPTY ROWS

SYMBOL	MATCH NO	TRADEDAY	PRICE	CLASSY	FIRSTP	LASTP
XYZ	2	2009-06-09	60	A	60	60
XYZ	6	2009-06-15	45	A	45	45
XYZ	10	2009-06-19	47	A	47	70
XYZ	10	2009-06-22	52	A	47	70
XYZ	10	2009-06-23	70	A	47	70

Note the gaps in the match numbering; also, the final empty match (number 11) is undetectable because there are no non-empty matches following it.

3.10.2 Handling unmatched rows

Some rows of the row pattern input table may be neither the starting row of an empty match, nor mapped by a non-empty match. Such rows are called unmatched rows.

The option ALL ROWS PER MATCH WITH UNMATCHED ROWS shows both empty matches and unmatched rows. Empty matches are handled the same as with SHOW EMPTY MATCHES. When displaying an unmatched row, all row pattern measures are null, somewhat analogous to the null-extended side of an outer join. Thus COUNT and MATCH_NUMBER may be used to distinguish an unmatched row from the starting row of an empty match. The exclusion syntax `{- -}` is prohibited as contrary to the spirit of WITH UNMATCHED ROWS.

The example in Subclause 3.2, “Example of ALL ROWS PER MATCH”, can be used to illustrate WITH UNMATCHED ROWS. The change in the query syntax is:

```
SELECT M.Symbol,   /* ticker symbol */
       M.Matchno,  /* sequential match number */
```



```
       M.Tradeday, /* day of trading */
       M.Price,    /* price on day of trading */
       M.Classy,   /* classifier */
       M.Startp,   /* starting price */
       M.Bottomp,  /* bottom price */
       M.Endp,     /* ending price */
       M.Avgp      /* average price */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES MATCH_NUMBER() AS Matchno,
                  CLASSIFIER AS Classy,
                  A.Price AS Startp,
                  FINAL LAST (B.Price) AS Bottomp,
                  FINAL LAST (C.Price) AS Endp,
                  FINAL AVG (U.Price) AS Avgp
         ALL ROWS PER MATCH WITH UNMATCHED ROWS
         AFTER MATCH SKIP PAST LAST ROW
         PATTERN (A B+ C+)
         SUBSET U = (A, B, C)
         DEFINE /* A defaults to True, matches any row */
                B AS B.Price < PREV (B.Price),
                C AS C.Price > PREV (C.Price)
       ) AS M
```

and the result on the data in the sample row pattern partition is:

Table 9 — Results of ALL ROWS PER MATCH

SYMBOL	MATCHNO	TRADEDAY	PRICE	CLASSY	STARTP	BOTTOMP	ENDP	AVGP
XYZ		2009-06-08	50					
XYZ	1	2009-06-09	60	A	60	35	45	45.8
XYZ	1	2009-06-10	49	B	60	35	45	45.8
XYZ	1	2009-06-11	40	B	60	35	45	45.8
XYZ	1	2009-06-12	35	B	60	35	45	45.8
XYZ	1	2009-06-15	45	C	60	35	45	45.8
XYZ		2009-06-16	45					
XYZ	2	2009-06-17	45	A	45	43	70	51.4
XYZ	2	2009-06-18	43	B	45	43	70	51.4
XYZ	2	2009-06-19	47	C	45	43	70	51.4
XYZ	2	2009-06-22	52	C	45	43	70	51.4

SYMBOL	MATCHNO	TRADEDAY	PRICE	CLASSY	STARTP	BOTTOMP	ENDP	AVGP
XYZ	2	2009-06-23	70	C	45	43	70	51.4
XYZ		2009-06-24	60					

In the sample output, note the rows in which the row pattern measures are null. These rows correspond to unmatched rows in the row pattern input table.

It is not possible for a pattern to permit empty matches and also have unmatched rows. The reason is that if a row of the row pattern input table cannot be mapped to a primary row pattern variable, then that row can still be the starting row of an empty match, and will not be regarded as unmatched, assuming that the pattern permits empty matches. Thus, if a pattern permits empty matches, then the output using ALL ROWS PER MATCH SHOW EMPTY MATCHES is the same as the output using ALL ROWS PER MATCH WITH UNMATCHED ROWS. Thus WITH UNMATCHED ROWS is primarily intended for use with patterns that do not permit empty matches. However, the user may prefer to specify WITH UNMATCHED ROWS if the user is uncertain whether a pattern may have empty matches or unmatched rows.

Note that if ALL ROWS PER MATCH WITH UNMATCHED ROWS is used with the default skipping behavior (AFTER MATCH SKIP PAST LAST ROW), then every row of the row pattern input table will appear exactly once in the output (as the location of an empty match, as a row that is mapped by a non-empty match, or as an unmatched row).

Other skipping behaviors are permitted using WITH UNMATCHED ROWS, in which case it becomes possible for a row to be mapped by more than one match and appear in the row pattern output table multiple times. Unmatched rows will appear in the output only once.

3.11 AFTER MATCH SKIP

The AFTER MATCH SKIP clause determines the point to resume pattern matching after a non-empty match has been found. The default for the clause is AFTER MATCH SKIP PAST LAST ROW. The options are as follows (*RPV* denotes a row pattern variable):

— AFTER MATCH SKIP TO NEXT ROW: resume pattern matching at the row after the first row of the current match.

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

— AFTER MATCH SKIP TO FIRST *RPV*: resume pattern matching at the first row that is mapped to the row pattern variable *RPV*.

— AFTER MATCH SKIP TO LAST *RPV*: resume pattern matching at the last row that is mapped to the row pattern variable *RPV*.

— AFTER MATCH SKIP TO *RPV*: same as AFTER MATCH SKIP TO LAST *RPV*.

When using AFTER MATCH SKIP TO FIRST or AFTER MATCH SKIP TO [LAST], it is possible that no row is mapped to the <row pattern variable name>. For example, the row pattern variable A in


```
AFTER MATCH SKIP TO A
PATTERN (X A* X),
```

might have no rows mapped to A. If there is no row mapped to A, then there is no row to skip to, so a run-time exception is generated.

Another aberrant condition is that AFTER MATCH SKIP may try to resume pattern matching at the same row that the last match started. For example,

```
AFTER MATCH SKIP TO X
PATTERN (X Y+ Z),
```

In this example, AFTER MATCH SKIP TO X tries to resume pattern matching at the same row where the previous match was found. This would result in an infinite loop; consequently a run-time exception is generated for this scenario.

Note that the AFTER MATCH SKIP syntax only determines the point to resume scanning for a match after a non-empty match. When an empty match is found, one row is skipped (as if SKIP TO NEXT ROW had been specified). Thus an empty match never causes one of these exceptions.

A query that gets one of these exceptions should be rewritten. For example,

```
AFTER MATCH SKIP TO A
PATTERN (X (A | B) Y)
```

will cause a run-time error if alternative A does not match. Instead of this example, perhaps the following will serve the uesr's needs:

```
AFTER MATCH SKIP TO C
PATTERN (X (A | B) Y)
SUBSET C = (A, B)
```

In the revised example, no run-time error is possible, whether A or B is matched.

As another example:

```
AFTER MATCH SKIP TO FIRST A
PATTERN (A* X)
```

This example will always get an exception after the first match, either for skipping to the first row of the match (if A* matches) or for skipping to a non-existent row (if A* does not match). In this example, SKIP TO NEXT ROW might be a better choice.

When using ALL ROWS PER MATCH together with skip options other than AFTER MATCH SKIP PAST LAST ROW, it is possible for consecutive matches to overlap, in which case a row *R* of the row pattern input table might occur in more than one match. In that case, the row pattern output table will have one row for each match in which *R* participates. The MATCH_NUMBER function may be used to distinguish between the multiple matches in which a row of the row pattern input table participates. When a row participates in more than one match, its classifier may be different in each match as well.

3.12 PATTERN

The PATTERN clause is used to specify a regular expression. The regular expression is enclosed in parentheses. It is built from primary row pattern variables, and may use the following operators:

— concatenation: indicated by the absence of any operator sign between two successive items in a pattern. Note that whitespace is required to delimit two successive primary row pattern variables.

— quantifiers: quantifiers are postfix operators with the following choices:

  * — 0 or more iterations

  + — 1 or more iterations

  ? — 0 or 1 iterations

  { *n* } — exactly *n* iterations ($n > 0$)

  { *n*, } — *n* or more iterations ($n \geq 0$)

  { *n*, *m* } — between *n* and *m* (inclusive) iterations ($0 \leq n \leq m$, $0 < m$)

  { , *m* } — between 0 and *m* (inclusive) iterations ($m > 0$)

  reluctant quantifiers, indicated by an additional question mark (*?, +?, ??, {*n*}?, {*n*,}?, { *n*, *m* }?, {,*m*}?). See below for the difference between reluctant and non-reluctant quantifiers.

— alternation: indicated by a vertical bar (|). Alternatives are preferred in the order in which they are specified.

— grouping: indicated by parentheses.

— PERMUTE: see Subclause 3.12.1, “PERMUTE”.

— exclusion: parts of the pattern to be excluded from the output of ALL ROWS PER MATCH are enclosed between `{-` and `-}`. See Subclause 3.12.2, “Excluding portions of the pattern”.

— anchors (not permitted with row pattern matching in windows):

  ^: matches the begining of a row pattern partition

  $: matches the end of a row pattern partition

— ( ): empty pattern, matches an empty set of rows

The difference between non-reluctant (or “greedy”) and reluctant quantifiers appended to a single row pattern variable is illustrated as follows: A* tries to map as many rows as possible to A (consistent with mapping the entire pattern), whereas A*? tries to map as few rows as possible to A (consistent with mapping the entire pattern). The semantics of quantifiers on complex regular expressions, such as (A | B)*, cannot be expressed succinctly; see Subclause 6.2.3, “Quantification”,

The precedence of the operators in a regular expression, in decreasing order, is as follows:

— primaries: primary row pattern variables, anchors, PERMUTE, parenthetic expressions, exclusion syntax, empty pattern

— quantifier; a primary may have zero or one quantifier

— concatenation

— alternation

Precedence of alternation is illustrated by this example:

```
PATTERN (A B | C D)
```

which is equivalent to

```
PATTERN ((A B) | (C D))
```

and is not equivalent to

```
PATTERN (A (B | C) D)
```

Precedence of quantifiers is illustrated by this example:

```
PATTERN (A B *)
```

which is equivalent to

```
PATTERN (A (B*))
```

and is not equivalent to

```
PATTERN ((A B)*)
```

A quantifier may not immediately follow another quantifier. For example

```
PATTERN (A**)
```

is prohibited, whereas

```
PATTERN ((A*)*)
```

is permitted (though the latter pattern is no more powerful than just A*).

It is permitted for a primary row pattern variable to occur more than once in a pattern. For example

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

3.12.1 PERMUTE

The PERMUTE syntax may be used to express a pattern that is a permutation of simpler patterns. For example,

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

is equivalent to an alternation of all permutations of three row pattern variables A, B and C, like this:

```
PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)
```

Note that PERMUTE is expanded lexicographically. (In this example, since the three row pattern variables A, B, and C are listed in alphabetic order, it follows from lexicographic expansion that the expanded possibilities are also listed in alphabetic order.) This is significant because alternatives are attempted in the order written in

the expansion. Thus a match to (A B C) will be attempted before a match to (A C B), *etc*.; the first attempt that succeeds is the “winner”.

As another example:

```
PATTERN (PERMUTE ( X{3}, B C?, D))
```

is equivalent to

```
PATTERN (   ( X{3} B C? D )
          | ( X{3} D B C? )
          | ( B C? X{3} D )
          | ( B C? D X{3} )
          | ( D X{3} B C? )
          | ( D B C? X{3} ) )
```

3.12.2 Excluding portions of the pattern

When using ALL ROWS PER MATCH with either the OMIT EMPTY MATCHES or SHOW EMPTY MATCHES suboptions, rows matching a portion of the PATTERN may be excluded from the row pattern output table. The excluded portion is bracketed between `{-` and `-}` in the PATTERN clause.

For example, the following example finds the longest periods of increasing prices that start with a price no less than 10.

```
SELECT M.Symbol,    /* row's symbol */
       M.Tradeday,  /* row's trade day */
       M.Price,     /* row's price */
       M.Avgp,      /* average price */
       M.Matchno    /* row's match number */
FROM Ticker
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES FINAL AVG (S.Price) AS Avgp,
         MATCH_NUMBER() AS Matchno
         ALL ROWS PER MATCH
         AFTER MATCH SKIP TO LAST B
         PATTERN ( {- A -} B+ {- C -} )
         SUBSET S = (A, B)
         DEFINE A AS A.Price >= 10
                B AS B.Price > PREV (B.Price),
                C AS C.Price <= PREV (C.Price)
       ) AS M;
```

The row pattern output table will only have rows that are mapped to B; the rows mapped to A and C will be excluded from the output.

Although the excluded rows do not appear in the row pattern output table, they are not excluded from the definitions of union row pattern variables, nor from the calculation of scalar expressions in the DEFINE or MEASURES. For example, see the definitions of the primary row pattern variables A and C, the definition of union row pattern variable S, or the Avgp row pattern measure in the example above.

Also, excluded rows do not alter the behavior of AFTER MATCH SKIP. That is, excluded rows are still used in deciding where to resume looking for the next match. For example, in the example above, suppose the AFTER MATCH SKIP clause were changed to

```
AFTER MATCH SKIP PAST LAST ROW
```

while leaving the pattern the same:

```
PATTERN ( {- A -} B+ {- C -} )
```

In that case, a match to the pattern must map a row to the row pattern variable C, and the skip will be to the next row after the last row of the match; that is, after the row that is mapped to C, even though the row that is mapped to C is excluded from the output.

The exclusion syntax is not permitted with ALL ROWS PER MATCH WITH UNMATCHED ROWS.

The exclusion syntax is permitted with ONE ROW PER MATCH, though it has no effect since in this case there is only a single summary row per match.

3.13 SUBSET

The SUBSET clause is optional. It is used to declare union row pattern variables. For example:

```
FROM Ticker
  MATCH_RECOGNIZE
    ( ORDER BY Tradeday
      MEASURES FIRST (X.time) AS x_firsttime,
      LAST (Y.time) AS y_lasttime,
      AVG (S.Price) AS xy_avgprice
      PATTERN (X+ Y+)
      SUBSET S = (X, Y)
      DEFINE X AS X.Price > PREV (X.Price),
             Y AS Y.Price < PREV (Y.Price)
    )
```

This example declares a union row pattern variable, S, and defines it as the union of the rows mapped to X and the rows mapped to Y. See Subclause 3.8, “Row pattern variables”, for an example of how such unions are formed.

There can be multiple union row pattern variables. For example:

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

The right hand side of a SUBSET item is a parenthesized, comma-separated list of distinct primary row pattern variables. This defines the union row pattern variable (on the left hand side) as the union of the primary row pattern variables (on the right hand side).

Note that the list of row pattern variables on the right hand side cannot include any union row pattern variables (there are no unions of unions).

3.14 DEFINE

DEFINE is a mandatory clause, used to specify the Boolean condition that defines a primary row pattern variable. 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). (PREV is a row pattern navigation operation which evaluates an expression in the previous row; see Subclause 4.5, “Row pattern navigation operations”, regarding the complete set of row pattern navigation operations.)

A primary row pattern variable does not require a definition; if there is no definition, the default is a predicate that is always true. Any row can be mapped to such a primary row pattern variable.

A union row pattern variable cannot be defined by DEFINE, but may appear in the Boolean condition of a primary row pattern variable.

The Boolean condition of a primary row pattern variable *RPV* may reference *RPV*, or other primary or union row pattern variables. For example:

```
FROM Ticker
  MATCH_RECOGNIZE
    ( PARTITION BY Symbol
      ORDER BY Tradeday
      MEASURES FIRST (A.Tradeday) AS A_Firstday,
      LAST (D.Tradeday) AS D_Lastday,
      AVG (B.Price) AS B_Avgprice,
      AVG (D.Price) AS D_Avgprice
      PATTERN ( A B+ C+ D )
      SUBSET BC = (B, C)
      DEFINE A AS Price > 100,
             B AS B.Price > A.Price,
             C AS C.Price < AVG (B.Price),
             D AS D.Price > MAX (BC.Price)
    ) AS M
```

In this example:

— The definition of A implicitly references the universal row pattern variable (because of the unqualified column reference Price).

— The definition of B references the primary row pattern variable A.

— The definition of C references the primary row pattern variable B.

— The definition of D references the union row pattern variable BC.

The Boolean conditions are evaluated on successive rows of a row pattern partition in a trial match, with the current row being tentatively mapped to a primary row pattern variable *PRPV* as permitted by the pattern. To be successfully mapped to *PRPV*, the Boolean condition that defines *PRPV* must evaluate to *True*.

In the preceding example:


```
A AS Price > 100
```

Here Price is an unqualified column reference, so it is implicitly qualified by the universal row pattern variable. All rows that are already mapped, including the current row, are mapped to the universal row pattern variable. Also, Price is an ordinary row pattern column reference, so it is evaluated in the last row mapped to the universal row pattern variable, *i.e.*, the current row. Thus this condition is equivalent to A AS A.Price > 100.

```
B AS B.Price > A.Price
```

Here B.Price and A.Price are ordinary row pattern column references. B.Price refers to the current row (since B is being defined), whereas A.Price refers to the last row mapped to A. In view of the pattern, the only row mapped to A is the first row to be mapped.

```
C AS C.Price < AVG (B.Price)
```

Here C.Price refers to the Price in the current row, since C is being defined. The aggregate AVG (B.Price) is computed as the average of all rows that are *already* mapped to B (but not to any rows that might be mapped to B later).

```
D AS D.Price > MAX (BC.Price)
```

This example is similar to the preceding, though it illustrates the use of a union row pattern variable in the Boolean condition.

The semantics of Boolean conditions are discussed in more detail in Clause 4, “Expressions in MEASURES and DEFINE”.

3.15 The row pattern output table

The result of MATCH_RECOGNIZE is called the row pattern output table. The shape (row type) of the row pattern output table depends on the choice of ONE ROW PER MATCH or ALL ROWS PER MATCH:

— If ONE ROW PER MATCH is specified or implied, then the columns of the row pattern output table are the row pattern partitioning columns in their order of declaration, followed by the row pattern measure columns in their order of declaration. Since a table must have at least one column, this implies that there must be at least one row pattern partitioning column or one row pattern measure column.

— If ALL ROWS PER MATCH is specified, then the columns of the row pattern output table are the row pattern partitioning columns in their order of declaration, the ordering columns in their order of declaration, the row pattern measure columns in their order of declaration, and finally any remaining columns of the row pattern input table, in the order they occur in the row pattern input table.

The order of columns in the row pattern output table is only significant when using SELECT *. The order of columns is designed to facilitate comparing the output when the query is toggled between ONE ROW PER MATCH and ALL ROWS PER MATCH.

The names and declared types of the row pattern measure columns are determined by the MEASURES clause. The names and declared types of the non-measure columns are inherited from the corresponding columns of the row pattern input table.

3.15.1 Row pattern output name

Optionally, a correlation name may be assigned to the row pattern output table, like this:

```
SELECT T.Matchno
FROM Ticker
       MATCH_RECOGNIZE ( ...
         MEASURES MATCH_NUMBER () AS Matchno
         ...
       ) AS T
```

In the preceding example, M is the correlation name assigned to the row pattern output table. The noise word AS is optional.

The benefit to assigning a correlation name is that the correlation name may be used to qualify the column names of the row pattern output table, as in M.Matchno in the preceding example. This is especially important to resolve ambiguous column names if there are other tables in the FROM clause. For example, suppose Matchmaker is a table with a column named Matchno, to be joined with the row pattern recognition shown above. In that case the query might be written:

```
SELECT T. Matchno, M.Matchno
FROM Ticker
       MATCH_RECOGNIZE ( ...
         MEASURES MATCH_NUMBER () AS Matchno
         ...
       ) AS T, Matchmaker AS M
WHERE ...
```

3.15.2 Row pattern output declared column list

Optionally, the row pattern output name may be followed by a parenthesized list of column names, called the row pattern output declared column list. The row pattern output declared column list may be used to rename the columns of the row pattern output table. There must be the same number of column names in the list as there are columns in the row pattern output table. The column names in the list are in one-to-one correspondence with the columns of the row pattern output table.

For example, the following is a modification of the example in Subclause 3.4.2, “The row pattern input declared column list”:

```
SELECT M.Cym,          /* ticker symbol */
       M.Mno,          /* sequential match number */
       M.Startprice,   /* starting price */
       M.Bottomprice,  /* bottom price */
       M.Endprice,     /* ending price */
       M.Avgprice      /* average price */
FROM Ticker AS T (Sym, Td, Pr)
       MATCH_RECOGNIZE (
         PARTITION BY T.Sym
         ORDER BY T.Td
         MEASURES MATCH_NUMBER() AS Matchno,
         A.Pr AS Startp,
         LAST (B.Pr) AS Bottomp,
         LAST (C.Pr) AS Endp,
```



```
  AVG (U.Pr) AS Avgp
  ONE ROW PER MATCH
  AFTER MATCH SKIP PAST LAST ROW
  PATTERN (A B+ C+)
  SUBSET U = (A, B, C)
  DEFINE /* A defaults to True, matches any row */
         B AS B.Pr < PREV (B.Pr),
         C AS C.Pr > PREV (C.Pr)
) AS M (Cym, Mno, Startprice, Bottomprice, Endprice, Avgprice)
```

The preceding example uses ONE ROW PER MATCH, so the columns of the row pattern output table are the row pattern partitioning column Sym, followed by the row pattern measure columns Matchno, Startp, Bottomp, Endp, and Avgp, for a total of six columns. The row pattern output declared column list renames these columns to Cym, Mno, Startprice, Bottomprice, Endprice, and Avgprice, respectively. Note that the SELECT list must use the column names of the row pattern output declared column list, since those are the final names of the columns.

In all, the preceding example has the originally defined column names and their renames as shown in the following table:

Table 10 — Original and renamed column names

row pattern input table		**row pattern output table**	
original column name	**renamed column name**	**original column name**	**renamed column name**
Symbol	Sym	Sym	Cym
Tradeday	Td		
Price	Pr		
		Matchno	Mno
		Startp	Startprice
		Bottomp	Bottomprice
		Endp	Endprice
		Avgp	Avgprice

3.16 Prohibited nesting

The following kinds of nesting are prohibited by [ISO9075-2]:

1) Nesting one row pattern recognition within another is prohibited.

2) Outer references in MEASURES or DEFINE are prohibited. This means that a row pattern recognition cannot reference any table in an outer query block except the row pattern input table. (The row pattern

input table is referenced using row pattern variables, not range variables defined outside the MATCH_RECOGNIZE.)

3) Subqueries in MEASURES or DEFINE cannot reference row pattern variables.

4) Row pattern recognition cannot be used in recursive queries.

These restrictions are illustrated in the following Subclauses.

3.16.1 Row pattern recognition nested within another row pattern recognition

Nesting one row pattern recognition within another is prohibited. For example, the following is a syntax error:

```
SELECT ...
FROM Ticker
       MATCH_RECOGNIZE (
         ...
         DEFINE A AS EXISTS ( SELECT *
                              FROM Stock2
                                MATCH_RECOGNIZE (...) )
     )
```

A possible workaround is to relegate the nested row pattern recognition to a view or SQL-invoked function.

3.16.2 Outer references within a row pattern recognition query

Here is an example of row pattern recognition nested within an outer query. Note the underlined range variables T:

```
SELECT ( SELECT M.Avg_Price
         FROM Ticker
                MATCH_RECOGNIZE (
                  ORDER BY Tradeday
                  MEASURES AVG (T.Price) AS Avg_Price
                  PATTERN (T+)
                  DEFINE T AS T.Price >= AVG (T.Price)
                  ) AS M
       )
FROM Toast AS T
```

In this example, T is both the range variable for Toast in the outer query, and also a row pattern variable in the scalar subquery.

SQL uses static scoping rules. This means that a range variable declared in an inner scope occludes a range variable of the same name declared in an outer scope. In the preceding example, there are two range variables named T. The row pattern variable T is declared in the PATTERN clause and visible in the DEFINE and MEASURES clauses, occluding the range variable T of the outer query block. Therefore, the scalar subquery (the row pattern recognition query) is not correlated with the outer query, the overall result will have one row for each row of Toast, and all rows will be identical.

This example is permitted because there are no outer references in the MATCH_RECOGNIZE. However, while legal, having multiple range variables with the same name can be confusing, so this example might be better written by changing one of the range variables. For example, changing the row pattern variable from T to X gives the equivalent query:

```
SELECT ( SELECT M.Avg_Price
         FROM Ticker
                MATCH_RECOGNIZE (
                  ORDER BY Tradeday
                  MEASURES AVG (X.Price) AS Avg_Price
                  PATTERN (X+)
                  DEFINE X AS X.Price >= AVG (X.Price)
                ) AS M
       )
FROM Toast AS T
```

On the other hand, the following is a syntax error:

```
SELECT ( SELECT M.Avg_Price
         FROM Ticker
                MATCH_RECOGNIZE (
                  ORDER BY Tradeday
                  MEASURES AVG (X.Price) AS Avg_Price
                  PATTERN (X+)
                  DEFINE X AS T.Price >= AVG (X.Price)
                ) AS M
       )
FROM Toast AS T
```

In the preceding example, the column reference T.Price in the DEFINE clause is an outer reference to the range variable T defined in the outer block; therefore, this example is a syntax error.

It may be possible to work around this limitation by placing the row pattern recognition in an SQL-invoked routine, passing as arguments the values that are prohibited as outer references.

3.16.3 Conventional query nested within row pattern recognition query

A subquery can be nested in an expression in MEASURES or DEFINE. Subqueries are permitted if they do not perform row pattern recognition themselves, and if they do not reference the row pattern variables of the outer query. Here is an example of the latter (note underlined A):

```
SELECT Firstday
FROM Ticker
       MATCH_RECOGNIZE (
         ORDER BY Tradeday
         MEASURES A.Tradeday AS Firstday
         PATTERN (A B+)
         DEFINE A AS A.Price > 100,
                B AS B.Price <
                  ( SELECT AVG (S.Price)
                    FROM Ticker S
                    WHERE S.Tradeday BETWEEN
                      A.Tradeday - INTERVAL '1' YEAR
```

```
                    AND A.Tradeday )
       )
```

In this example, the definition of B involves a subquery that is correlated with the row pattern variable A (note underlining). This is a syntax error, since subqueries of row pattern matching cannot reference row pattern variables.

It may be possible to work around this limitation by placing the correlated subquery in an SQL-invoked routine, passing as arguments the values that are prohibited as outer references.

3.16.4 Recursion

Row pattern matching is prohibited in recursive queries. For example, the following is a syntax error:

```
CREATE RECURSIVE VIEW Problem (Kolo, Xoro) AS
  SELECT Kolo, Xoro
  FROM T
     UNION
       SELECT Kolo + 1, Xoro
       FROM Problem
              MATCH_RECOGNIZE (
                ORDER BY Kolo
                MEASURES MATCH_NUMBER () AS Xoro
                ALL ROWS PER MATCH
                PATTERN (A+)
                DEFINE A AS A.Xoro > PREV (A.Xoro)
              )
```

3.16.5 Concatenated row pattern recognition

Note that it is not prohibited to feed the output of one row pattern recognition into the input of another, as in this example:

```
SELECT ...
FROM ( SELECT *
       FROM Ticker
              MATCH_RECOGNIZE (...) )
      MATCH_RECOGNIZE (...)
```

In this example, the first MATCH_RECOGNIZE is in a derived table, which then provides the input to the second MATCH_RECOGNIZE.

4 Expressions in MEASURES and DEFINE

Scalar expression syntax as defined in [ISO9075-2] is available in row pattern matching. This provides such familiar capabilities as arithmetic and aggregates. Note though that scalar expressions have special semantics in MEASURES and DEFINE; this is the subject of this Subclause.

In addition, [ISO9075-2] provides the following scalar expressions that are unique to row pattern matching:

— The COUNT aggregate has special syntax and semantics to count rows that are mapped to a row pattern variable by the current row pattern match.

— Row pattern navigation operations, using the functions PREV, NEXT, FIRST, and LAST. Row pattern navigation operations are discussed in Subclause 4.5, “Row pattern navigation operations”.

— The MATCH_NUMBER function, which returns the sequential number of a row pattern match within its row pattern partition, discussed in Subclause 4.7, “MATCH_NUMBER function”.

— The CLASSIFIER function, which returns the name of the primary row pattern variable to which a row is mapped, discussed in Subclause 4.8, “CLASSIFIER function”.

Expressions in MEASURES and DEFINE clauses have the same syntax and semantics, with the following exceptions:

1) DEFINE clause only supports running semantics; MEASURES defaults to running semantics, but also supports final semantics. This distinction is discussed in Subclause 4.2, “Running *vs.* final semantics”.

2) In DEFINE, the CLASSIFIER function does not return the classifier of rows after the current row, whereas in MEASURES, the CLASSIFIER function does return the classifier of rows after the current row. (This is only an issue when CLASSIFIER function is nested within the NEXT row pattern navigation operation; see Subclause 4.8, “CLASSIFIER function”.)

4.1 Row pattern column references

A column reference is a column name qualified by an explicit or implicit range variable, such as

```
A.Price
```

A column name with no qualifier, such as Price, is implicitly qualified by the universal row pattern variable, which references the set of all rows in a match.

Column references may in general be nested within other syntactic elements, notably aggregates and subqueries. (However, nesting in row pattern matching is subject to limitations described in Subclause 3.16, “Prohibited nesting”, for the FROM clause and Subclause 5.16, “Prohibited nesting”, for the WINDOW clause.)

A column reference that is qualified by an explicit or implicit row pattern variable is called a row pattern column reference. Row pattern column references are classified as follows:

— Nested within an aggregate, such as SUM: aggregated row pattern column reference.

— Nested within a row pattern navigation operation (PREV, NEXT, FIRST, and LAST): a navigated row pattern column reference.

— Otherwise: an ordinary row pattern column reference.

All row pattern column references in an aggregate or row pattern navigation operation must be qualified by the same row pattern variable. For example

```
PATTERN (A+ B+)
DEFINE B AS AVG (A.Price + B.Tax) > 1000
```

The preceding example is a syntax error, because A and B are two different row pattern variables. Aggregate semantics require a single set of rows; there is no way to form a single set of rows on which to evaluate A.Price + B.Tax. On the other hand, this is acceptable:

```
DEFINE B AS AVG (B.Price + B.Tax) > 1000
```

In the preceding example, all row pattern column references in the aggregate are qualified by B.

An unqualified column reference is implicitly qualified by the universal row pattern variable, which references the set of all rows in a match. For example

```
DEFINE B AS AVG (Price + B.Tax) > 1000
```

The preceding example is a syntax error, because the unqualified column reference Price is implicitly qualified by the universal row pattern variable, whereas B.Tax is explicitly qualified by B. On the other hand, this is acceptable:

```
DEFINE B AS AVG (Price + Tax) > 1000
```

In the preceding example, both Price and Tax are implicitly qualified by the universal row pattern variable.

4.2 Running vs. final semantics

Pattern matching in a sequence of rows is usually envisioned as an incremental process, with one row after another examined to see if it fits the pattern. With this incremental processing model, at any step until the complete pattern has been recognized, there is only a partial match and it is not known what rows might be added in the future, nor what variables those future rows might be mapped to. Therefore, in [ISO9075-2], a row pattern column reference in the Boolean condition of a DEFINE clause has “running” semantics. This means that a row pattern variable represents the set of rows that have already been mapped to the row pattern variable, up to and including the current row, but not any future rows.

After the complete match has been established, it is possible to talk about “final” semantics. Final semantics is the same as running semantics on the last row of a successful match. Final semantics is only available in MEASURES, since in DEFINE there is uncertainty about whether a complete match has been achieved.

The keywords RUNNING and FINAL are used to indicate running or final semantics, respectively; the rules for these keywords are discussed in Subclause 4.3, “RUNNING *vs.* FINAL keywords”.

The fundamental rule for expression evaluation in MEASURES and DEFINE is as follows:

1) When an expression involving a row pattern variable *RPV* is computed on a group of rows, then the set of rows *SR* that is mapped to *RPV* is used. If *SR* is empty, then COUNT is 0 and any other expression involving *RPV* is null.

2) When an expression requires evaluation in a single row, then the latest row of *SR* is used. If *SR* is empty, then the expression is null.

For example:

```
SELECT M.Symbol, M.Tradeday, M.Price, M.RunningAvg, M.FinalAvg
FROM TICKER
       MATCH_RECOGNIZE (
         PARTITION BY Symbol
         ORDER BY Tradeday
         MEASURES RUNNING AVG (A.Price) AS RunningAvg,
         FINAL AVG (A.Price) AS FinalAvg
         ALL ROWS PER MATCH
         PATTERN (A+)
         DEFINE A AS A.Price >= AVG (A.Price)
       ) AS M
```

Consider the following ordered row pattern partition of data:

Table 11 — Ordered row pattern partition of data

Row	SYMBOL	TRADEDAY	PRICE
R_1	XYZ	2009-06-09	10
R_2	XYZ	2009-06-10	16
R_3	XYZ	2009-06-11	13
R_4	XYZ	2009-06-12	9

The following logic can be used to find a match:

1) On the first row of the row pattern partition, tentatively map row $R = R_1$ to row pattern variable A. At this point $SR = \{ R_1 \}$. To confirm whether this mapping is successful, evaluate the predicate

   ```
   A.Price >= AVG (A.Price)
   ```

   On the left hand side, A.Price must be evaluated in a single row, which is the last row of *SR* using running semantics. The last row of *SR* is R_1; therefore A.Price is 10.

   On the right hand side, AVG (A.Price) is an aggregate, which is computed using the rows of *SR*. This average is 10/1 = 10.

   Thus the predicate asks if 10 ≥ 10. The answer is yes, so the mapping is successful. However, the pattern A+ is “greedy”, so the engine must try to match more rows if possible.

2) On the second row of the row pattern partition, tentatively map $R = R_2$ to row pattern variable A. At this point there are two rows mapped to A, so $SR = \{ R_1, R_2 \}$. Confirm whether the mapping is successful by evaluating the predicate

```
A.Price >= AVG (A.Price)
```

On the left hand side, A.Price must be evaluated in a single row, which is the last row of *SR* using running semantics. The last row of *SR* is R_2; therefore A.Price is 16.

On the right hand side, AVG (A.Price) is an aggregate, which is computed using the rows of *SR*. This average is (10+16)/2 = 13.

Thus the predicate asks if 16 ≥ 13. The answer is yes, so the mapping is successful.

3) On the third row of the row pattern partition, tentatively map $R = R_3$ to row pattern variable A. Now there are three rows mapped to A, so $SR = \{ R_1, R_2, R_3 \}$. Confirm whether the mapping is successful by evaluating the predicate

```
A.Price >= AVG (A.Price)
```

On the left hand side, A.Price is evaluated in R_3; therefore A.Price is 13.

On the right hand side, AVG (A.Price) is an aggregate, which is computed using the rows of *SR*. This average is (10+16+13)/3 = 13.

Thus the predicate asks if 13 ≥ 13. The answer is yes, so the mapping is successful.

4) On the fourth row of the row pattern partition, tentatively map $R = R_4$ to row pattern variable A. At this point $SR = \{ R_1, R_2, R_3, R_4 \}$. Confirm whether the mapping is successful by evaluating the predicate

```
A.Price >= AVG (A.Price)
```

On the left hand side, A.Price is evaluated in R_4; therefore A.Price is 9.

On the right hand side, AVG (A.Price) is an aggregate, which is computed using the rows of *SR*. This average is (10+16+13+9)/4 = 12.

Thus the predicate asks if 9 ≥ 12. The answer is no, so the mapping is not successful.

R_4 did not satisfy the definition of A, so the longest match to A+ is $\{ R_1, R_2, R_3 \}$. Since A+ has a greedy quantifier, this is the preferred match.

The averages computed in the DEFINE are always running averages. In MEASURES, especially with ALL ROWS PER MATCH, it is possible to distinguish final and running aggregates. Notice the use of the keywords RUNNING and FINAL in the MEASURES clause. The distinction can be observed in the result of the example:

Table 12 — RUNNING and FINAL in MEASURES

SYMBOL	TRADEDAY	PRICE	RUNNINGAVG	FINALAVG
XYZ	2009-06-09	10	10	13
XYZ	2009-06-10	16	13	13

SYMBOL	TRADEDAY	PRICE	RUNNINGAVG	FINALAVG
XYZ	2009-06-11	13	13	13

It is possible that the set of rows *SR* mapped to a row pattern variable *RPV* is empty. When evaluating over an empty set:

1) COUNT is 0.

2) Any other aggregate, row pattern navigation operation, or ordinary row pattern column reference is null.

For example:

```
PATTERN ( A? B+ )
DEFINE A AS A.Price > 100,
       B AS B.Price > COUNT (A.*) * 50
```

With the preceding example, consider the following ordered row pattern partition of data :

Table 13 — Ordered row pattern partition of data

Row	PRICE
R_1	60
R_2	70
R_3	40

A match can be found in this data as follows:

1) Tentatively map $R = R_1$ to row pattern variable A. (The quantifier ? means to try first for a single match to A; if that fails, then an empty match is taken as matching A?). To see if the mapping is successful, the predicate

   ```
   A.Price > 100
   ```

   is evaluated. A.Price is 60; therefore the predicate is false and the mapping to A does not succeed.

2) Since the mapping to A failed, the empty match is taken as matching A?.

3) Tentatively map $R = R_1$ to B. The predicate to check for this mapping is

   ```
   B.Price > COUNT (A.*) * 50
   ```

   No rows are mapped to A, therefore COUNT (A.*) is 0. Since B.Price = 60 is greater than 0, the mapping is successful.

4) Similarly, rows R_2 and R_3 can be successfully mapped to B. Since there are no more rows, this is the complete match: no rows mapped A, { R_1, R_2, R_3 } mapped to B.

A row pattern variable can make a forward reference; that is, a reference to a row pattern variable that has not been matched yet. For example,

```
PATTERN (X+ Y+)
DEFINE X AS COUNT (Y.*) > 3,
       Y AS Y.Price > 10
```

is legal syntax. However, this example will never be matched since, at the time that a row is mapped to X, no row has yet been mapped to Y. Thus, COUNT (Y.*) is 0 and can never be greater than 3. This is true even if there are four future rows that might be successfully mapped to Y. Consider this data set:

Table 14 — Ordered row pattern partition of data

Row	PRICE
R_1	2
R_2	11
R_3	12
R_4	13
R_5	14

Mapping { R_2, R_3, R_4, R_5 } to Y would be successful, since all four of these rows satisfy the Boolean condition defined for Y. In that case, one might think that one could map R_1 to X and have a complete successful match. However, the rules of [ISO9075-2] will not find this match, because, according to the pattern X+ Y+, at least one row must be mapped to X before any rows are mapped to Y.

4.3 RUNNING vs. FINAL keywords

RUNNING and FINAL are keywords used to indicate whether running or final semantics are desired. RUNNING and FINAL are available for use with aggregates and the row pattern navigation operations FIRST and LAST.

Aggregates, FIRST, and LAST can occur in the following places in a row pattern matching query:

1) In the DEFINE clause. When processing the DEFINE clause, the engine is still in the midst of recognizing a match; therefore, the only supported semantics is running.

2) In the MEASURES clause. When processing the MEASURES clause, the engine has finished recognizing a match; therefore, it becomes possible to consider final semantics. There are two subcases:

   a) If ONE ROW PER MATCH is specified, or if row pattern matching is done in a window, then the engine is conceptually positioned on the last row of the match, and there is no real difference between running *vs.* final semantics.

   b) If ALL ROWS PER MATCH is specified, then the row pattern output table will have one row for each row of the match. In this circumstance, the user may wish to see both running and final values, so [ISO9075-2] provides the RUNNING and FINAL keywords to support that distinction.

Based on this analysis, [ISO9075-2] specifies the following:

1) In MEASURES, the keywords RUNNING and FINAL may be used to indicate the desired semantics for an aggregate, FIRST, or LAST. The keyword is written before the operator, for example, `RUNNING COUNT (A.*)` or `FINAL SUM (B.Price)`.

2) In both MEASURES and DEFINE, the default is RUNNING.

3) In DEFINE, FINAL is not permitted; RUNNING may be used for added clarity if desired.

4) In MEASURES with ONE ROW PER MATCH or in windows, all aggregates, FIRST, and LAST are computed after the last row of the match has been recognized, so that the default RUNNING semantics is actually no different from FINAL semantics. The user may prefer to think of expressions defaulting to FINAL in these cases. Alternatively, the user may choose to write FINAL for added clarity.

5) Ordinary column references always have running semantics. (To get final semantics in MEASURES, use the FINAL LAST row pattern navigation operation instead of an ordinary column reference.)

4.4 Aggregates

Aggregates (COUNT, SUM, AVG, *etc.*) may be used in both the MEASURES and DEFINE clauses. When used in row pattern matching, aggregates operate on a set of rows that are mapped to a particular row pattern variable, using either running or final semantics. For example:

```
MEASURES SUM (A.Price)      AS RunningSumOverA,
         FINAL SUM(A.Price) AS FinalSumOverA
ALL ROWS PER MATCH
```

In this example, A is a row pattern variable. The first row pattern measure, RunningSumOverA, does not specify either RUNNING or FINAL, so it defaults to RUNNING. This means that it is computed as the sum of Price in those rows that are mapped to A by the current match, up to and including the current row. The second row pattern measure, FinalSumOverA, computes the sum of Price over all rows that are mapped to A by the current match, including rows that may be later than the current row. Final aggregates are only available in MEASURES, not in DEFINE.

An unqualified column reference contained in an aggregate is implicitly qualified by the universal row pattern variable, which references all rows of the current row pattern match. For example:

```
SUM (Price)
```

computes the running sum of Price over all rows of the current row pattern match.

All column references contained in an aggregate must be qualified by the same row pattern variable. For example:

```
SUM (Price + A.Tax)
```

is a syntax error, because Price is implicitly qualified by the universal row pattern variable, whereas A.Tax is explicitly qualified by A.

The COUNT aggregate has special syntax for row pattern matching, so that COUNT(A.*) may be specified. COUNT(A.*) is the number of rows that are mapped to the row pattern variable A by the current row pattern match, using running or final semantics as appropriate. As for COUNT(*), the * is implicitly qualified by the universal row pattern variable, so that COUNT(*) is the number of rows in the current row pattern match, with running or final semantics as appropriate.

4.5 Row pattern navigation operations

There are four functions — PREV, NEXT, FIRST, and LAST — which enable navigation within the row pattern by either physical or logical offsets.

4.5.1 PREV and NEXT

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

```
DEFINE A AS PREV (A.Price) > 100
```

The preceding example says that a row R_n can be mapped to A if the preceding row R_{n-1} has a price greater than 100. If the preceding row does not exist (*i.e.*, R_n is the first row of a row pattern partition), then PREV(A.Price) is null, so the Boolean condition is not True, and therefore the first row cannot be mapped to A.

The PREV function can accept an optional non-negative integer argument indicating the physical offset to the previous rows. Thus:

— PREV (A.Price, 0) is equivalent to A.Price

— PREV (A.price, 1) is equivalent to PREV (A.Price). (Note: 1 is the default offset.)

— PREV (A.Price, 2) is the value of Price in the row two rows prior to the row denoted by A with running semantics. (If no row is mapped to A, or if there is no row two rows prior, then PREV (A.Price, 2) is null.)

The offset must be a run-time constant (literal, embedded variable, and the like, but not a column or a subquery). There is an exception if the value of the offset is negative or null.

The NEXT function may be used to reference rows in the forward direction in the row pattern partition using a physical offset. The syntax is the same as for PREV, except for the name of the function. For example,

```
DEFINE A AS NEXT (A.Price) > 100
```

The preceding example looks forward one row in the row pattern partition. Note that [ISO9075-2] does not support aggregates that look past the current row during DEFINE, because of the difficulty of predicting what row will be mapped to what row pattern variable in the future. The NEXT function does not violate this principle, since it navigates to “future” rows on the basis of a physical offset, which does not require knowing the future mapping of rows.

For example, to find an isolated row that is more than twice the average of the two rows before and two rows after it: using NEXT, this can be expressed:

```
PATTERN ( X )
DEFINE X AS X.Price > 2 * ( PREV (X.Price, 2)
                          + PREV (X.Price, 1)
                          + NEXT (X.Price, 1)
                          + NEXT (X.Price, 2) ) / 4
```

This query can also be expressed:


```
PATTERN ( {- Y Y -} X {- Y Z -} )
SUBSET W = (Y, Z)
DEFINE Z AS X.Price > 2 * AVG (W.Price)
```

The second formulation (without NEXT) requires the use of exclusion syntax using {- -} and the non-intuitive definition of Z in terms of row pattern variables X and W. The row X is never really defined at all, though that is the only row of interest. The first formulation (using NEXT) avoids these issues.

Note that the row in which PREV or NEXT is evaluated is not necessarily mapped to the row pattern variable in the argument. For example, in the first formulation of the example, PREV (X.Price, 2) is evaluated in a row that is not even part of the match. The purpose of the row pattern variable is to identify the row from which to offset, not the row that is ultimately reached. This point is discussed further in Subclause 4.5.3, "Nesting FIRST and LAST within PREV or NEXT".

PREV and NEXT may be used with more than one column reference; for example:

```
DEFINE A AS PREV (A.Price + A.Tax) < 100
```

When using a complex expression as the first argument of PREV or NEXT, all qualifiers must be the same row pattern variable (in this example, A).

The first argument of PREV and NEXT must have at least one column reference or CLASSIFIER function. For example, this is a syntax error:

```
PREV (1)
```

The preceding example is a syntax error because there is no row pattern column reference or CLASSIFIER function. Without a column reference or CLASSIFIER function, there is no way to determine the row that is the starting point for offsetting. (The use of CLASSIFIER function within PREV or NEXT is discussed in Subclause 4.8, "CLASSIFIER function".)

PREV and NEXT always have running semantics; the keywords RUNNING and FINAL cannot be used with PREV or NEXT. To obtain final semantics, use, *e.g.*, PREV (FINAL LAST (A.Price)) as explained in Subclause 4.5.3, "Nesting FIRST and LAST within PREV or NEXT".

4.5.2 FIRST and LAST

FIRST returns the value of an expression evaluated in the first row of the group of rows mapped to a row pattern variable. For example:

```
FIRST (A.Price)
```

The preceding example evaluates A.Price in the first row that is mapped to A. If there is no row mapped to A, then the value is null.

Similarly, LAST returns the value of an expression evaluated in the last row of the group of rows mapped to a row pattern variable. For example:

```
LAST (A.Price)
```

The preceding example evaluates A.Price in the last row that is mapped to A (null if there is no such row).

The FIRST and LAST operators can accept an optional non-negative integer argument indicating a logical offset within the set of rows mapped to the row pattern variable. For example:

```
FIRST (A.Price, 1)
```

evaluates Price in the second row that is mapped to A. Consider the following data set and mappings:

Table 15 — Example data set and mappings for FIRST and LAST

Row	PRICE	mapping
R_1	10	→ A
R_2	20	→ B
R_3	30	→ A
R_4	40	→ C
R_5	50	→ A

Thus:

— FIRST (A.Price) = FIRST (A.Price, 0) = LAST (A.Price, 2) = 10

— FIRST (A.Price, 1) = LAST (A.Price, 1) = 30

— FIRST (A.Price, 2) = LAST (A.Price, 0) = LAST (A.Price) = 50

— FIRST (A.Price, 3) is null, as is LAST (A.Price, 3)

Note that the offset is a logical offset, moving within the set of rows { R_1, R_3, R_5 } that are mapped to the row pattern variable A. It is not a physical offset, as with PREV or NEXT.

The optional integer argument must be a run-time constant (literal, embedded variable, and the like, but not a column or subquery). There is an exception if the value of the offset is negative or null.

The first argument of FIRST or LAST must have at least one row pattern column reference or CLASSIFIER function. (The use of CLASSIFIER function within FIRST or LAST is discussed in Subclause 4.8, “CLASSIFIER function”.) Thus FIRST(1) is a syntax error.

The first argument of FIRST or LAST may have more than one row pattern column reference, in which case all qualifiers must be the same row pattern variable. For example:

```
FIRST (A.Price + B.Tax)
```

is a syntax error, but

```
FIRST (A.Price + A.Tax)
```

is acceptable.

FIRST and LAST support both running and final semantics. RUNNING is the default, and the only supported option in DEFINE. Final semantics may be accessed in the MEASURES by using the keyword FINAL (particularly with ALL ROWS PER MATCH), as in:

```
MEASURES FINAL LAST (A.Price) AS FinalPrice
ALL ROWS PER MATCH
```

4.5.3 Nesting FIRST and LAST within PREV or NEXT

FIRST and LAST provide navigation within the set of rows already mapped to a particular row pattern variable; PREV and NEXT provide navigation using a physical offset from a particular row. These kinds of navigation may be combined by nesting FIRST or LAST within PREV or NEXT. This permits expressions such as the following:

```
PREV (LAST (A.Price + A.Tax, 1), 3)
```

In this example, A must be a row pattern variable. It is required to have a row pattern column reference or CLASSIFIER function, and all row pattern variables in the compound operator must be equivalent (A, in this example). The use of CLASSIFIER function nested within row pattern navigation operations is discussed in Subclause 4.8, “CLASSIFIER function”.

This compound operator is evaluated as follows:

1) The inner operator, LAST, operates on the set of rows that are mapped to the row pattern variable A. In this set, find the row that is “the last minus 1”. (If there is no such row, the result is null.)

2) The outer operator, PREV, starts from the row found in step 1 and backs up 3 rows in the row pattern partition. (If there is no such row, the result is null.)

3) Let *R* be an implementation-dependent range variable that references the row found by step 2. In the expression A.Price + A.Tax, replace every occurrence of the row pattern variable A with *R*. The resulting expression *R*.Price + *R*.Tax is evaluated and determines the value of the compound navigation operation.

For example, consider the data set and mappings:

Table 16 — Data set and mappings for nesting example

Row	PRICE	TAX	mapping
R_1	10	1	
R_2	20	2	→ A
R_3	30	3	→ B
R_4	40	4	→ A
R_5	50	5	→ C
R_6	60	6	→ A

To evaluate

```
PREV (LAST (A.Price + A.Tax, 1), 3)
```

the following steps may be used:

1) The set of rows mapped to A is { R_2, R_4, R_6 }. LAST operates on this set, offsetting from the end to arrive at row R_4.

2) PREV performs a physical offset, 3 rows prior to R_4, arriving at R_1.

3) Let *R* be a range variable pointing at R_1. *R*.Price + *R*.Tax is evaluated, giving 10+1 = 11.

Note that this nesting is not defined as a typical evaluation of nested functions. The inner operator LAST does not actually evaluate the expression A.Price + A.Tax; it merely uses this expression to designate a row pattern variable (A) and then navigate within the rows mapped to that variable. The outer operator PREV performs a further physical navigation on rows. The expression A.Price + A.Tax is not actually evaluated as such, since the row that is eventually reached is not necessarily mapped to the row pattern variable A. In this example, R_1 is not mapped to any row pattern variable.

4.6 Ordinary row pattern column references reconsidered

An ordinary row pattern column reference is one that is neither aggregated nor navigated. For example:

```
A.Price
```

Subclause 4.3, “RUNNING *vs.* FINAL keywords”, stated that ordinary row pattern column references always have running semantics. This means:

1) In DEFINE, an ordinary column reference references the last row that is mapped to the row pattern variable, up to and including the current row. If there is no such row, then the value is null.

2) In MEASURES, there are two subcases:

   a) If ALL ROWS PER MATCH is specified, then there is also a notion of current row, and the semantics are the same as in DEFINE.

   b) If ONE ROW PER MATCH is specified, then conceptually the engine is positioned on the last row of the match. An ordinary column reference references the last row that is mapped to the row pattern variable, anywhere in the pattern. If the variable is not mapped to any row, then the value is null.

These semantics are the same as the LAST operator, with the implicit RUNNING default. Consequently, an ordinary column reference such as:

```
X.Price
```

is equivalent to:


```
RUNNING LAST (X.Price)
```

4.7 MATCH_NUMBER function

Matches within a row pattern partition are numbered sequentially starting with 1 in the order they are found. Note that match numbering starts over again at 1 in each row pattern partition, since there is no inherent ordering between row pattern partitions. MATCH_NUMBER () is a nullary function that returns an exact numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.

All previous examples of MATCH_NUMBER () have shown it used in MEASURES. It is also possible to use MATCH_NUMBER () in DEFINE, where it can be used to define conditions that depend upon the match number. For example:

```
PATTERN ( (A+ | B+) )
DEFINE A AS ( MOD (MATCH_NUMBER (), 2) = 1)
              AND A.Price > PREV (A.Price) ),
       B AS ( MOD (MATCH_NUMBER (), 2) = 0)
              AND B.Price < PREV (B.Price) )
```

The condition for A can only be true on odd-numbered matches, and the condition for B can only be true on even-numbered matches. Thus, the matches will alternate between A+ and B+.

MATCH_NUMBER () is not permitted except in MEASURES and DEFINE. For example, the following is a syntax error:

```
SELECT MATCH_NUMBER ()
FROM Ticker
```

4.8 CLASSIFIER function

The classifier of a row is the primary row pattern variable to which the row is mapped by a row pattern match. The classifier of a row that is not mapped by a row pattern match is null.

The CLASSIFIER function returns a character string whose value is the classifier of a row. The CLASSIFIER function has one optional argument, a row pattern variable, defaulting to the universal row pattern variable.

The simplest usage of CLASSIFIER is with no argument, as seen in the example in Subclause 3.2, “Example of ALL ROWS PER MATCH”:

```
MEASURES ...
         CLASSIFIER () AS Classy, ...
         ALL ROWS PER MATCH
```

In this example, the CLASSIFIER () function returns the classifier of the current row, which is assigned as the value of the row pattern measure Classy. Subclause 3.2, “Example of ALL ROWS PER MATCH”, shows the result of the example query on the sample data, illustrating how the CLASSIFIER function returns the classifier of each row of a row pattern match.

The classifier of the starting row of an empty match is the null value. This can be seen in the example in Subclause 3.10.1, "Handling empty matches", of a query using ALL ROWS PER MATCH SHOW EMPTY MATCHES.

Optionally, the CLASSIFIER function may take a single argument, a row pattern variable *RPV*; the default is the universal row pattern variable. The argument is used to specify a set of rows using running semantics, namely, the set of rows up to and including the current row that are mapped to *RPV*.

The argument will typically be a union row pattern variable; the value returned tells which primary row pattern variable among the components of the union row pattern variable to which a row was mapped. For example:

```
MEASURES CLASSIFIER (AB) AS AorB
. . .
PATTERN (A | B | C)+
SUBSET AB = (A, B)
```

In this example, AB is a union row pattern variable. The value of the row pattern measure AorB is either A or B, whichever is the classifier of the last row that mapped to A or B. If no row mapped to A or B, the value is null.

The CLASSIFIER function may be used in an aggregate. For example:

```
ORDER BY Tradeday
MEASURES ARRAY_AGG (CLASSIFIER () ORDER BY Tradeday)
           AS ClassifierArray
ONE ROW PER MATCH
```

In the preceding example, one row is created for each row pattern match *RPM*, with a single row pattern measure, which is an array of character strings. The elements of the array are the classifiers of the rows in *RPM*, with one array element for each row of the row pattern input table that is mapped by *RPM*. Note that the array is ordered using the same ordering as the row pattern partition. This technique can be used to obtain a value reflecting the precise pattern that was detected, while using ONE ROW PER MATCH instead of ALL ROWS PER MATCH.

The CLASSIFIER function may be nested within a row pattern navigation operation. For example:

```
PREV (CLASSIFIER ())
```

The preceding example returns the classifier of the preceding row. This might be used in DEFINE so that the definition of one row pattern variable might depend on the classifier of a previously matched row. For example:

```
PATTERN ( (A | B) C )
DEFINE A AS ...,
       B AS ...,
       C AS CASE
              WHEN PREV (CLASSIFIER ()) = 'A' AND Price > 100
                THEN 1
              WHEN PREV (CLASSIFIER ()) = 'B' AND Price < 100
                THEN 1
              ELSE 0
            END = 1
```

In this example, the first row might be mapped to either A or B. The definition of C can test PREV (CLASSIFIER ()) to learn to which row pattern variable the first row mapped. If the first row mapped to A, then C will be true if Price > 100; if the first row mapped to B, then C will be true if Price < 100.

This particular example would be more easily written as:

```
PATTERN ( (A AC | B BC) )
DEFINE A AS ...,
       B AS ...,
       AC AS Price > 100,
       BC AS Price < 100
```

However, the example does illustrate a general technique that might be useful for more complex patterns that want to inquire about the mapping chosen in earlier rows.

When nesting CLASSIFIER within NEXT, there is an important distinction between MEASURES and DEFINE. Consider, for example:

```
NEXT (CLASSIFIER ())
```

which asks for the classifier of the next row. This expression, used in DEFINE, will return null, because the next row has not been mapped yet when considering how to map the current row. Used in MEASURES, the entire pattern has been mapped, and the preceding example is able to return the classifier of the row after the current row. (If the current row is the last row that is mapped, the result is null.)

Here is an example of the CLASSIFIER function nested in a compound row pattern navigation operation:

```
NEXT (RUNNING LAST (CLASSIFIER (U), 2) 3)
```

This example would be evaluated as follows:

1) Find the set of rows mapped to U. In DEFINE, this can only be rows up to and including the current row; in MEASURES, this can be any rows mapped to U in the completed match.

2) RUNNING LAST restricts to the set of rows that map to U up to and including the current row. In DEFINE, this is no change from step 1. In MEASURES with ONE ROW PER MATCH, we are positioned on the last row, so this also is no change. In MEASURES with ALL ROWS PER MATCH, this may result in discarding some of the rows mapped to U.

3) In the set of rows remaining after step 2, find the row that is offset 2 from the end. This requires at least three rows remaining after step 2; if there are not that many, then the result is null. (This is the functionality of LAST.)

4) Now move forward in the row pattern partition three rows. If there are not enough rows in the row pattern partition, the result is null. (This is the functionality of NEXT.)

5) Finally, find the primary row pattern variable to which the row is mapped; this is the result. If the row is not mapped, the result is null. (This is the functionality of CLASSIFIER.)

The explicit or implicit argument of CLASSIFIER is a row pattern variable. This row pattern variable is used in the same fashion as the qualifier of a column reference in the argument of a row pattern navigation operation. For example:

```
NEXT (CLASSIFIER () || Name)
```

is a permissible expression, since CLASSIFIER () and Name both reference the universal row pattern variable. On the other hand,

```
NEXT (CLASSIFIER (A) || Name)
```

is not permissible, because the CLASSIFIER function references the row pattern variable A, whereas the column reference Name references the universal row pattern variable. Similarly:

```
NEXT (CLASSIFIER () || A.Name)
```

is not permissible.

The same rule applies to the argument of an aggregate: all row pattern variables referenced explicitly or implicitly must be the same. Thus the following is a syntax error:

```
ARRAY_AGG (CLASSIFIER () || A.Name)
```

The CLASSIFIER function is not permitted except in MEASURES and DEFINE. For example, the following is a syntax error:

```
SELECT CLASSIFIER ()
FROM Ticker
```


5 Row pattern recognition: WINDOW clause

Feature R020, “Row pattern recognition: WINDOW clause” of [ISO9075-2] enhances the WINDOW clause to include row pattern matching. In [ISO9075-2], a window is a data structure defined on the result of a <table expression> (the FROM...WHERE...GROUP BY...HAVING... clauses), producing a derived table. This data structure does three things:

1) Partitions the rows of the derived table according to zero or more columns.

2) Within each window partition, orders the rows of the derived table according to zero or more expressions.

3) For each row *R* in a window partition, defines a window frame, which is a subset of the ordered window partition. The endpoints of the window frame may be the beginning or end of the window partition, or may be defined relative to the current row using either a physical offset (row count), a logical offset (a value added to or subtracted from the only sort column), or a group count (number of groups, defined as peers under the ordering).

Using R020, row pattern recognition may be used to further reduce the window frame. Row pattern recognition is applicable only to window frames that start at the current row *R*. The window frame resulting from step 3 will be called the “full window frame” of *R*, and the window frame after reduction by pattern matching will be called the “reduced window frame” of *R*. When performing row pattern recognition in a window, the window partition serves as the row pattern partition and the window ordering serves as the row pattern ordering.

5.1 Example of row pattern recognition in a window

The example from Subclause 3.1, “Example of ONE ROW PER MATCH”, is adapted to use row pattern matching in the WINDOW clause below:

```
SELECT T.Symbol,        /* ticker symbol */
       T.Tradeday,      /* trade day */
       T.Price,         /* price on day of trading */
       Classy OVER W,   /* classifier */
       Startp OVER W,   /* starting price */
       Bottomp OVER W,  /* bottom price */
       Endp OVER W,     /* ending price */
       Avgp OVER W      /* average price */
FROM Ticker AS T
WINDOW W AS
  ( PARTITION BY Symbol
    ORDER BY Tradeday
    MEASURES A.Price AS Startp,
             LAST (B.Price) AS Bottomp,
             LAST (C.Price) AS Endp,
             AVG (U.Price) AS Avgp
    ROWS
             BETWEEN CURRENT ROW
                 AND UNBOUNDED FOLLOWING
    AFTER MATCH SKIP PAST LAST ROW
```

```
    INITIAL
    PATTERN (A B+ C+)
    SUBSET U = (A, B, C)
    DEFINE /* A defaults to True, matches any row */
           B AS B.Price < PREV (B.Price),
           C AS C.Price > PREV (C.Price)
)
```

In the preceding example, the principle syntactic elements are presented on separate lines. In this example:

— The SELECT list contains window functions over the window W defined in the WINDOW clause. Note the window functions Classy OVER W, Startp OVER W, Bottomp OVER W, Endp OVER W, and Avgp OVER W. These are examples of row pattern measure functions, which are window functions used to access the row pattern measures defined in the MEASURES clause of the window definition.

— FROM introduces a conventional FROM clause. This example has a single table, Ticker, which is assigned the range variable T. Since there is no WHERE, GROUP BY, or HAVING clause, the result of the FROM clause is the row pattern input table in this example.

— WINDOW W declares the window name W.

— PARTITION BY specifies how to partition the row pattern input table. The PARTITION BY clause is a list of columns of the row pattern input table. This clause is optional; if omitted, there are no row pattern partitioning columns, and the entire row pattern input table constitutes a single row pattern partition.

— ORDER BY specifies how to order the rows within row pattern partitions of the row pattern input table. The ORDER BY clause is a list of columns of the row pattern input table. This clause is optional; if omitted, the order of rows is completely non-deterministic. However, since non-deterministic ordering will defeat the purpose of most row pattern recognition, the ORDER BY clause will usually be specified.

— MEASURES specifies row pattern measures, whose values are calculated by evaluating expressions related to the match. The values of row pattern measures are accessed using row pattern measure functions, as illustrated in the SELECT list.

— ROWS specifies the unit to use in defining the full window frame. The other choices, RANGE and GROUPS, are not permitted with row pattern matching in windows.

— BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING is one way to specify the full window frame. In this example, for any row *R* in a row pattern partition *P*, the full window frame consists of *R* and all rows that follow *R* in the row pattern partition *P*. The full window frame is subsequently reduced to just the rows constituting a pattern match.

— AFTER MATCH SKIP clause specifies where to resume looking for the next row pattern match after successfully finding a match. In this example, AFTER MATCH SKIP PAST LAST ROW specifies that pattern matching will resume after the last row of a successful match. When a row is skipped, its reduced window frame is empty.

— INITIAL specifies that the pattern must match starting at the first row of the full window frame. If there is no such pattern match, the reduced window frame is empty. The alternative to INITIAL is SEEK, which specifies to seek the first row pattern match in the full window frame; if there is none, the reduced window frame is empty.

— PATTERN specifies the row pattern that is sought in the row pattern input table. A row pattern is a regular expression using primary row pattern variables. In this example, the row pattern has three primary row pattern variables (A, B, and C).

— SUBSET defines the union row pattern variables. In this example, U is defined as the union of the primary row pattern variables A, B, and C. The SUBSET clause is optional.

— DEFINE specifies the Boolean condition that defines a primary row pattern variable; a row must satisfy the condition in order to be mapped to a particular primary row pattern variable. If a primary row pattern variable is not defined in the DEFINE clause, then its definition defaults to a condition that is always true, meaning that the primary row pattern variable can match any row.

The result of the preceding query on the sample row pattern partition is:

Table 17 — Window Example Query Results

SYMBOL	TRADEDAY	PRICE	CLASSY	STARTP	BOTTOMP	ENDP	AVGP
XYZ	2009-06-08	50					
XYZ	2009-06-09	60	A	60	35	45	45.8
XYZ	2009-06-10	49					
XYZ	2009-06-11	40					
XYZ	2009-06-12	35					
XYZ	2009-06-15	45					
XYZ	2009-06-16	45					
XYZ	2009-06-17	45	A	45	43	70	51.4
XYZ	2009-06-18	43					
XYZ	2009-06-19	47					
XYZ	2009-06-22	52					
XYZ	2009-06-23	70					
XYZ	2009-06-24	60					

5.2 Summary of the syntax

The syntax of row pattern recognition in windows is summarized in the following table:

Table 18 — Row pattern recognition in windows — syntax summary

Clause of window definition	Optional?	Notes	Cross-reference
Existing window name	yes	no default	Subclause 5.5, "Windows defined on windows"
PARTITION BY	yes	if omitted, the row pattern input table constitutes one row pattern partition	Subclause 5.6, "PARTITION BY"
ORDER BY	yes	if omitted, there is a non-deterministic ordering in each row pattern partition	Subclause 5.7, "ORDER BY"
MEASURES	yes	no default	Subclause 5.8, "MEASURES"
ROWS, RANGE, GROUPS	no	only ROWS is permitted with row pattern recognition	Subclause 5.9.1, "ROWS BETWEEN CURRENT ROW AND"
BETWEEN CURRENT ROW AND ...	no	BETWEEN CURRENT ROW AND is required with row pattern recognition	Subclause 5.9.1, "ROWS BETWEEN CURRENT ROW AND"
EXCLUDE NO OTHERS	yes	EXCLUDE NO OTHERS is the default; other EXCLUDE options are forbidden with row pattern matching	Subclause 5.9.2, "EXCLUDE NO OTHERS"
AFTER MATCH SKIP	yes	default is AFTER MATCH SKIP PAST LAST ROW	Subclause 5.10, "AFTER MATCH SKIP"
INITIAL, SEEK	yes	default is INITIAL	Subclause 5.11, "INITIAL *vs.* SEEK"
PATTERN	no	same as MATCH_RECOGNIZE	Subclause 5.12, "PATTERN"
SUBSET	yes	same as MATCH_RECOGNIZE	Subclause 5.13, "SUBSET"
DEFINE	no	same as MATCH_RECOGNIZE	Subclause 5.14, "DEFINE"

5.2.1 Syntactic comparison to windows without row pattern recognition

Note the following differences between windows with and without row pattern matching:

1) Windows with row pattern matching must have the PATTERN and DEFINE clauses, and optionally may also have the MEASURES, AFTER MATCH SKIP, INITIAL, SEEK, and SUBSET clauses. Windows without row pattern matching have none of these clauses.

2) Windows with row pattern matching must start with the current row, and must specify ROWS (nor RANGE or GROUPS).

3) The only permitted EXCLUDE clause with row pattern matching is EXCLUDE NO OTHERS, which is the default.

5.2.2 Syntactic comparison to MATCH_RECOGNIZE

The syntax for row pattern recognition in a window differs from MATCH_RECOGNIZE in the following respects:

1) Row pattern recognition in windows includes the window syntax of conventional windows, with some constraints described in Subclause 5.2.1, "Syntactic comparison to windows without row pattern recognition".

2) Range variables declared in the FROM clause are visible in the PARTITION BY and ORDER BY clause of a window, unlike MATCH_RECOGNIZE. See Subclause 5.6, "PARTITION BY", and Subclause 5.7, "ORDER BY".

3) The ORDER BY clause may use scalar value expressions, not just columns. See Subclause 5.7, "ORDER BY".

4) The options ONE ROW PER MATCH and ALL ROWS PER MATCH are not applicable to windows, and cannot be specified. (Row pattern recognition in windows is closer in spirit to ONE ROW PER MATCH, though it also has some similarity to ALL ROWS PER MATCH WITH UNMATCHED ROWS.)

5) Row pattern recognition in a window has a choice between INITIAL and SEEK.

6) The MATCH_NUMBER function is not supported.

7) Row pattern measures are not columns in the result of a window; instead, row pattern measures are referenced using OVER, like a window function.

5.3 Row pattern input table

The row pattern input table for row pattern recognition in a WINDOW clause is the result of the FROM, WHERE, GROUP BY, and HAVING clauses that precede the WINDOW clause.

The example in Subclause 5.1, "Example of row pattern recognition in a window", does not have WHERE, GROUP BY, or HAVING clauses, so the row pattern input table in that example is the result of the FROM clause, that is, the table Ticker.

5.4 Row pattern variables and other range variables

There are two sets of range variables in a window with row pattern recognition:

1) The range variables declared in the FROM clause. (In the example in Subclause 5.1, "Example of row pattern recognition in a window", T is such a range variable.)

   These range variables may be used in the PARTITION BY and ORDER BY clauses, and of course in the SELECT list, WHERE clause, *etc.*

2) The row pattern variables declared in the PATTERN and SUBSET clauses. (In the example in Subclause 5.1, "Example of row pattern recognition in a window", A, B, C, and U are row pattern variables.)

   Row pattern variables may be referenced in the MEASURES, DEFINE, and SUBSET clauses. They cannot be used in the SELECT list.

Note that the two sets of range variables are declared in different clauses and have mutually exclusive scope. (Since they are walled off in mutually exclusive scopes, it is permitted to use the same range variables in each scope, though that is a confusing possibility that it is probably best to avoid.)

For example, the following is a syntax error:

```
SELECT Runlength OVER W
FROM Ticker T
WINDOW W AS ( PARTITION BY Symbol
              ORDER BY Tradeday
              MEASURES COUNT (T.*) AS Runlength
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              PATTERN (A*)
              DEFINE A AS T.Price > PREV (T.Price) )
```

There are three syntax errors in this example. The first is COUNT (T.*). T is a range variable defined in the FROM clause and cannot be referenced in MEASURES. Instead of T, the variable to use here is A, since A is declared in the PATTERN. Similarly, in the DEFINE, the two instances of T are errors.

Any column names to be referenced in either the MEASURES or DEFINE must be unique across the entire FROM clause, because the range variables in the FROM clause are not available to disambiguate in MEASURES or DEFINE. The workaround is to rename column names in the FROM clause as necessary to remove ambiguities.

For example, suppose both Emp and Dept have a column called Name. In the following query:

```
SELECT Aname OVER W, Bname OVER W
FROM Emp AS E, Dept AS D
WHERE E.Deptno = D.Deptno
WINDOW W AS ( PARTITION BY E.Deptno
              ORDER BY E.Empno
              MEASURES A.Name AS Aname,
                       B.Name AS Bname
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
```



```
              PATTERN (A B)
              DEFINE B AS B.Salary > A.Salary )
```

The expressions A.Name and B.Name in the MEASURES clause are ambiguous, because they could refer to either Emp.Name or Dept.Name. The solution is to rename at least one of them in the FROM clause, like this:

```
SELECT Aname OVER W, Bname OVER W
FROM ( SELECT Name AS Ename, Deptno, Salary
       FROM Emp ) AS E, Dept AS D
WHERE E.Deptno = D.Deptno
WINDOW W AS ( PARTITION BY E.Deptno
              ORDER BY E.Empno
              MEASURES A.Name AS Aname,
                       B.Name AS Bname
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              PATTERN (A B)
              DEFINE B AS B.Salary > A.Salary )
```

Note that row pattern variables are not available in the SELECT list. The following example is a syntax error:

```
SELECT A.Price
FROM Ticker AS T
WINDOW W AS ( PARTITION BY Symbol
              ORDER BY Tradeday
              MEASURES COUNT (A.*) AS Runlength
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              PATTERN (A*)
              DEFINE A AS A.Price > PREV (A.Price) )
```

In this example, A is a row pattern variable, which makes it visible in MEASURES and DEFINE. A is not visible in the SELECT list. There is no loss of expressive power; any expression of row pattern variables can be placed in MEASURES and then referenced by its measure name, like this:

```
SELECT Lasta OVER W
FROM Ticker AS T
WINDOW W AS ( PARTITION BY Symbol
              ORDER BY Tradeday
              MEASURES LAST (A.Price) AS Lasta
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              PATTERN (A*)
              DEFINE A AS A.Price > PREV (A.Price) )
```

5.5 Windows defined on windows

[ISO9075-2] allows one window to be defined on another window by referencing an existing window name. For example:

```
FROM Ticker AS T
WINDOW W1 AS ( PARTITION BY Symbol ),
       W2 AS ( W1 ORDER BY Tradeday ),
```

```
       W3 AS ( W2 ROWS BETWEEN CURRENT ROW
                      AND UNBOUNDED FOLLOWING )
```

Here window W2 inherits its partitioning from W1, and W3 inherits its partitioning and ordering from W2.

As an example of this capability using row pattern recognition:

```
SELECT Lastprice OVER W3
FROM Ticker AS T
WINDOW W1 AS ( PARTITION BY Symbol ),
       W2 AS ( W1 ORDER BY Tradeday ),
       W3 AS ( W2 MEASURES LAST(A.Price) AS Lastprice
                  ROWS BETWEEN CURRENT ROW
                       AND UNBOUNDED FOLLOWING
                  PATTERN (A+)
                  DEFINE A AS A.Price > PREV (A.Price) )
```

It is not possible to further subdivide the window definitions. For example, it is not permitted to put ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING in one window definition, and then inherit that in another window definition that adds the row pattern recognition features.

5.6 PARTITION BY

PARTITION BY is almost the same in windows and in MATCH_RECOGNIZE. The one difference is that range variables declared in the FROM clause are available in the PARTITION BY of a window, but not in MATCH_RECOGNIZE. Note that a row pattern partition is the same thing as a window partition when performing row pattern recognition in a window.

5.7 ORDER BY

ORDER BY is almost the same in windows and in MATCH_RECOGNIZE. The differences are:

1) Range variables declared in the FROM clause are available in the ORDER BY of a window, but not in MATCH_RECOGNIZE.

2) General scalar value expressions may be used in the ORDER BY of a window, but only column references may be used in the ORDER BY in MATCH_RECOGNIZE.

5.8 MEASURES

Row pattern measures in a window definition differ from row pattern measures in MATCH_RECOGNIZE as follows:

1) The MATCH_NUMBER function is not supported in windows.

2) Row pattern measures are referenced as window functions in the SELECT list using OVER, not as column references.

3) There is no real distinction between running and final semantics. The RUNNING and FINAL keywords may be used with aggregates, FIRST, and LAST, but the semantics is the same no matter which keyword is used. Row pattern measures are computed positioned on the last row of the match, where running and final semantics are identical.

5.9 Full window frame and reduced window frame

A window associates with each row *R* a set of rows, called the window frame of *R*. The definition of the window frame is essentially a subtractive process:

1) At the outset, there is the entire window partition that contains *R*.

2) Next, zero or more rows are removed from the window partition, based on their position relative to *R* in the ordering of rows of the window partition. The criterion at this stage is called the "window frame extent".

3) Next, zero or more rows are removed, based on peer relationships to *R*, using the EXCLUDE clause.

The three steps above are used for all windows. The result is called the "full window frame". When row pattern recognition is used, the window partition is also the row pattern partition, and there is one more step:

4) A match to the row pattern is sought within the full window frame; the rows that are mapped by this match (if any) constitute the "reduced window frame". If there is no match, the reduced window frame is empty. (Skipped rows can also cause an empty reduced window frame; see Subclause 5.15, "Empty matches and empty reduced window frames".)

5.9.1 ROWS BETWEEN CURRENT ROW AND

When performing row pattern recognition in a window, only two options are allowed for specifying the window frame extent:

— ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING: this option specifies that the full window frame consists of the set of rows from the current row through the end of the row pattern partition.

— ROWS BETWEEN CURRENT ROW AND *offset* FOLLOWING: this option specifies that the full window frame extends from the current row through some positive offset, which must be a positive integer, and specifies the number of rows after the current row. For example, ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING specifies a full window frame with 2 rows, the current row and the one after it.

5.9.2 EXCLUDE NO OTHERS

The window EXCLUDE clause has four possibilities:

1) EXCLUDE CURRENT ROW: this is not permitted with row pattern recognition, since the design is that the full window frame must begin with the current row.

2) EXCLUDE GROUP: also not permitted with row pattern recognition, because this would exclude the current row, plus any ties under the window ordering.

3) EXCLUDE TIES: not permitted with row pattern recognition, because this could create a hole in the full window frame, which is contrary to the spirit of row pattern recognition.

4) EXCLUDE NO OTHERS: permitted with row pattern recognition. This is the default.

Thus the only permitted option with row pattern matching is the default, EXCLUDE NO OTHERS.

5.10 AFTER MATCH SKIP

The options for AFTER MATCH SKIP are the same as in MATCH_RECOGNIZE; see Subclause 3.11, "AFTER MATCH SKIP", for details.

As in MATCH_RECOGNIZE, it is a run-time error to skip to a non-existent row, or to skip to the first row of a match.

Since only one row pattern match per full window frame is sought, the semantics of AFTER MATCH SKIP in a window are as follows. Windows are processed in the sort order within a row pattern partition. If a row *R* is skipped as a consequence of a row pattern match in a full window frame prior to *R*, then the reduced window frame of *R* is set to empty, without attempting any row pattern match for *R*. This is illustrated in an example in Subclause 5.15, "Empty matches and empty reduced window frames".

5.11 INITIAL vs. SEEK

If a row *R* has been skipped by a prior row *PR*, then the reduced window frame of *R* is empty.

If *R* has not been skipped, then a row pattern match is attempted in the full window frame of *R*. INITIAL and SEEK are two options that determine where to look for a match within the full window frame:

1) INITIAL is used to look for a match whose first row is *R*.

2) SEEK is used to permit a search for the first match anywhere from *R* through the end of the full window frame.

In either case, the reduced window frame comprises the rows that are mapped by the match; if there is no match, then the reduced window frame is empty. For a worked example, see Subclause 5.15, "Empty matches and empty reduced window frames".

The keyword INITIAL or SEEK is placed as a modifier before the PATTERN. The default is INITIAL.

5.12 PATTERN

This clause is precisely the same as in MATCH_RECOGNIZE, except that the anchors (^ and $) are not permitted with row pattern matching in windows. See Subclause 3.12, "PATTERN", for details.

5.13 SUBSET

This clause is precisely the same as in MATCH_RECOGNIZE. See Subclause 3.13, "SUBSET", for details.

5.14 DEFINE

This clause is precisely the same as in MATCH_RECOGNIZE. See Subclause 3.14, "DEFINE", for details.

5.15 Empty matches and empty reduced window frames

An empty match will cause the reduced window frame to be empty. Empty reduced window frames can also arise if there is no match at all, as in these circumstances:

1) AFTER MATCH SKIP on a prior row has caused the current row to be skipped, so no match is attempted.

2) The query specifies or implies INITIAL and there is no match starting at the current row.

3) The query specifies SEEK but there is no match anywhere between the current row and the end of the full window frame.

So there are two ways to get an empty reduced window frame: by finding an empty match, or by not finding a match at all.

The semantics for row pattern measures of empty reduced window frames are shown in this table:

Table 19 — Results for empty match and no match

Measure	**empty match**	**no match**
CLASSIFIER ()	null	null
COUNT	0	null
other aggregates (*e.g.*, SUM, AVG, *etc.*	null	null
row pattern navigation operations (*e.g.*, PREV, NEXT, FIRST, LAST	null	null
ordinary column references	null	null

Thus COUNT (*) may be used to distinguish an empty match from no match at all. If an empty match is found, then COUNT (*) as a row pattern measure will be 0; if there is no match at all, then COUNT (*) as a row pattern measure will be null.

Note the following subtlety: If the query specifies COUNT (*) as a non-measure window function, then the count over an empty window frame is 0 in any case. It is only when COUNT (*) is used as a row pattern measure that it can be used to distinguish an empty match from no match at all. For example:

```
SELECT S, D,
       Kount OVER W AS "Measure",
       COUNT (*) OVER W AS "Window Function"
FROM T
WINDOW W AS ( ORDER BY S
              MEASURES COUNT (*) AS Kount
              ROWS BETWEEN CURRENT ROW
                   AND UNBOUNDED FOLLOWING
              AFTER MATCH SKIP PAST LAST ROW
              INITIAL PATTERN (A*)
              DEFINE A AS A.D = 'yes' )
```

Consider the following data, shown in the first two columns, with the other two columns of output shown in the next two columns, and the internal information (skip indicator, whether a match was found, and the reduced window frame) in the right three columns:

Table 20 — Computation of matches and window function results

result table				**internal information**		
S	**D**	**Measure**	**Window Function**	**skipped?**	**match found?**	**reduced window frame**
1	yes	2	2	*False*	yes	{ Rows 1, 2 }
2	yes	null	0	*True*	no	{ }
3	no	0	0	*False*	yes	{ }
4	no	0	0	*False*	yes	{ }
5	yes	3	3	*False*	yes	{ Rows 5, 6, 7 }
6	yes	null	3	*True*	no	{ }
7	yes	null	0	*True*	no	{ }

Notes:

1) On Row 1, a match of length 2 is found. The window frame consists of rows 1 and 2. COUNT (*) as row pattern measure and COUNT (*) as non-measure window function both compute 2. Row 1 is not skipped, but Row 2 is skipped because of AFTER MATCH SKIP PAST LAST ROW.

2) On Row 2, this row was skipped as a result of the match on Row 1. Consequently the window frame is empty. COUNT (*) as row pattern measure is null because the row was skipped, whereas COUNT (*) as non-measure window function is 0 because the window frame is empty.

3) On Row 3, the engine is at a row that was not skipped, so it looks for an initial pattern matching A*. An empty match is found, so the window frame is empty but the row is not skipped. COUNT (*) as row pattern measure and COUNT (*) as non-measure window function are both 0. With an empty match, no rows are marked as skipped.

4) Row 4 shows a row following an empty match; its skipped indicator is not set. Once again there is an empty match, so the behavior is the same as at Row 3.

5) At Row 5, the skipped indicator is not set. There is a match of length 3. The window frame consists of Rows 5 through 7. COUNT (*) as row pattern measure and COUNT (*) as non-measure window function both compute 3. The skipped indicator for Row 6 and Row 7 is set.

6) Row 6 and Row 7 have the skipped indicator set, so they behave like Row 2.

5.16 Prohibited nesting

Row pattern recognition in windows is subject to the same restrictions on nesting as in MATCH_RECOGNIZE:

1) Nesting one row pattern recognition within another is prohibited.

2) Outer references in MEASURES or DEFINE are prohibited. This means that a row pattern recognition may not reference any table in an outer query block except the row pattern input table. (The row pattern input table is referenced using row pattern variables.)

3) Subqueries in MEASURES or DEFINE may not reference row pattern variables.

4) Row pattern recognition may not be used in recursive queries.

These restrictions are illustrated in the following Subclauses.

5.16.1 Row pattern recognition nested within another row pattern recognition

Nesting one row pattern recognition within another is prohibited. For example, the following is a syntax error:

```
SELECT ...
FROM Ticker
     MATCH_RECOGNIZE (
         ORDER BY ...
         MEASURES ...
         PATTERN ...
         DEFINE A AS 0 = ( SELECT Kount OVER W
                           FROM Stock2
                           WINDOW W AS (
                              ...
                              MEASURES COUNT (B.*) AS Kount
                              ...
                          )
                     ) AS M
```

A possible workaround is to relegate the nested row pattern recognition to a view or SQL-invoked function.

5.16.2 Outer references within a row pattern recognition query

The following is a syntax error (note the underlined range variables T):

```
SELECT ( SELECT Avg_Price OVER W
         FROM Ticker
         WINDOW W AS (
             ORDER BY Tradeday
             MEASURES AVG (X.Price) AS Avg_Price
             PATTERN (X+)
             DEFINE X AS T.Price >= AVG (X.Price) )
       )
FROM Toast AS T
```

In the preceding example, the column reference T.Price in the DEFINE clause is an outer reference to the range variable T defined in the outer block; therefore this example is a syntax error.

It may be possible to work around this limitation by placing the row pattern recognition in an SQL-invoked routine, passing as arguments the values that are prohibited as outer references.

5.16.3 Conventional query nested within row pattern recognition query

A subquery can be nested in an expression in MEASURES or DEFINE. Subqueries are permitted if they do not perform row pattern recognition themselves, and if they do not reference the row pattern variables of the outer query. Here is an example of the latter (note underlined A)

```
SELECT Firstday OVER W
FROM Ticker
WINDOW W AS (
    ORDER BY Tradeday
    MEASURES A.Tradeday AS Firstday
    PATTERN (A B+)
    ROWS
      BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
    DEFINE A AS A.Price > 100,
           B AS B.Price <
                ( SELECT AVG (S.Price)
                  FROM Ticker S
                  WHERE S.Tradeday BETWEEN A.Tradeday - INTERVAL '1' YEAR
                                       AND A.Tradeday )
            )
```

In this example, the definition of B involves a subquery that is correlated with the row pattern variable A (note underlining). This is a syntax error, since subqueries of row pattern matching may not reference row pattern variables.

5.16.4 Recursion

Row pattern matching is prohibited in recursive queries. For example, the following is a syntax error:

```
CREATE RECURSIVE VIEW Problem (Kolo, Xoro) AS
    SELECT Kolo, Xoro
    FROM T
      UNION
```



```
        SELECT Kolo + 1, Xoro OVER W
        FROM Problem
        WINDOW W AS (
            ORDER BY Kolo
            MEASURES COUNT (A.*) AS Xoro
            ROWS
              BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
            PATTERN (A+)
            DEFINE A AS A.Xoro > PREV (A.Xoro)
                    )
```

5.16.5 Concatenated row pattern recognition

Note that it is permitted to feed the output of one row pattern recognition into the input of another, as in this example:

```
SELECT Lastcount OVER W2
FROM ( SELECT Tradeday, Kount OVER W1 AS Innercount
       FROM Ticker
       WINDOW W1 AS (
           ...
           MEASURES COUNT (A.*) AS Kount) )
       WINDOW W2 AS (
           ...
           MEASURES LAST (A.Innercount) AS Lastcount
...  )
```

(Blank page)

6 Pattern matching rules

6.1 Regular expression engines

Regular expression engines are commonly classified using the terms DFA, traditional NFA and Posix NFA. The first question to address is which of these types is most appropriate to row pattern matching.

NOTE 5 — Regular expressions as defined in the theory of formal languages and automata may be recognized by either determinstic finite state automata or, equivalently, by non-determinstic finite state automata. Regular expressions as defined in commonly available software products are frequently more powerful than the original, mathematically defined regular expressions, particularly because of back references. While DFA technology in practice is reasonably similar to deterministic finite state automata in theory, NFA technology is generally more powerful than non-deterministic finite state automata, and does not have the same computational complexity characteristics.

DFA technology is attractive because, after compiling the regular expression into a DFA, the time to run the algorithm is linear in the length of the input, with bounded memory requirement independent of the length of the input. However, row pattern recognition is not amenable to a DFA solution. Note that row pattern recognition, while it uses the notation of regular expressions to express patterns, is actually a far richer capability, because the row pattern variables may be defined so as to depend upon previous matches. This dependency defeats the algorithm used to convert regular expressions into a DFA. The algorithm (described in standard texts) requires that at any point during pattern recognition, the next step depends only on the current state and the character currently being examined; *i.e.*, the history of how the machine got to its present state and location is immaterial. While this is true for classic regular expression recognition (*i.e.*, without backreferences), the technique breaks down when the outcome depends upon the prior history of the pattern matching.

Indeed, although row pattern recognition lacks syntax usually associated with backreferences, queries similar to backreferencing can be written. Consider for example:

```
PATTERN ( A B )
DEFINE B AS B.V = A.V
```

This basically says to find two adjacent rows that have the same value in the column V. The primary row pattern variable B is performing the role of a backreference in this query.

In fact, row pattern recognition is an NP-complete problem. For example, row pattern recognition may be used to solve the subset sum problem, which is known to be NP-complete. The subset sum problem for a non-empty set *S* of integers is to find a non-empty subset of *S* whose sum is 0. To solve this problem using row pattern matching, let *T* be a table with one integer column *X* and with one row for each element of *S*, where the value of *X* in a row is the corresponding element of *S*. The following row pattern recognition may be used to solve the subset sum problem on *S*:

```
FROM T MATCH_RECOGNIZE (
           PATTERN (A | B)*? C
           SUBSET U = (A, C)
           DEFINE C AS SUM(U.X) = 0 )
```

If a match is found, then the set of rows mapped to U (*i.e.*, to either A or C), is a non-empty set whose sum is 0. Conversely, the pattern explores all possible non-empty subsets U of *S*, so if there is a solution, this pattern will find one. Thus, the subset sum problem of *S* has a solution if and only if the query finds a match.

The best known algorithms for the subset sum problem have a time complexity that is exponential in the size of the input; thus row pattern matching in the worst case will be exponential and consequently a conversion to a DFA (which would have a linear time complexity) is not to be expected.

As for space complexity, a pattern such as (A | B)* will in the worst case need to maintain bookkeeping information for each row of the row pattern partition; thus, the space complexity in the worst case will be linear in the length of the longest row pattern partition. Since there might be only one row pattern partition, this is really the same as being linear in the number of rows of the row pattern input table.

This leaves the choice between traditional NFA and Posix NFA. The difference between these is that a traditional NFA exits (declares a match) as soon as it finds the first possible match, whereas a Posix NFA is obliged to find all possible matches and then choose the "leftmost longest". There are examples that show that, even for conventional regular expression matching on text strings and without backreferences, there are patterns for which a Posix NFA is orders of magnitude slower than a traditional NFA. In addition, reluctant quantifiers cannot be defined in a Posix NFA, because of the lefmost longest rule.

Therefore it was decided not to use the Posix NFA model, which leaves the traditional NFA as the model for row pattern matching. Among available tools that use traditional NFA engines, Perl is the most influential; therefore Perl was adopted as the design target for pattern matching rules.

6.2 Parenthesized language and preferment

Internally, a traditional NFA has logic to systematically explore all possible mappings of the input to sequences of primary row pattern variables as permitted by the pattern. [ISO9075-2] uses a device called parenthesized language to model the internal states of a traditional NFA. The parenthesized language is totally ordered by a relation called preferment. This ordering mimics the order in which a traditional NFA would visit the various possible mappings permitted by a pattern. Taking Perl as the model, the rules of preferment are designed to specify the order in which Perl's regex engine considers various potential matches.

It is common to define the language generated by a regular expression. The definition in [ISO9075-2] differs from most other presentations by incorporating "special symbols" — "^", "$", "(" , ")", "-", "[", and "]" — in addition to the "alphabet" (primary row pattern variable names). The special symbols are used for bookkeeping. For example, the special symbols "[" and "]" correspond to the exclusion syntax `{- -}`, which would not be captured by a conventional approach to a regular language. The special symbols "(" and ")" are used to encode when a traditional NFA enters or leaves the state of matching certain subpatterns. The special symbol "-" encodes which alternative in an alternation is taken.

Note that the special symbols are <identifier>s, as are the row pattern variables; therefore, [ISO9075-2] calls the "words" in the regular language <identifier> strings. For readability, the special symbols are shown without the double quotes after this point.

For any <identifier> string in the parenthesized language, the special symbols show how the <identifier> string was derived from the row pattern. There are ambiguous row patterns for which there are multiple ways to derive the same sequence of primary row pattern variables, but the ambiguity is removed when looking at the parenthesized language.

For example, given the pattern


```
PATTERN (A | A)
```

there is only one string in the regular language (namely, A), but there are two derivations of that string, taking either the first or the second alternative. These two derivations are represented in the parenthesized language as follows:

```
( A - )
( - A )
```

These two elements of the parenthesized language represent the two possible states of the traditional NFA that recognizes PATTERN (A | A).

The conventional regular language can be obtained from the parenthesized language by simply removing all special symbols and discarding duplicate <identifier> strings. The resulting <identifier> strings are simply strings of primary row pattern variables.

The following subclauses describe how the parenthesized language and preferment are defined for each kind of pattern.

6.2.1 Alternation

The special symbol "-" is used to indicate which alternative is not taken. For example, given the pattern

```
PATTERN (V | W)
```

the parenthesized language has two <identifier> strings, listed below in descending order of preferment:

```
( V - )
( - W )
```

Following Perl, the first alternative is preferred over the second. This means that a match to V is tried first. If the match to V succeeds, then there is no need to attempt the match to W.

When the alternatives each have multiple matches, then the order of preferment is as follows: all matches to the first alternative (in their order of preferment) followed by all matches to the second alternative (in their order of preferment). For example, given the pattern:

```
PATTERN ( A{1,2} | B{2,3} )
```

the parenthesized language, in descending order of preferment, is:

```
( ( A A ) - )
( ( A ) - )
( - ( B B B ) )
( - ( B B ) )
```

6.2.2 Concatenation

The result of concatenation is enclosed in parentheses. For example, given the pattern:

```
PATTERN (V W)
```

the parenthesized language has a single element:

```
( V W )
```

Concatenation becomes more interesting when quantifiers or alternations are concatenated. For example:

```
PATTERN (( A | B ) ( C | D ))
```

which generates the following parenthesized language, listed in descending order of preferment:

```
( ( A - ) ( C - ) )
( ( A - ) ( - D ) )
( ( - B ) ( C - ) )
( ( - B ) ( - D ) )
```

Note that preferment uses a lexicographic ordering. In this example, since the original pattern was presented in alphabetic order, the four possible states encoded in the parenthesized language also appear in alphabetic order.

6.2.3 Quantification

The result of expanding a quantifier is enclosed in parentheses. For example, given the pattern:

```
PATTERN (V{2,4})
```

the parenthesized language, in descending order of preferment, is:

```
( V V V V )
( V V V )
( V V )
```

Since {2,4} is a greedy quantifier, the longer <identifier> strings are preferred over the shorter <identifier> strings. On the other hand, using a reluctant quantifier, the pattern:

```
PATTERN (V{2,4}?)
```

has the same parenthesized language, but the preferment is the reverse, as listed here in descending order:

```
( V V )
( V V V )
( V V V V )
```

A greedy unbounded quantifier such as *, + or {n,} has no most preferred <identifier> string, so the parenthesized language can not be enumerated in descending order of preferment. There is no first <identifier> string. In practice, this is not a problem because the number of rows in the row pattern partition provides an upper bound. (The problem of infinite iterations of an empty match is discussed in Subclause 6.2.7, “Infinite repetitions of empty matches”.)

When a quantifier is applied to a subpattern with multiple matches, then:

1) Lexicographic ordering is the first priority.

2) Length is used to decide preferment only when one <identifier> string is a substring of the other.

As an example, consider:

```
PATTERN ((A|B){1,2})
```

This has a greedy quantifier, so longer <identifier> strings are preferred over shorter ones, but the first priority goes to lexicographic ordering. The parenthesized language, in order of decreasing preferment, is:

```
( (A -) (A -) )
( (A -) (- B) )
( (A -)       )
( (- B) (A -) )
( (- B) (- B) )
( (- B)       )
```

Note that a match to a singleton A (on the third line above) is preferred over longer matches that begin with B, even though the quantifier is greedy. This is because matches beginning with A are preferred lexicographically to matches beginning with B.

As an example with a reluctant quantifier,

```
PATTERN ((A|B){1,2}?)
```

the parenthesized language, in order of decreasing preferment, is:

```
( (A -)       )
( (A -) (A -) )
( (A -) (- B) )
( (- B)       )
( (- B) (A -) )
( (- B) (- B) )
```

Here, note that the matches of length 2 beginning with A (on the second and third lines) are preferred over the shorter match to a singleton B (fourth line). Again, this is because the lexicographic order is the first priority, and length is the second priority.

These priorities can be derived by considering the construction of a traditional NFA. Essentially, each part of a regular expression is converted into a small machine, which tries the options dictated by the regular expression in a particular order. To handle nesting of regular expressions, these machines are nestable. To handle a nested regular expression such as (A|B)*?, the outer machine (for *?) has a loop that invokes the inner machine (for A|B). Each time the inner machine is entered, it creates a “thread”; the threads are maintained with a stack discipline. The stack of threads effectively enumerates the regular language of (A|B)*?. This enumeration proceeds in lexicographic order of the inner machine (A|B). Thus the first attempt is A, the next attempt is A A, the next attempt is A A A, and so forth. At any depth of the stack, A is always attempted before B. Thus the first priority in this enumeration is lexicographic; the second priority (imposed by the outer machine) is to prefer shorter strings, in the case of reluctant quantifiers, or longer strings, in the case of greedy quantifiers.

6.2.4 Exclusion

Exclusion is indicated in the SQL language by the operators {- and -}. The parenthesized language uses the special symbols "[" and "]" to correspond to SQL language {- and -}. The reason is because [ISO9075-2] uses set theoretic notation, which already uses the symbols { and }.

For example, consider:

```
PATTERN ( {- A -} B )
```

This pattern has a single element in its parenthesized language:

```
( [ A ] B )
```

6.2.5 Anchors

The anchors are represented in the parenthesized language by the special symbols “^” and “$”.

6.2.6 The empty pattern

The empty pattern is represented in the parenthesized language by a balanced pair of parentheses, like this:

```
( )
```

6.2.7 Infinite repetitions of empty matches

Consider a pattern such as:

```
PATTERN ((A??)* B)
```

which will be matched against a sequence of rows whose only feasible match is:

```
B
```

The inner pattern A?? can have an empty match, and in fact the empty match is the most preferred match because of the reluctant quantifier. The parenthesized language for A?? consists of two <identifier> strings:

```
( )
( A )
```

listed in order of decreasing preferment.

The parenthesized language for (A??)* is defined as all finite strings (length ≥ 0) of elements of the parenthesized language of A??, with a wrapper of ( ) around each string as bookkeeping. Thus, the shortest member of the parenthesized language of (A??)* is ( ), and there is no longest member. Preferment is defined as preferring longer strings, as explained in Subclause 6.2.3, “Quantification”.

For example, the following is a series of members of the parenthesized language of (A??)* in *increasing* order of preferment:

```
( )
( () )
( () () )
```



```
( () () () )
. . .
```

That is, the machine could go on recognizing the empty match an arbitrary number of times, before deciding to exit from the * quantifier. The more empty matches that are recognized, the better. Left unchecked, this would cause an infinite loop. Consequently traditional NFAs adopt some kind of stopping rule when an empty match occurs within a quantifier.

There are many possible algorithms to prevent infinite loops. [ISO9075-2] takes Perl as its model. Perl's rule for handling empty matches is to continue iterating the quantifier if the lower bound has not been met. Once the lower bound has been met, if the inner expression returns an empty match, then the quantifier stops looping. This means that an empty match can only occur in the following positions within the trace of a quantifier with lower bound *n*:

1) as the last match, or

2) as a match in a position between 1 and *n*–1.

Consider three examples:

```
PATTERN ((A?){0,3})

PATTERN ((A?){1,3})

PATTERN ((A?){2,3})
```

These three examples are chosen because lower bounds of 0 and 1 are handled almost identically, in contrast to lower bound 2 and greater.

To start, the parenthesized language of A? has two members:

```
STRA = ( A )
STRE = ( )
```

Now let's consider (A?){0,3}. The set of all <identifier> strings of *STRA* and *STRE* of length at most 3 is listed below:

— length 0:

```
STR00 = ( )
```

— length 1:

```
STR01 = ( STRA )
STR02 = ( STRE )
```

— length 2:

```
STR03 = ( STRA STRA )
STR04 = ( STRA STRE )
STR05 = ( STRE STRA )
STR06 = ( STRE STRE )
```

— length 3:

```
STR07 = ( STRA STRA STRA )
STR08 = ( STRA STRA STRE )
```

```
STR09 = ( STRA STRE STRA )
STR10 = ( STRA STRE STRE )
STR11 = ( STRE STRA STRA )
STR12 = ( STRE STRA STRE )
STR13 = ( STRE STRE STRA )
STR14 = ( STRE STRE STRE )
```

Some of these are not possible according to Perl's behavior, namely numbers 05, 06, 09, 10, 11, 12, 13, and 14, because they involve an occurrence of the empty match *STRE* that is neither last nor necessary to fill the lower bound (0).

Note that this does not exclude the acceptable case STR02, which has *STRE* in the last position. Similarly, *STR04* and *STR08* are acceptable, because the only occurrence of *STRE* in these strings is in the last position.

This leaves the following in the parenthesized language of (A?){0,3}:

— length 0:

```
STR00 = ( )
```

— length 1:

```
STR01 = ( STRA )
STR02 = ( STRE )
```

— length 2:

```
STR03 = ( STRA STRA )
STR04 = ( STRA STRE )
```

— length 3:

```
STR07 = ( STRA STRA STRA )
STR08 = ( STRA STRA STRE )
```

Next, consider (A?){1,3}. *STR00* is excluded because it does not meet the lower bound of 1. Otherwise, the analysis is just the same as for lower bound of 0.

Finally, consider (A?){2,3}. Now, strings *STR00*, *STR01*, and *STR02* are all excluded because they do not meet the lower bound of 2. On the other hand, some of the <identifier> strings that violate Perl rules for lower bounds 0 and 1 are acceptable for lower bound 2, because a non-final empty match *STRE* can be used to satisfy the lower bound in position 1. This means that numbers 05, 06, 11, and 12 become acceptable. (Numbers 09, 10, 13, and 14 are still unacceptable because they involve the empty match *STRE* at position 2, where they complete the lower bound and therefore no further iteration should occur.) Thus, the parenthesized language for (A?){2,3} has the following elements:

— length 2:

```
STR03 = ( STRA STRA )
STR04 = ( STRA STRE )
STR05 = ( STRE STRA )
STR06 = ( STRE STRE )
```

— length 3:


```
STR07 = ( STRA STRA STRA )
STR08 = ( STRA STRA STRE )
STR11 = ( STRE STRA STRA )
STR12 = ( STRE STRA STRE )
```

6.3 Pattern matching in theory and practice

For the most part, examples of pattern matching have been presented using an informal traditional NFA model. This Subclause reconsiders one of these examples in the light of the parenthesized language and preferement.

The last example in Subclause 4.2, "Running *vs.* final semantics", is:

```
PATTERN ( A? B+ )
DEFINE A AS A.Price > 100
       B AS B.Price > COUNT (A.*) * 50
```

performed on the following ordered row pattern partition of data :

Table 21 — Input data

Row	PRICE
R_1	60
R_2	70
R_3	40

Using an informal NFA model, this example is analyzed as follows:

1) Tentatively map $R = R_1$ to row pattern variable A. The Boolean condition for A is *False*, so this mapping does not succeed.

2) Since the mapping to A failed, the empty match is taken as matching A?.

   NOTE 6 — In common NFA terminology, this is called "backtracking", which means to revisit the last decision and try the next alternative for that decision, if any. When all alternatives for one decision have been exhausted, the traditional NFA will backtrack further, looking for an earlier decision with untried alternatives to consider.

3) Tentatively map $R = R_1$ to B. This mapping is successful.

4) Similarly, rows R_2 and R_3 can be successfully mapped to B.

5) Since there are no more rows, this is the complete match: no rows mapped to A, and { R_1, R_2, R_3 } mapped to B.

This problem can be analyzed using parenthesized language and preferment as follows. The pattern involves an unbounded greedy quantifier (B+); therefore, the parenthesized language is infinite and there is no first <identifier> string. However, any match must have at most three row pattern variables, reducing the effective

size of the parenthesized language to an enumerable finite set. The relevant <identifier> strings, in descending order of preferment, are as follows:

```
( ( A ) ( B B   ) )
( ( A ) ( B     ) )
( (   ) ( B B B ) )
( (   ) ( B B   ) )
( (   ) ( B     ) )
```

The first element in the parenthesized language gives this mapping:

Table 22 — Mapping of first element

Row	PRICE	mapping	Boolean condition
R_1	60	→ A	*False*
R_2	70	→ B	*True*
R_3	40	→ B	*False*

Since the Boolean condition for at least one row is *False*, this match is rejected.

NOTE 7 — A traditional NFA is not obliged to actually attempt the mappings to R_2 and R_3 because the mapping to R_1 failed. See step 1 above.

The second element of the parenthesized language gives the following mapping:

Table 23 — Mapping of second element

Row	PRICE	mapping	Boolean condition
R_1	60	→ A	*False*
R_2	70	→ B	*True*
R_3	40		

This proposed match is also rejected because the first row fails its Boolean condition.

NOTE 8 — A traditional NFA does not need to consider the second element of the parenthesized language at all. The traditional NFA already knows that mapping the first row to A fails; therefore, it can rule out any further pattern matches that begin with A. In effect, the traditional NFA prunes the set of possible matches when it rules out an initial A.

The third element of the parenthesized language gives the following mapping:

Table 24 — Mapping of third element

Row	PRICE	mapping	Boolean condition
R_1	60	→ B	*True*
R_2	70	→ B	*True*
R_3	40	→ B	*True*

All three rows now have a true Boolean condition. Therefore this is a successful match. Since the first and second elements of the parenthesized language failed, this is the most preferred match.

NOTE 9 — A traditional NFA does not need to consider the remaining elements in the parenthesized language; it simply "exits" when it finds a successful match.

(Blank page)

Index

Index entries appearing in **boldface** indicate the page where the word, phrase, or BNF nonterminal was defined; index entries appearing in *italics* indicate a page where the BNF nonterminal was used in a Format; and index entries appearing in roman type indicate a page where the word, phrase, or BNF nonterminal was used in a heading, Function, Syntax Rule, Access Rule, General Rule, Conformance Rule, Table, or other descriptive text.

— N —

— O —

— P —

— R —

— S —

— T —

— U —

— V —

— W —

— Y —

ICS 35.060
Price based on 80 pages