Pattern Matching in SQL – analytical power at your fingertips

Hermann Bär
Partitioning Product Management
Oracle USA
Redwood Shores, CA USA

Schlüsselworte
Patttern recognition, SQL Pattern Matching, analytical SQL

Introduction

Patterns are usually defined as a repetitive series or sequence of specific events or actions and they occur everywhere in business. The ability to find, analyze and quantify individual or groups of patterns within a data set is now a key business requirement.
Oracle Database 12c adds native pattern matching capabilities to SQL. This brings the simplicity and efficiency of the most common data analysis language to the process of identifying patterns within a data set. It offers significant gains in term of performance, maintainability and scalability compared to the continued use of bespoke code embedded within applications and executed either on the client side or within the middletier application server.

Native Pattern Matching in SQL

Oracle Database 12c provides a completely new native SQL syntax that has adopted the regular expression capabilities of Perl by implementing a core set of rules to define patterns in sequences (streams of rows). This new inter-row pattern search capability complements the already existing capabilities of regular expressions that match patterns within character strings of a single record.

The 12c MATCH_RECOGNIZE SQL construct allows the definition of patterns, in terms of characters or sets of characters, and provides the ability to search for those patterns across row boundaries. The new clause MATCH_RECOGNIZE offers native declarative pattern matching capabilities within SQL.

Below is a simple example of the syntax used to identify a typical stock price pattern where the price declines and then rises (a down-and-up or ‘V’ pattern):

```
SELECT * FROM stock_ticker
MATCH_RECOGNIZE (
  PARTITION BY symbol ORDER BY timestamp
  MEASURES  STRT.timestamp AS start_timestamp,
            FINAL LAST(DOWN.timestamp) AS bottom_timestamp,
            FINAL LAST(UP.timestamp) AS end_timestamp
  ALL ROWS PER MATCH
  AFTER MATCH SKIP TO LAST UP
  PATTERN(STRT DOWN+ UP+)
  DEFINE DOWN AS DOWN.price < PREV(DOWN.price),
         UP AS UP.price > PREV(UP.price)
)MR
ORDER BY MR.symbol, MR.timestamp;
```

In this example, the data source for out pattern matching process is a table called stock_ticker, which contain market trading data.

The MATCH_RECOGNIZE clause aims to logically partition and order the data stream (in our example the table stock_ticker) that you wish to analyze. As can be seen from the above code extract, the syntax for this new clause is very rich and comprehensive yet it is easy to understand.

The PATTERN clause of the MATCH_RECOGNIZE construct defines the patterns that you are seeking to identify within your stream of records and uses regular expression syntax.
Each pattern variable is then described in the DEFINE clause using SQL-like syntax to identify individual rows or inter-row changes in behavior (events). The structure used to define the patterns will be well known to developers who are familiar with regular expression declarative languages such as PERL.

There four basic steps for building a MATCH_RECONGIZE clause:

1. Define the partitions/buckets and ordering needed to identify the 'stream of events' you are analyzing
2. Define the pattern of events and pattern variables identifying the individual events within the pattern
3. Define measures: source data points, pattern data points and aggregates related to a pattern
4. Determine how the output will be generated

The following sections provide an overview of the most important keywords within the MATCH_RECOGNIZE clause within these four simple steps. For a more detailed view of this feature please refer to the Oracle Database 12c SQL documentation.

Define the partitions (buckets of data) and the ordering within each of these streams
The bucketing and ordering of data are controlled by the keywords "PARTITION BY" and "ORDER BY".

PARTITION BY: This clause divides the data into logical groups so that you can search within each group for the required pattern. "PARTITION BY" is an optional clause and is typically followed by the "ORDER BY" clause.

ORDER BY: The order of the data is very important as this controls the "visibility" of the pattern we are searching for within the data set. The MATCH_RECOGNIZE feature uses the ORDER BY clause to organize the data so that it is possible to test across row boundaries for the existence of a sequence of rows within the overall "stream of events".

```
PARTITION BY symbol
ORDER BY timestamp
```

In the case of our example we want to separate the data in the table stock_ticker in to an individual stream of records for each stock ticker symbol. This will allow us to search for the down-up V-pattern within each stock ticker symbol. Our example then orders the data based on the timestamp information within each stock ticker symbol grouping. This is probably the most common way to order a dataset since most patterns are in some way time dependent.

Define the pattern
The PATTERN clause makes use of regular expressions to define the criteria for a specific pattern (or patterns). The subsequent syntax clause DEFINE is used to define the associated pattern variables.
Uou define three basic elements:

1. The pattern variables that must be matched
2. The sequence in which they must be matched
3. The frequency of patterns that must be matched

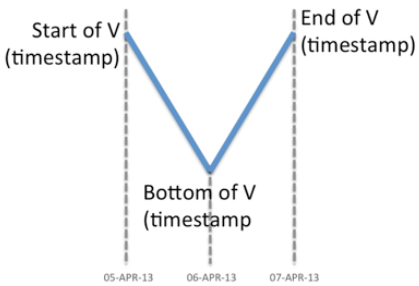
The PATTERN clause depends on pattern variables which means you must have a clause to define these variables and this is done using the DEFINE clause. It is a required clause and is used to specify the conditions that a row must meet to be mapped to a specific pattern variable.

```
PATTERN(STRT DOWN+ UP+)
DEFINE
DOWN AS DOWN.price < PREV(DOWN.price),
UP AS UP.price > PREV(UP.price)
```

Our example defines a pattern in a stream of events where we want to see one or multiple ‘DOWN’ events ‘DOWN’ followed by one or multiple ‘UP’ events, beginning from a start event ‘STRT’.
The ‘DOWN’ event as a row where the current price is less than the price of its previous row and the ‘UP’ event as a row where the current price is higher than the price of its previous row. As you see, we do not have a definition for the event ‘STRT’ used in our pattern clause. A pattern variable does not require a definition: any row can be mapped to an undefined pattern variable, creating an always-true event.

Define the measures
This section allows you to define the output measures that are derived from the evaluation of an expression or calculated by evaluating an expression related to a particular match.
Measures can be individual data points within a pattern such as start or end values, aggregations of the event row set such as average or count, or SQL pattern matching specific values such as a pattern variable identifier.

```
MEASURES
STRT.timestamp AS start_timestamp,
FINAL LAST(DOWN.timestamp) AS bottom_timestamp,
FINAL LAST(UP.timestamp) AS end_timestamp
```



In our example we want to identify V-shape patterns within a stock price over time. To identify this shape we want to track the following measures as characteristics of this pattern:
- The start date of the V shape (start_timestamp)
- The bottom date of the V shape (bottom_timestamp)
- The last day of the V shape (end_timestamp)

Determine how the output will be generated
Note that it is important to look at the measures in the context of how to retrieve the rows that match the specified pattern since there could be multiple occurrences of an event within a pattern.

For example, in the pattern above there could be multiple DOWN events before the bottom of the V is reached. Therefore, the developer needs to determine which event they want to include in the output – the first, last, minimum or maximum value. It is important to be very precise when defining measures to ensure the correct, or expected, element is returned

When patterns are identified in streams of rows you sometimes want the ability to report summary information (for each pattern) in other cases you will need to report the detailed information. The ability to generate summaries as part of this process is very important. The "PER MATCH" clause can create either summary data or detailed data about the matches and these two types of output are controlled by the following key phrases:

- ONE ROW PER MATCH - each match produces one summary row and this is the default.
- ALL ROWS PER MATCH - a match spanning multiple rows will produce one output row for each row in the match.

Below is an example of the syntax with the additional statement to control where the process should restart once a match has been established:

```
ALL ROWS PER MATCH
AFTER MATCH SKIP TO LAST UP
```

In the above example we want to report all records that fall into the V shape of a stock ticker. The previously defined measures will be calculated for each matching row and reported in addition to the column values of table stock ticker. Once the type of output has been determined the next issue is to determine the flow of control once a complete pattern has been matched.

The AFTER MATCH SKIP clause determines the point at which we can resume the row pattern matching process once we have established a match. The options for controlling this are as follows:

- AFTER MATCH SKIP TO NEXT ROW|PAST LAST ROWResume pattern matching at the row after the first/last row of the current match.
- AFTER MATCH SKIP TO FIRST| LAST <pattern_variable>: Resume pattern matching at the first/last row that is mapped to the pattern variable.
- AFTER MATCH SKIP TO <pattern_variable> - Resume pattern matching at the last row that is mapped to the pattern variable.

Using this simple four-step approach you are capable of solving business problems in a way that it was nver before in SQL.

Conclusion

The release of the new 12c pattern-matching feature provides significant benefits to both SQL developers and business users. What SQL developers are looking for is a way to combine the pattern processing flexibility offered by languages such as Perl and Java with the declarative and analytical power of SQL.

This approach offers significant gains in term of performance, maintainability and scalability compared to the continued use of bespoke code embedded within the application and executed either on the client side or within the middle-tier application server. Moving forward developers should switch their pattern matching logic to the new SQL-based pattern matching features offered by Oracle Database 12c so they can benefit from simplified code, more efficient processing and improved performance. For business users and data scientists the key advantage of in-database pattern matching is that it allows them to incorporate pattern enrichment into their existing analytical workflows without having to resort to complex externally coded routines that increases network traffic and increase data latency.

Overall, the new in-database pattern matching reduces data complexity, data movement and latency while at the same time increasing data security and providing significant performance improvements. Using SQL pattern-matching customers can further maximize their investment in Oracle Database technology.

Kontaktadresse:

Hermann Bär
Oracle USA
400 Oracle Parkway
Redwood Shores, CA 94065, USA

Telefon:	+1 (0) 650.506.6833
Fax:	+1 (0) 650.506.6833
E-Mail	hermann.baer@oracle.com
Internet:	www.oracle.com