Specification of Row Pattern Recognition in the SQL Standard and its Implementations

Dušan Petković¹

Received: 13 September 2021 / Accepted: 24 January 2022 / Published online: 18 February 2022
© The Author(s) 2022

Abstract

Recently, the SQL standardization Committee published a specification for support of the concept called Row Pattern Recognition in SQL. That way, the focus for the members of the Committee turned back to a kernel issue of the language, after standardizing the storage and manipulation of data formats, such as XML and JSON. In this paper we discuss the specified features using several examples and show to what extent different relational database systems as well as Data Analytics tools have integrated them. At the end of the paper, we describe the main inaccuracies of the proposal and the ways how to solve them. From our point of view, the following should be modified in one of the future proposals of the standard in relation to this concept: (a) Naming of several pattern navigation operations is inappropriate and should be changed; (b) The concept of implicit definition for row pattern variables in DEFINE should be changed to explicit; (c) The set of existing functions should be extended.

1 Introduction

The ISO/IEC 9075:2016 standard [1] introduces support for a concept called Row Pattern Recognition as a part of SQL. Row Pattern Recognition (RPR) uses regular expressions to identify patterns in a sequence of rows. For each pattern match, it is possible either to return a summary row like in the GROUP BY clause of the SELECT statement, or to return detailed rows, analogous to the window functionality.

Row Pattern Recognition can be seen as the next step in the evolution of the window functionality, with a higher level of sophistication. Similar to the window functionality, RPR supports partitioning and ordering. Additionally, the notion of summary rows as well as detailed rows exists in both concepts and has the same meaning.

The necessity to recognize the patterns in table's rows exists in many business areas. Generally, areas using all kinds of business processes driven by *sequences of events* need this feature. RPR is used in security applications, where unusual behaviour must be detected, and financial applications, where you seek patterns of pricing and trading vol-

ume. Other common uses are finding anomalies in time series and sensor data analysis.

Row Pattern Recognition in the SQL standard comes in two flavours. The first one concerns the common search for patterns inside the rows of the table. For this issue, SQL has been extended with a new clause, called MATCH_RECOGNIZE, and the clause can be applied to a table expression and written as a part of the FROM clause of the SELECT statement. In that case, there are two different modes concerning matching: one that displays a single row for each match of the pattern and one that displays all rows that matched it. In both cases, calculations on variables are possible.

Row Pattern Recognition can also be done inside an existing window frame. Generally, a window partition restricts the rows of the input table and, after that, the window frame restricts the derived window partition, in the similar way. In other words, RPR restricts the full window frame to a reduced window frame made of the subsequence of rows constituting the pattern match.

1.1 Roadmap

The rest of the paper is organized as follows. Sect. 2 discusses the specification of Row Pattern Recognition inside the FROM clause of the SELECT statement as well as inside the WINDOW clause. Sect. 3 describes inaccuracies in the standard, and the possible extensions of it. Sect. 4

✉ Dušan Petković
petkovic@fh-rosenheim.de

¹ University of Applied Sciences,
Hochschulstr. 1, 83024 Rosenheim, Germany

```

SELECT <select list>
FROM <source table>
MATCH_RECOGNIZE
(
  [ PARTITION BY <partition list> ]
  [ ORDER BY <order by list> ]
  [ MEASURES <measure list> ]
  [ <row pattern rows per match> ::=
    ONE ROW PER MATCH | ALL ROWS PER MATCH ]
  [ AFTER MATCH <skip to option>
  PATTERN ( <row pattern> )
  [ SUBSET <subset list> ]
  DEFINE <definition list>
) AS <table alias>;

```

Fig. 1 The syntax of the MATCH_RECOGNIZE clause

explains which relational database systems and Data Analytics tools have implemented Row Pattern Recognition and to what extent. Additionally, the adjustment of the implementation of each system with the specification in the SQL standard will be considered. At the end, in the summary, we discuss the ways how the current proposal can be improved in one of future specifications of the SQL standard.

2 Row Pattern Recognition in the SQL Standard

The SQL:2016 standard introduces the MATCH_RECOGNIZE clause in the SQL SELECT statement to support a concept called Row Pattern Recognition.

Table 1 The tabular form of the **stock_price** table

Company	Price_date	Price
A	2020-10-01	50
B	2020-10-01	89
A	2020-10-02	36
B	2020-10-02	24
A	2020-10-03	39
B	2020-10-03	37
A	2020-10-04	42
B	2020-10-04	63
A	2020-10-05	30
B	2020-10-05	65
A	2020-10-06	47
B	2020-10-06	56
A	2020-10-07	71
B	2020-10-07	50
A	2020-10-08	80
B	2020-10-08	54
A	2020-10-09	75
B	2020-10-09	30
A	2020-10-10	63
B	2020-10-10	32

Fig. 2 Stock chart for the **stock_price** table

This concept is described in SQL/Foundation (Part 2 of the SQL standard) [2]. Also, the short description of this feature can be found in [3]. The SQL standard describes three features concerning Row Pattern Recognition:

- Feature R010, “Row pattern recognition: FROM clause”
- Feature R020, “Row pattern recognition: WINDOW clause”
- Feature R030, “Row pattern recognition: Full aggregate support”

Note that the last feature will not be discussed further, because there is no detailed description of it in the SQL standard. This feature defines the conditions for specifying aggregate functions (MIN, MAX, SUM, COUNT, or AVG) inside row pattern measures or row pattern definition search conditions.

2.1 Row Pattern Recognition: Introduction

The first part of the SQL standard concerning Row Pattern Recognition describes the MATCH_RECOGNIZE clause. The syntax of this clause in a SQL query is given in Fig. 1. All subclauses of the MATCH_RECOGNIZE clause will be discussed below, using several examples.

We use the **stock_price** table introduced in Example 1 to illustrate how the MATCH_RECOGNIZE clause and all its subclauses can be applied for stock markets. The **stock_price** table contains three columns, **company**, **price_date** and **price**, and represents a history of stock prices. The **company** column specifies the name of the company, **price_date** the date and the **price** column the price of the stock on the particular day for the particular company.

Example 1

```
-- Creates the stock_price table
CREATE TABLE stock_price
(company CHAR(50), price_date DATE, price INT);
```

Table 1 shows the content of the **price_table** in the tabular form, after inserting twenty rows. Fig. 2 shows the content of the same table in the graphical form.

2.2 PATTERN and DEFINE Subclauses

As can be seen from Fig. 1, the only subclauses that are required in the MATCH_RECOGNIZE clause are PATTERN and DEFINE, which will be explained in the example below. The goal of Example 2 is twofold: First, it applies the PATTERN and DEFINE subclauses to show how all stock prizes equal to 50 can be found. Second, to show that an ordinary SELECT statement can be also used to solve elementary tasks concerning row pattern recognition.

Example 2

```
SELECT mr.*
FROM stock_price
MATCH_RECOGNIZE
(all rows per match
 pattern (fifty)
 define
 fifty as fifty.price = 50
) mr
ORDER BY price_date;
```

Note that the query in Example 2 is equivalent to the following SELECT statement:

```
SELECT * FROM stock_price WHERE
price = 50 ORDER BY price_date;
```

The result is:

Company	Price_date	Price
A	2020-10-01	50
B	2020-10-07	50

The PATTERN subclause specifies the pattern to search. It includes one or more variables. In example above, the single pattern variable called **fifty** is specified.

The DEFINE subclause comprises the definition of pattern variables. Each pattern variable specifies a different condition for tables' rows. In example above, there is one condition: the value of the **price** column should be equal to 50. Note that there can exist pattern variables without

the corresponding condition in the DEFINE subclause. In that case, the variable matches any row (see Example 5, for instance).

All rows processed with the MATCH_RECOGNIZE clause belong to row pattern input table. The result table of the MATCH_RECOGNIZE clause is called row pattern output table. The form of this table depends on the choice between ONE ROW PER MATCH and ALL ROWS PER MATCH: If the former option is specified, the row pattern output table displays a single summary row for each match of the pattern. (This subclause is similar to the outcome of the GROUP BY clause and is the default value.) If the latter is specified, the columns of the row pattern output table will have one row for each row of each match. In other words, detailed rows will be displayed.

Optionally, an alias name may be assigned to the row pattern output table. This is called row pattern output name. (In Example 2, the alias name of the row pattern output table is **mr**.) The use of an alias name is important when ambiguous column names have to be resolved. This happens when there are several tables in the same FROM clause.

2.3 PARTITION BY and ORDER BY Subclauses

The PARTITION BY subclause specifies how to partition the row pattern input table. This subclause contains a list of columns of the row pattern input table. If PARTITION BY is omitted, there are no row pattern partitioning columns, and the entire input table constitutes a single row pattern partition.

The ORDER BY subclause specifies how to order the rows within row pattern partitions. It contains one or more columns of the row pattern input table. Although the subclause is optional, it is strongly recommended to use it, because non-deterministic ordering will destroy the purpose of using row pattern recognition.

Example 3 uses the PARTITION BY subclause to partition the rows by company, so that the search across each company's stock prices is possible. Additionally, using the ORDER BY subclause, we sort the rows within each partition by company's name and date. That way, it is easy to find changes to a company's stock price over time.

Example 3

```
SELECT company, price_date, price
FROM stock_price
MATCH_RECOGNIZE (
partition by company
order by price_date
all rows per match
pattern (limit_50 decrease_10 increase_5)
define
```

```

        limit_50 as price <= 50.00,
        decrease_10 as prev(price) * 0.90 >= price,
        increase_5 as prev(price) * 1.05 <= price
    )
ORDER BY company, price_date;

```

The result is:

Company	Price_date	Price
A	2020-10-01	50
A	2020-10-02	36
A	2020-10-03	39
A	2020-10-04	42
A	2020-10-05	30
A	2020-10-06	47

Example 3 shows the way how to find sequences of the following three events:

- On a given day, the stock price for a company is less or equal to 50
- On the next day, the stock price for that company decreases by at least 10%
- On the following day, the stock price for that company increases by at least 5%

To find this sequence, a pattern that matches three rows with the following conditions must be specified:

- In the first row of the sequence, the value of the **price** column must be less or equal to 50.
- In the second row, the value of the **price** column must be less than or equal to 90% of the value of the previous row.
- In the third row, the value of the price column must be greater than or equal to 105% of the value of the previous row.

Therefore, three variables are needed, and they have been specified in the PATTERN subclause of Example 3: **limit_50**, **decrease_10** and **increase_5**. After that, the following conditions for these three pattern variables must be defined in the DEFINE subclause:

```

limit_50 as price <= 50,
decrease_10 as prev(price) * 0.90 >= price,
increase_5 as prev(price) * 1.05 <= price

```

The second and the third definition of conditions for pattern variables use the pattern navigation operation called PREV. This function evaluates the value of the pattern variable of the previous row and use it for further calculations. Therefore, in Example 3, the value of the **price** column in the previous row will be multiplied with the value 0.9 to

test the condition (b) above. Also, the value of this column will be multiplied with 1.05, to test the condition (c) above.

2.4 Regular Expressions

Row Pattern Recognition is based upon regular expressions. In Example 4 we use three different PATTERN subclauses to show different regular quantifiers and their applications (The detailed discussion of regular expressions can be found in [4].).

Example 4

```

-- This example has three different patterns.
-- Comment two of them, to get error-free query.
SELECT company, price_date, price
FROM stock_price
MATCH_RECOGNIZE (
    partition by company
    order by price_date
    all rows per match
    -- pattern ( limit_50 up+ down+)
    pattern ( limit_50 up up* down down* )
    -- pattern ( limit_50 up {- up* -} down {- down* -} )
    define
        limit_50 as price <= 50.00,
        up as price > prev(price),
        down as price < prev(price)
    )
WHERE company = 'B'
ORDER BY price_date;

```

Example 4 shows the way how to find sequences of the following three events:

- On a given day, the stock price for the company B is less or equal to 50
- On the following one or more days, the stock price for that company rises
- On the following one or more days, the stock price for that company falls

The first and second form of the PATTERN subclause are equivalent. So, for both patterns, the row pattern output table is:

Company	Price_date	Price
B	2020/10/02	24
B	2020/10/03	37
B	2020/10/04	63
B	2020/10/05	65
B	2020/10/06	56
B	2020/10/07	50

* – zero (0) or more matches
 + – one (1) or more matches
 ? – no match or one (1) match, optional
 { n } – exactly n matches
 { n, } – n or more matches
 { n, m } – between n and m (inclusive) matches
 { , m } – between zero (0) and m (inclusive) matches
 {-Variable -}, e.g., {-A-} – indicates that matching rows are to be excluded from the output
 |, e.g., A | B – alternation
 (), e.g., (A | B) – grouping
 ^, e.g., ^A{1, 3} – start of a row pattern partition
 \$, e.g., A{1, 3}\$ – end of a row pattern partition

Fig. 3 The list of regular expression quantifiers

The output table of Example 4 displays exactly one sequence of events specified in the DEFINE subclause. In other words, the first row in the output table fulfils the first condition (limit_50 as price ≤ 50.00). The next three output rows fulfil the second one (up as price > prev(price), and the last two rows fulfil the third condition (down as price < prev(price)).

Such an output table can be quite large. In that case the use of the exclusion quantifier “{- var -}” can reduce the output significantly. (Note that the exclusion quantifier does not change the semantics of a query: it changes only the content of the row pattern output table.) The third pattern:

```
pattern (limit_50 up {- up* -} down {- down* -})
```

reduces the output to the following table:

Company	Price_date	Price
B	2020-10-02	24
B	2020-10-03	37
B	2020-10-06	56

The first row of the output table above will be displayed, because the **limit_50** variable has not been excluded. The following two patterns (up {- up* -}) display the first day, where the stock rises, and exclude all other following rows with the same property. The last two patterns (down {- down* -}) display the first day, when the stock price falls, and exclude all other following rows with the same property. Fig. 3 shows regular expression quantifiers, which can be used for Row Pattern Recognition.

2.5 Functions in the MEASURES Subclause

Example 5 introduces two functions: MATCH_NUMBER and CLASSIFIER, which are often used together. The pattern in this example searches for series of subsequent rows, where the stock price of the company “A” rises.

Example 5

```

SELECT price_date, matchno, first_price, cl, price
from (select * from stock_price where company='A')
MATCH_RECOGNIZE (
  order by price_date
  measures
    match_number() as matchno,
    classifier() as cl,
    first(strt. price) as first_price
  all rows per match
  pattern(strt up+)
  define
    /* strt defaults to TRUE, matches any row */
    up as price > prev(price) )
ORDER by matchno, price_date ;

```

The result is:

Price_date	Matchno	First_price	Cl	First_price
2020-10-02	1	36	STRT	36
2020-10-03	1	36	UP	39
2020-10-04	1	36	UP	42
2020-10-05	2	30	STRT	30
2020-10-06	2	30	UP	47
2020-10-07	2	30	UP	71
2020-10-08	2	30	UP	80

Measures are additional columns of data that are calculated for each matching sequence of rows. They provide additional semantics for evaluating scalar expression in relation to row pattern matches. The MEASURES subclause defines named expressions that are accessed using so called row pattern measure functions. In Example 5, there are three such functions: MATCH_NUMBER, CLASSIFIER and FIRST.

When the result of the MATCH_RECOGNIZE clause contains multiple rows from multiple matches, it is necessary to know which rows are members of which match. In that case, MATCH_NUMBER is used. This function as-

signs the same number to each row of the common match. In Example 5, the value 1 will be assigned to the first sequence of rows, where the stock price of the company “A” rises. After that, the value 2 will be assigned to the second sequence of rows with the same property.

The FIRST function belongs to row pattern navigation operations, together with PREV and NEXT. In contrast to PREV and NEXT, FIRST navigates only among the rows mapped to the corresponding pattern variable. In other words, FIRST uses logical, not physical offsets. Therefore, in Example 5, FIRST returns the values of the expression evaluated in the first row of the first and second match.

Additionally, to know which rows map to which variable, the CLASSIFIER function is used. This function extends the semantics of the MATCH_NUMBER function and returns a value of the CHARACTER data type that contains the string that the respective row matched. For example, if a row matched the string ABCD, then the CLASSIFIER function returns “ABCD”.

The string “UP” will be assigned to the classifier **cl** for all rows, where the stock price rises, because in that case the condition “price > PREV(price)” is true. In all other cases, the string “STRT” will be assigned to the classifier, because the variable with the same name does not appear in the DEFINE subclause, and in that case, the variable is always true, i.e. matches any row.

2.6 SUBSET Subclause

The following example shows the use of the SUBSET subclause and the LAST row pattern navigation function. This example partitions the data by companies and sort them according to the values in the **price_date** column. After that, it searches for all strictly falling prices, followed by all strictly increasing prices. (This behaviour is called “V shape”).

Example 6

```
SELECT *
FROM stock_price
MATCH_RECOGNIZE (
  partition by company
  order by price_date
  measures
  match_number() AS matchno,
  A.price AS startp,
  last (up.price) AS bottomp,
  last (down.price) AS endp,
  avg (U.price) AS avgp
  one row per match
  after match skip past last row
  pattern (A up+ down+)
  subset U = (A, up, down)
```

```
define
up AS up.Price < prev (up.Price),
down AS down.Price > prev (down.Price)
) ;
```

The result is:

Company	Matchno	Startp	Bottomp	Endp	Avgp
A	1	50	36	42	41.75
B	1	89	24	65	55.6
B	2	56	50	54	53.3

As can be seen from the result of Example 6, all displayed rows are summary rows. The reason is that ONE ROW PER MATCH is used. Additionally, the example uses the AFTER MATCH SKIP option. This option specifies where to resume looking for the next match after successfully finding a previous one. The option has four different forms:

- AFTER MATCH SKIP TO NEXT ROW
- AFTER MATCH SKIP PAST LAST ROW
- AFTER MATCH SKIP TO FIRST pattern_var
- AFTER MATCH SKIP TO LAST pattern_var

The first form resumes pattern matching at the row after the first row of the current match. Similarly, the second form resumes pattern matching at the row after the last row of the current match (The default value is AFTER MATCH SKIP PAST LAST ROW.).

AFTER MATCH SKIP TO FIRST| LAST] pattern_var resumes pattern matching at the first (last) row that is mapped to pattern variable **pattern_var**. Therefore, in Example 6, this option starts at the next row, when the current match is ended (The default value is AFTER MATCH SKIP TO LAST pattern_var.).

The LAST function belongs to row pattern navigation operations. In contrast to PREV and NEXT, LAST navigates only among the rows mapped to the corresponding pattern variable. Therefore, LAST returns the values of the expression evaluated in the *last row* of rows mapped to the pattern variable. In Example 6, the LAST function has been used twice. The first LAST function calculates the last stock price in the sequence of strictly falling prices using the given condition. The second LAST function calculates the endpoint of the strictly increasing prices.

Sometimes it is necessary to create a grouping of multiple pattern variables that can be referred to with a variable name of its own. These groupings are called union row pattern variables and are defined using the SUBSET subclause.

In Example 6, we define U as the union of row pattern variables A, **up** and **down**, to calculate the average value of all stock prices for the sequence of all strictly falling prices,

followed by strictly increasing prices. Therefore, the union will contain one starting value, one or more values with falling prices and one or more values with rising prices. For instance, the only “V shape” for the company A contains altogether four values (50, 36, 39 and 42). The average stock price from this “V shape” is 41.75.

2.7 Empty Matches

The next example shows how empty matches can be handled as well as the use of the FINAL LAST option.

Example 7

```
SELECT price_date, matchno, firstp, lastp, price
FROM ( SELECT * FROM
      stock_price WHERE company = 'A')
MATCH_RECOGNIZE (
partition by company
order by price_date
measures match_number() AS matchno,
first (up.Price) AS firstp,
final last (up.Price) AS lastp
all rows per match show empty matches
after match skip past last row
pattern (up*)
define up AS up.price > prev (up.Price) );
```

The result is:

Price_date	Matchno	Firstp	Lastp	Price
2020-10-01	1	–	–	50
2020-10-02	2	–	–	36
2020-10-03	3	39	42	39
2020-10-04	3	39	42	42
2020-10-05	4	–	–	30
2020-10-06	5	47	80	47
2020-10-07	5	47	80	71
2020-10-08	5	47	80	80
2020-10-09	6	–	–	75
2020-10-10	7	–	–	63

Some patterns allow matches with zero rows. For instance, PATTERN (A*) can be matched by zero or more rows that are mapped to A. An empty match does not map any rows to row pattern variables; nevertheless, an empty match has a starting row. Depending how the empty matches should be handled, the ALL ROWS PER MATCH option has three different forms:

- ALL ROWS PER MATCH SHOW EMPTY MATCHES
- ALL ROWS PER MATCH OMIT EMPTY MATCHES
- ALL ROWS PER MATCH WITH UNMATCHED ROWS

With the ALL ROWS PER MATCH SHOW EMPTY MATCHES option, any empty match generates a single row in the row pattern output table. In contrast to it, with ALL ROWS PER MATCH OMIT EMPTY MATCHES, each empty match is omitted from the row pattern output table.

Sometimes, 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. These rows can be handled with the third option listed above: ALL ROWS PER MATCH WITH UNMATCHED ROWS. This option shows both empty matches and unmatched rows. Empty matches are handled the same way as with ALL ROWS PER MATCH SHOW EMPTY MATCHES. When displaying an unmatched row, all row pattern measures are null. Therefore, the functions, COUNT and MATCH_NUMBER may be used to distinguish an unmatched row from the starting row of an empty match.

Example 7 performs the search for sequences of increasing prices. (The PATTERN subclause contains one variable, up*, and the DEFINE subclause defines the condition for runs of increasing prices.) This example uses the FINAL LAST function, which returns the value for the last row mapped to the corresponding variable. Therefore, in Example 7 the program line

```
final last (up.Price) AS lastp
```

displays the last price in the sequence of rising prices.

The first and second row of the row pattern output table have empty values for the **firstp** and **lastp** columns, because the condition in DEFINE is not fulfilled. Nonetheless, the rows are displayed, because the ALL ROWS PER MATCH SHOW EMPTY MATCHES option is used in this example. The third row is the first row, where the condition is true. (The value of stock price increases from 36–39.) Therefore, all values of the output columns will be displayed.

If the ALL ROWS PER MATCH option in the example above is replaced with ALL ROWS PER MATCH OMIT EMPTY MATCHES, the row pattern output table lacks the rows with the empty matches, and is as follows:

Price_date	Matchno	Firstp	Lastp	Price
2020-10-03	3	39	42	39
2020-10-04	3	39	42	42
2020-10-06	5	47	80	47
2020-10-07	5	47	80	71
2020-10-08	5	47	80	80

2.8 PERMUTE Operator

The next example shows the use of the PERMUTE operator in the pattern.

Example 8

```
SELECT *
FROM stock_price MATCH_RECOGNIZE(
  partition by company
  order by price_date
  measures
    match_number() as no,
    first(up.price_date) as "start",
    last(up.price_date) as "end",
    -- count(up.price) as ups,
    -- the alias name ups has always the value 2
    count(*) as pr_ct
  after match skip to next row
  pattern(any_row permute(up{2}, down+))
  define
    up as up.price > prev(price),
    down as down.price < prev(price)
)
ORDER BY company;
```

The result is:

Company	No	Start	End	Pr_ct
A	1	2020-10-01	2020-10-04	4
A	2	2020-10-02	2020-10-05	4
A	3	2020-10-04	2020-10-07	4
A	4	2020-10-06	2020-10-10	5
B	1	2020-10-01	2020-10-04	4
B	2	2020-10-03	2020-10-07	5

The PERMUTE option is 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 pattern variables A, B, and C (six, altogether). Therefore, in Example 8, in the PERMUTE subclause:

```
permute(up{2}, down+))
```

there are two simpler patterns: up{2} and down+. The first one limits the number of rising prices in the sequence to two, while the second one is true when stock price falls for one or more days. The PERMUTE subclause permutes these two patterns, meaning that upward spikes can follow downward ones, and vice versa. Therefore, the pattern in Example 8 searches for all upward and downward spikes in

the stock price history, limiting the number of rising prices to two.

The row pattern output table above lacks two sequences of stock prices, one beginning on 2020-10-03 and the other one beginning on 2020-10-05. The prices in the first sequence rise until 2020-10-08 and then fall for two days. This sequence does not belong to the row pattern output table, because the number of days with rising prices is three. The same is true for the sequence for company A beginning on 2020-10-03, because, in this case, the number of days with rising prices is one.

2.9 Row Pattern Recognition: WINDOW Clause

The specification “Row pattern recognition: WINDOW clause” in the SQL Standard enhances the WINDOW clause to include row pattern matching. In the same way as the window partition restricts the rows of the input table, and the window frame further restricts the derived window partition, row pattern recognition in the WINDOW clause restricts the full window frame to a reduced window frame made of the subsequence of rows constituting the pattern match. The phrase “full window frame” in the standard specifies the result table after the WINDOW clause is applied, and the phrase “reduced window frame” specifies the row pattern output table, generated in the next step, when the specified pattern match is applied. The detailed specification can be found in [2].

Note that, when using row pattern recognition with the WINDOW clause, the full window frame must start at the current row.

The following example is semantically identical to Example 6, but applies the WINDOW clause to search for all “V shapes” for the companies A and B.

Example 9

```
SELECT company, price_date, price,
       Start_price over w,
       Bottom_price over w,
       End_price over w,
       Avg_price over w
FROM stock_price SP
WINDOW w AS
  ( partition by company
    order pby rice_date
    measures
      A.Price AS start_price,
      LAST (B.Price) AS bottom_price,
      LAST (C.Price) AS end_price,
      AVG (U.Price) AS avg_price
    rows between current row
      and unbounded following
```



```

after match skip past last row
  INITIAL
pattern (A B + C+)
subset U = (A, B, C)
define
  B AS B.Price < PREV (B.Price),
  C AS C.Price > PREV (C.Price) )
) ;

```

The projection operation in Example 9 contains, among other column names, window functions over the window *w*. “Start_price OVER *w*”, “Bottom_price OVER *w*”, etc. are examples of row pattern measure functions, which are window functions used to access corresponding measures defined in the MEASURES subclause. Therefore, each row pattern measure function specifies a reduced window frame.

The WINDOW clause replaces the MATCH_RECOGNIZE clause and specifies that row pattern recognition will be performed in the specified window frame. In Example 9, the window frame is specified with the following line:

```

ROWS BETWEEN CURRENT ROW
AND UNBOUNDED FOLLOWING

```

The PARTITION BY, ORDER BY, PATTERN, SUBSET and DEFINE subclauses in the WINDOW clause have the same semantics as in the MATCH_RECOGNIZE clause. The MEASURES subclause specifies row pattern measures. In contrast to MATCH_RECOGNIZE, the MATCH_NUMBER function is not supported in the WINDOW clause.

The row pattern recognition in the WINDOW clause supports two new options—INITIAL and SEEK. The INITIAL option means that a match is generated only if the pattern starts at the first row of the full pattern match. The alternative is SEEK, meaning that a search for a match starts with the current row, but is permitted through to the end of the full window frame. Either way, if a match is found, the reduced window frame consists of only the pattern match rows, otherwise, the reduced window frame is empty. Only one row pattern match per full window frame is sought.

Finally, the subclauses ONE ROW PER MATCH and ALL ROWS PER MATCH cannot be specified, because they are not applicable with the WINDOW clause.

3 Comments to the Standard’s Specification

There are three topics in the specification of Row Pattern Recognition in the SQL standard, which will be discussed in this section:

- Renaming of the PREV and NEXT functions
- Explicit definition of all pattern variables in DEFINE
- Extending the set of specified functions

3.1 Renaming of PREV and NEXT Functions

Generally, Row Pattern Recognition has a lot of similarities to the window functionality. For instance, the MATCH_RECOGNIZE clause operates on a “window” of rows. Therefore, the semantical meaning of the PARTITION subclause for both concepts is the same.

Row Pattern Recognition can be seen as the next step in the evolution of the window functionality. For this reason, the specification of the window functionality in the SQL standard [2] should be the source for the specification of functions in relation to Row Pattern Recognition. And this is not always the case.

The standardized names of the window functions that are used to access columns of the previous/next row are LAG and LEAD, respectively. Both functions, which have the same semantical meaning in Row Pattern Recognition are called PREV i.e. NEXT. Therefore, to keep the uniformity of different parts of the SQL standard these two functions should be renamed in LAG and LEAD, respectively.

3.2 Explicit Definition of All Row Pattern Variables

In the MATCH_RECOGNIZE clause, the PATTERN subclause is used to specify the row pattern variables. Additionally, the definition of these variables is done within the DEFINE subclause. The SQL standard allows that some of the variables specified in the PATTERN subclause do not appear at all in DEFINE. In other words, the explicit definition of pattern variables with the special meaning can be omitted. Such pattern variables implicitly have the default value TRUE, and therefore match any row.

From our point of view, the decision to allow implicit definition for some row pattern variables is not helpful, especially for programmers, who are accustomed to the phrase “All variables must be declared (defined), before they can be used in a program.”

For this reason, we recommend that each row pattern variable specified in the PATTERN subclause must have an explicit definition in the DEFINE subclause. The new symbol (ALLWAYS_TRUE, for instance) should be assigned to the variables with the property to match any row. The following two code snippets show the difference:

```
-- In accord to the SQL Specification
PATTERN (A B+C+)
DEFINE
B AS B.Price < PREV (B.Price),
C AS C.Price > PREV (C.Price)

-- Our proposition
PATTERN (A B+C+)
DEFINE
A AS ALWAYS_TRUE,
B AS B.Price < PREV (B.Price),
C AS C.Price > PREV (C.Price)
```

3.3 Extending the Set of Functions

Using the MATCH_NUMBER function, the user can assign a sequential number to each resulting match. In other words, each time a match is found, it is assigned a sequential match number, starting from one.

On the other hand, each match has one or more data points, and the user might want to know which data point is associated with each value from the resulting table. The function with such semantics is not specified in the SQL standard.

The following example shows the use of such a function (For instance, the new function can be called MATCH_SEQNO.).

Example 10

```
-- This example does not conform to the Standard
SELECT price_date, matchno, seqno, price, cl
FROM (SELECT * FROM stock_price where company='A')
MATCH_RECOGNIZE (
order by price_date
measures
    match_number() as matchno,
-- match_seqno returns the row # within a match
    match_seqno() as seqno,
    classifier() as cl,
    first(strt.price) as first_price
all rows per match
pattern(strt up+)
define
    /* strt defaults to TRUE, matches any row */
    up as price > prev(price) )
ORDER BY matchno, price_date ;
```

The result would be:

Price_date	Match_no	Seqno	Price	Cl
2020-10-02	1	1	36	STRT
2020-10-03	1	2	39	UP
2020-10-04	1	3	42	UP
2020-10-05	2	1	30	STRT
2020-10-06	2	2	47	UP
2020-10-07	2	3	71	UP
2020-10-08	2	4	80	UP

4 Row Pattern Recognition: Implementations

Until now, the MATCH_RECOGNIZE clause has been implemented in one enterprise RDBMS and several Data Analytics tools. The following two sections discuss its implementation in Oracle, as well as the existing implementations in the Data Analytics tools.

4.1 The Implementation of MATCH_RECOGNIZE in Oracle

Starting with Version 12c, Oracle is the only RDBMS, where the Feature R010, “Row pattern recognition: FROM clause” from the SQL standard has been implemented as a part of the SELECT statement. On the other hand, the Feature R020, “Row pattern recognition: WINDOW clause” has not been implemented at all with this database system [5].

The implementation of the MATCH_RECOGNIZE clause follows exactly the specification of this clause in the SQL standard. In other words, all examples shown in Sect. 2 are valid Oracle queries and can be executed with this database system.

4.2 The Implementation of MATCH_RECOGNIZE in Data Analytics Tools

As far as we know, there are two Cloud Analytics Tools, where the MATCH_RECOGNIZE clause has been implemented: Snowflake and MS Azure Stream Analytics. Snowflake is a computing-based data warehousing company, which offers a cloud-based data storage and analytics services. Azure Stream Analytics is Microsoft’s event processing engine that enables users to develop and run real-time analytics on multiple streams of data.

Note that the implementation of MATCH_RECOGNIZE for Azure Stream Analytics will not be discussed further in this paper, because the implementation is rudimentary and is stream processing oriented. For instance, non-standard-

ized subclause called LIMIT DURATION is implemented. This subclause defines a window of time for the pattern to be searched.

On the other hand, several important subclauses are not implemented at all. ORDER BY as a part of PARTITION BY subclause is not implemented as well as the SUBSET and PERMUTE subclauses. The detailed description of this implementation can be found in [6].

4.2.1 Snowflake

Snowflake supports Feature R010, “Row pattern recognition: FROM clause” as well as Feature R020, “Row pattern recognition: WINDOW clause”, but with different outcome. The syntax of the former is almost identical to the specification of RPR in the SQL standard, while the syntax of the latter is totally different to the standard (The detailed description of this implementation can be found in [7].).

Snowflake uses the MATCH_RECOGNIZE clause inside the FROM clause of SELECT in the same way as it is specified in the standard. Additionally, all standardized subclauses are implemented with the same semantics and with the same syntax. Several minor syntax differences to the SQL standard will not be discussed here, because of their insignificance.

The concept and syntax of Snowflake’s implementation of Row Pattern Recognition inside a window frame is significantly different from the SQL standard, as the following example shows (The semantics of the example below is similar to the semantics of Example 9.).

Example 11

```
SELECT * FROM stock_price
  match_recognize(
    partition by company
    order by price_date
    measures
      match_number() as match_number
    all rows per match omit empty matches
    pattern(overavg*)
    define
      overavg as price > avg(price)
      over (rows between unbounded
        preceding and unbounded following) )
ORDER BY company, price_date;
```

The Snowflake’s concept of Row Pattern Recognition inside a window frame handles this feature in the same way as for tables’ rows. In other words, the MATCH_RECOGNIZE clause is used for pattern recognition in tables’ rows as well as inside a window frame. In contrast to it, the SQL standard makes the strict distinction between these two

different features using the MATCH_RECOGNIZE clause exclusively for pattern recognition inside tables’ rows and the WINDOW clause for the same process inside a window frame.

5 Summary

The standardized specification of Row Pattern Recognition in SQL has two main parts, one which describes the functionality of RPR in relation to tables’ rows and one in relation to window frames. The former part represents the extensive and successful specification of all necessary functions and subclauses, which are necessary to search for arbitrary patterns inside rows of tables.

The specification in relation to RPR inside window frames is also well done. The decision to specify RPR functions and subclauses inside the WINDOW clause, which is already specified in the standard, allows the seamless transition from window frames to the process of searching inside them.

In spite of it, we recommend several modifications as well extensions to the existing specification. These features concern the following concepts:

- Renaming of pattern navigation operations PREV and NEXT in LAG and LEAD, respectively.
- The definition of *all* row pattern variables should be explicit.
- The set of existing functions should be extended with at least one new function, which is related to the existing MATCH_NUMBER function.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- ISO/IEC TR 19075-5:2016 Information technology—Database languages—SQL Technical Reports—Part 5: Row Pattern Recognition in SQ. <https://standards.iteh.ai/catalog/standards/iso/2521dd59-7047-4938-a25a-2617fde630df/iso-iec-tr-19075-5-2016>. Accessed 8 Aug 2021

2. ISO/IEC TR 19075-5:2016 Information technology—Database languages—SQL Technical Reports—Part 2. <https://www.iso.org/standard/63556.html>. Accessed 15 Aug 2021
3. Michels J et al (2018) The New and Improved SQL:2016 Standard. Sigmod Rec. <https://doi.org/10.1145/3299887.3299897>
4. Friedl J (2006) Mastering regular expressions
5. Oracle 12c MATCH_RECOGNIZE. https://docs.oracle.com/cd/E28280_01/apirefs.1111/e12048/pattern_recog.htm#CQLLR1531. Accessed 9 Jan 2021
6. MS Stream Analytics MATCH_RECOGNIZE. <https://docs.microsoft.com/en-us/stream-analytics-query/match-recognize-stream-analytics>. Accessed 30 July 2021
7. Snowflake MATCH_RECOGNIZE. https://docs.snowflake.com/en/sql-reference/constructs/match_recognize.html. Accessed 9 Sept 2021