An Oracle White Paper
June 2013

Patterns everywhere – Find them fast!
SQL pattern matching in Oracle Database 12*c*

ORACLE

Executive Overview

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. It can help you gain a better understanding of your customers' behavior and associated operational activities, seek out new opportunities to drive additional revenue streams and even help identify malicious activities that could lead to your business incurring significant costs.

Up until today, the most commonly used language for database systems – SQL – did not provide comprehensive support to quickly and easily define and track patterns. This has resulted in developers creating bespoke, complex code that is hard to understand and time consuming to optimize and maintain. It is not easy to share these developer-centric routines with other business users, which means they cannot incorporate pattern matching into their own data discovery workflows. This constrains the overall value of existing approaches to pattern matching to your business.

Oracle Database 12*c* 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 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.

This whitepaper will explore this new SQL functionality and explain how different industries can leverage the full richness of Oracle's new SQL pattern matching capabilities.

Introduction to pattern matching

Patterns everywhere

The world is full of data. The volume of data being generated, captured and moved in to data repositories by businesses is growing rapidly along with the rate of capture. The growing interest in new areas of technology related to big data is driving the creation and capture of even more data. Sensors are being fitted to more and more devices to measure and record many different types of activities and interactions and business need simple and efficient ways to search for patterns within these new data sets. Within these vast oceans of data there are all sorts of patterns waiting to be discovered.

How to Discover Patterns

Pattern recognition techniques are used in a number of ways by many businesses. Typical examples include processing financial time series data sets to look for market opportunities, analysis of network logs to look for specific call patterns that can be used to measure service quality, fraud detection where a series of small transactions followed by a high-value transaction indicate an attempt to acquire goods or services by deception and analysis of firewall and security logs to search for and detect intruders.

The Oracle Database already supports the discovery of simple patterns through the use of features such as the LIKE condition and regular expressions. Using these features developers and business users have moved some of their pattern matching processing back inside the Oracle Database to benefit from increased performance, scalability and security.

For more sophisticated pattern matching requirements application and database developers have been forced to implement additional pattern-matching technologies, such as specialized software, or they added bespoke code to their applications using a variety of languages - either directly within those applications or within the middle-tier platform. In general terms this has resulted in a lot of data movement – pulling data out of the database, executing the pattern matching code and then pushing the data back into the same database and/or other target applications/databases.

In many cases this approach of moving data in and out of the database creates a series of problems in terms of increased complexity of performance optimizations, increased data latency and reduced data security. In a similar manner to that dictated by advanced analytical processing, such as data mining, the ideal state is to bring the analysis to the data, not the other way round. For pattern matching what developers and business users need is a rich expressive SQL feature that they can embed within their existing application processing and BI reports to processes the data in-place, i.e. inside the Oracle Database.

SQL pattern matching with Oracle Database 12*c*

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 feature 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 which is available via the OTN website[1].

Overview of keywords used in the pattern matching

This section provides an overview of the most important keywords that make up the MATCH_RECOGNIZE clause and the keywords are organized according to the four steps outlined above.

Step 1: Bucket and order the data

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

```
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.

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
```

[1] Link to Database documentation library: http://www.oracle.com/technetwork/indexes/documentation/index.html

Our example 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.

Step 2: 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.

PATTERN

In this section you 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

For example:

```
PATTERN(STRT DOWN+ UP+)
```

This 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'.

DEFINE

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.

For example:

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

This example defines 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. In our example this ensures that every record is considered as possible event to be test against for the specified pattern. The inclusion of a "STRT" event can be useful within the MEASURE clause and this is covered in the next section.

Step 3: 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

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. Here is a simple example of how data points within a pattern can be defined:

```
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)

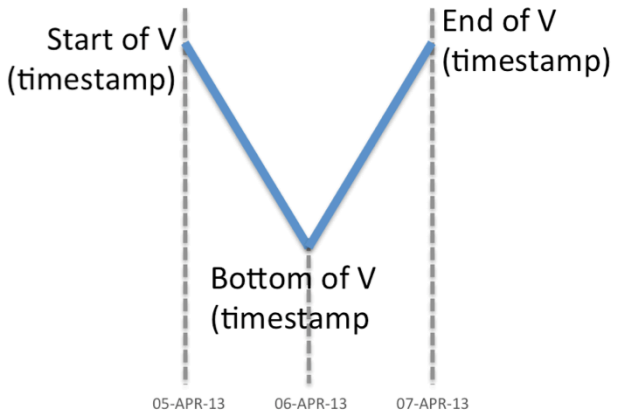

Figure 1 – V-Shape pattern typically found in stock-ticker data sets

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.

Step 4: Controlling the output

[ALL ROWS | ONE ROW] PER MATCH

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.

AFTER MATCH SKIP

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 ROW- Resume 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.

The default for this clause is AFTER MATCH SKIP PAST LAST ROW.

Additional features

The MATCH_RECOGNIZE clause is able to search for very sophisticated patterns. Creating and testing these patterns is made significantly easier by two key words that can be included in the output:

- MATCH_NUMBER
- CLASSIFIER

These two measures can be included queries to filter the output and provide BI users with more control over how the data is presented in dashboards and reports. Developers are likely to find all sorts of uses for this type of information. In some cases, where the results of MATCH_RECOGNIZE clause have to be audited it is a good idea to include these two measures as part of the output as the information they provide is very important.

MATCH_NUMBER

Most patterns are likely to generate be a large number of matches within a given partition. The **MATCH_ NUMBER** returns sequential number, starting with 1, for each match of a pattern in the order that they are found. This makes it easier to analyze how your pattern criteria are being applied.

CLASSIFIER

The **CLASSIFIER** returns the name of the variable within the overall pattern that applies to a specific row. This is the pattern variable that the row is mapped to by a row pattern match. As with the MATCH_NUMBER feature this can be used during testing to help developers determine if their pattern has been correctly defined and is returning the expected results.

The code below shoes the inclusion of these two measures:

```
MEASURES
      STRT.tstamp AS start_tstamp,
      FINAL LAST(DOWN.tstamp) AS bottom_tstamp,
      FINAL LAST(UP.tstamp) AS end_tstamp,
      MATCH_NUMBER() AS match_num,
      CLASSIFIER() AS var_match
```

The output from the above statement would be as follows:

```
SYMBOL    START_TST BOTTOM_TS END_TSTAM MATCH_NUM  VAR_MATCH
--------- --------- --------- --------- ---------- --------------
ACME       05-APR-11 06-APR-11 10-APR-11          1 STRT
ACME       05-APR-11 06-APR-11 10-APR-11          1 DOWN
ACME       05-APR-11 06-APR-11 10-APR-11          1 UP
ACME       05-APR-11 06-APR-11 10-APR-11          1 UP
ACME       05-APR-11 06-APR-11 10-APR-11          1 UP
ACME       05-APR-11 06-APR-11 10-APR-11          1 UP
ACME       05-APR-11 06-APR-11 10-APR-11          1 UP
ACME       10-APR-11 12-APR-11 13-APR-11          2 STRT
ACME       10-APR-11 12-APR-11 13-APR-11          2 DOWN
ACME       10-APR-11 12-APR-11 13-APR-11          2 DOWN
ACME       10-APR-11 12-APR-11 13-APR-11          2 UP
```

Adding aggregations within patterns

Creating aggregations of key elements within the pattern matching workflow is a normal requirement. The use of SQL to define pattern-matching logic allows us to simply extend the way data is manipulated during the pattern matching process using standard SQL operators. It is now possible to

incorporate SQL aggregates, such as **COUNT, SUM, AVG, MAX**, and **MIN**, within both the **MEASURES** and **DEFINE** syntax to create richer result sets. This is especially useful for developers as it allows them to greatly simplify their application code. Business users can leverage this feature to quickly and efficiently summarize patterns or groups of patterns for easier analysis.

When used in row pattern matching, aggregates operate on a set of rows that are mapped to a particular pattern variable and this allows the creation of running and/or final aggregations within the result set. Below we have extended the original stock_ticker example to use the **COUNT**() and **AVG**() aggregate features:

```
SELECT * FROM stock_ticker
MATCH_RECOGNIZE (
 PARTITION BY symbol
 ORDER BY timestamp
 MEASURES
      COUNT(*) as total_no_of days,
      COUNT(DOWN.*) AS down_days,
      COUNT(UP.*) AS up_days,
      AVG(DOWN.price) AS avg_down_price,
      AVG(UP.price) AS avg_up_price
 ONE ROW PER MATCHAFTER 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;
```

The syntax for **COUNT(*)** has been extended to provide a richer syntax for pattern matching. It is now possible to count individual pattern so that **COUNT(**DOWN.***)** will count the number of rows that are mapped to the pattern variable DOWN.

Additional control of aggregate processing is provided by two important key words: **RUNNING** and **FINAL**. These two keywords are used to indicate whether running or final aggregates should be returned within the result set. They can be used with both aggregates and the standard row pattern navigation operations, such as FIRST and LAST.

Summary

The general syntax for SQL pattern matching is very rich which means it is easy to define complex patterns. The above examples only provide an overview of the most basic elements of this

functionality; more information is available in the Data Warehousing Guide and the SQL Language Reference Guide for Oracle Database 12*c* which is available via the OTN website[2].

SQL pattern matching use cases

In this section we will review a series of uses cases and provide conceptual simplified SQL to solve these business requirements using the new SQL pattern matching functionality. These examples are taken from real-life requirements from customers in the financial services, retail, life sciences and telecommunications industries, although the scenarios are also applicable across many other industries.

Stock market analysis

The stock market is driven by the need to seek out and understand patterns. Managers within trading departments usually have a set of predefined scenarios, or patterns, that should trigger alerts within their dashboards. Extracting typical trade price patterns such as U-shapes, V-shapes, Inverted V-shapes W-shapes and Elliot Waves, as shown in figure 2, are easy to achieve using the rich pattern definition syntax.

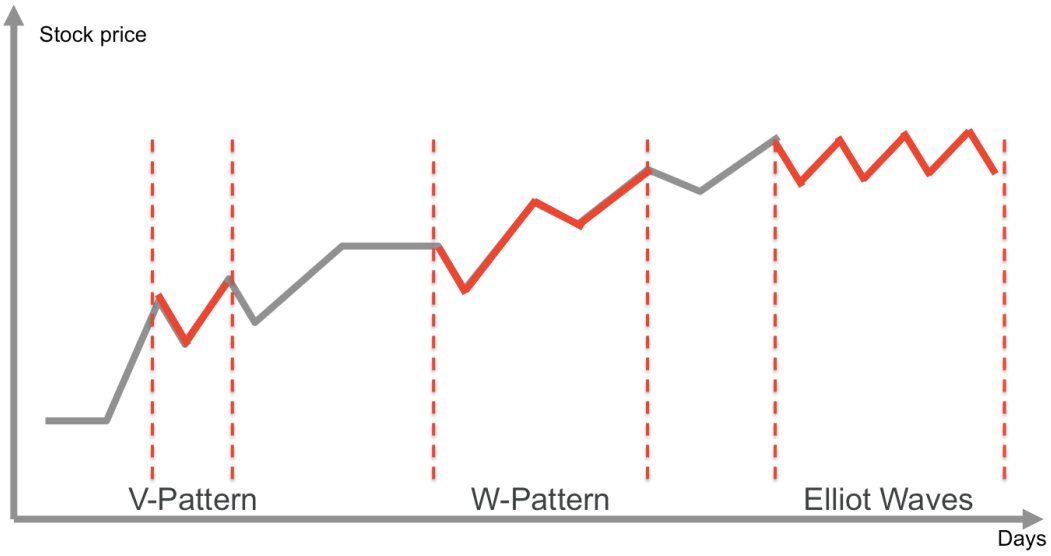

Figure 2 – Typical patterns in financial markets

[2] Link to Database documentation library: http://www.oracle.com/technetwork/indexes/documentation/index.html

The richness of the MATCH_RECOGNIZE syntax allows developers to create quite complex patterns such as searching for a stock with a price drop of more than 8% and then where, within zero or more additional days, the stock price remains below the original price. The syntax for the pattern described above would be as follows:

```
... MATCH_RECOGNIZE (
      PARTITION BY symbol
      ORDER BY timestamp
    MEASURES
      A.tstamp as start_timestamp,
      A.price as start_price,
      B.price as drop_price,
      COUNT(C.*)+1 as cnt_days,
      D.tstamp as end_timestamp,
      D.price as end_price
   ONE ROW PER MATCH
   AFTER MATCH SKIP PAST LAST ROW
   PATTERN (A B C* D)
   DEFINE
      B as (B.price - A.price)/A.price < -0.08,
      C as C.price < A.price,
D as D.price >= A.price)
```

The measure `CNT_DAYS` calculates how long the price remains below the initial price before the 8%+ drop and `END_PRICE` returns the price it reaches when it comes out of this dip again.

The pattern syntax even supports much more complex patterns such as Elliot Waves which have multiple consecutive patterns of inverted V-shapes. The pattern expression has to search for one or more days where the prices goes up, followed by one or more days where the price goes down. The sequence must appear consecutively, for a specified period of time, with no gaps.

Whatever stock price development pattern you are after, the result can be fed into further deeper analysis using other analytical functionality such as data mining and multi-dimensional analysis to further slice and dice the data set. By moving the pattern matching process inside the database it much easier to create this sort of joined-up analysis since there is no need to move data and this reduces latency, which means real-time analysis becomes possible.

Ensuring governance and compliance

Auditors and compliance teams can use SQL pattern matching to help then search through large amounts of trading data looking for stocks that show heavy trading – a large number of transactions in a concentrated period – and/or unusual activity. For example, heavy trading could be defined as three transactions occurring in a single hour where each transaction was for more than 30,000 shares.

```
SELECT *
```

```
FROM stock_trades
MATCH_RECOGNIZE (
   PARTITION BY symbol
   ORDER BY tstamp
   MEASURES
         FIRST (A.tstamp) AS in_hour_of_trade,
         SUM (A.volume) AS sum_of_large_volumes
   ONE ROW PER MATCH
   AFTER MATCH SKIP PAST LAST ROW
   PATTERN (A B* A B* A)
   DEFINE
      A AS ((A.volume > 30000) AND
     ((A.tstamp - FIRST (A.tstamp)) < '001:00:00.00' )),
     B AS ((B.volume <= 30000)
     AND ((B.tstamp - FIRST (A.tstamp)) < '01:00:00.00'))
   );
```

Compliance teams use application logs to look for patterns that indicate unauthorized changes to customers account. For example a tax clerk might make a series of small changes to a person's tax record before making a major change such as an unauthorized manual adjustment with the objective of reducing a tax bill.

SQL based pattern matching is ideally suited to compliance type requirements because it is much easier to make and manage the changes. These types of patterns tend to evolve over time as the perpetrators try to stay ahead of the compliance teams. The ability to quickly and easily modify a given search pattern to look for slightly different events, or even new events.

Monitoring online customer behavior

One of the most popular uses cases for pattern matching is to analyze user activity on a website. Pattern matching is used to identify each session within a series of clicks and then track user activity that typically involves multiple events, such as: click activity, moving from page A to page B, responding to recommendations etc. The data from this analysis can be pushed to a report such as the

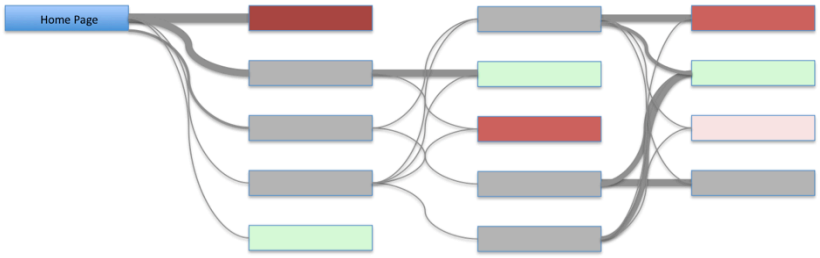

Figure 3: real-time results from A-B testing for website usage

one shown here in figure 3. This highlights the flow of traffic through a website. The overall objective is to understand why some paths end in failure (no purchase or no click through when a promotion is shown) and why some paths lead to positive results (a purchase or a click through on a promotion).

Assigning session numbers at the detail-level is just the start of the analytical process. The real business value in using pattern matching for this type of analysis comes after aggregating the data by session to give a higher-level picture. Typical aggregations include summarizing the events per session along with the total session duration.

The first step in this analysis is to define a session - a sequence of one or more time-ordered rows with the same partition key where the time gap between timestamps is less than a specified threshold. The example below sets the time threshold for a session as 10 seconds.

```
SELECT time_stamp, user_id, session_id
FROM Events
MATCH_RECOGNIZE
  (PARTITION BY User_ID
   ORDER BY Time_Stamp
   MEASURES
       MATCH_NUMBER() AS session_id,
       COUNT(*) AS no_of_events,
       FIRST(time_stamp) AS start_time, LAST(time_stamp) -
       FIRST(time_stamp) AS duration
   PATTERN (b s*)
   DEFINE
       s AS (s.Time_Stamp - PREV(Time_Stamp) <= 10)
   )
   ORDER BY user_id, session_id;
```

If rows have a time difference between the timestamps that is greater then the specified threshold then they are considered to be in different sessions. After identifying a session it is possible to do further analysis such as quantifying:

- How many pages each users visits during a typical session
- How long each user spends on each page before moving to the next page

Organizations can gain significant benefit from understanding the behavior of their online customers

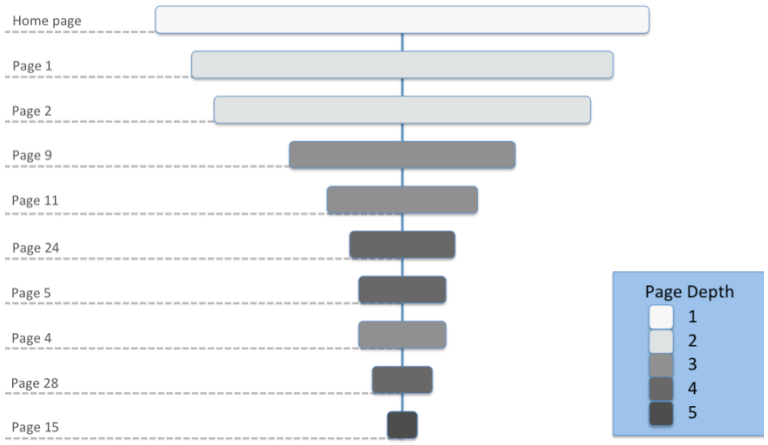

Figure 4 – Web site analytics: page depth of visitor navigation

because it can help them define service offerings, enhancements, pricing, marketing campaigns and more.

Using data visualization, as shown in figure 4, and analysis tools such as those used to support Oracle Enterprise R, business users can use this new pattern-matching feature to quickly and easily determine how many clicks each page receives and total length of stay for each page. These results can then be used them to drive many additional layers of analysis such as maximum, minimum, and average time spent on each page.

Tracking call service quality

Telecommunications companies run service quality analysis on their network logs to help them understand the characteristics of phone sessions where the sessions involve dropped connections and users redialing.

This analysis of network logs is a little more complex compared to other use cases because even if there are clear markers to indicate that a call finished, those markers may not indicate that the user actually intended to end the call. Consider the scenario where a person is using their mobile phone when the phone connection is dropped: typically, the user will redial and continue the phone call. In that scenario, multiple phone calls involving the same pair of phone numbers need to be considered part of a single phone session.

The following code snippet finds the sessions where calls between two people are grouped into a session if the gap between subsequent calls is within a threshold of 60 seconds. That threshold is specified in the DEFINE clause. The MEASURES clause returns information about each call such as:

- How many times calls were restarted in a session
- Total effective call duration
- Total interrupted duration

```
SELECT Caller,
       Callee,
       Start_Time,
       Effective_Call_Duration,
       (End_Time - Start_Time) - Effective_Call_Duration
                  AS Total_Interruption_Duration,
       No_Of_Restarts,
       Session_ID
FROM my_cdr
MATCH_RECOGNIZE
      PARTITION BY Caller, Callee
      ORDER BY Start_Time
      MEASURES
            A.Start_Time AS callstart_time,
            End_Time AS call_end_time,
            SUM(End_Time - Start_Time) AS call_duration,
            COUNT(B.*) AS no_of_dropped_calls
            MATCH_NUMBER() AS pm_reference
```

```
        PATTERN (A B*)
        DEFINE
              B AS B.Start_Time - PREV(B.end_Time) < 60
        );
```

This is an important metric because it is often a good indicator of service quality. Customers are more likely to terminate or not renew their contract if they have poor in-call experiences and this type of churn drives up costs.

Tracking money laundering

Many companies are looking for simple and more efficient ways to identify and track fraud. Industries such as financial services and telecoms companies are finding their products and services are being used to launder money. The ability to search for suspicious financial patterns is becoming a primary requirement and Oracle's SQL pattern matching is the perfect in-database SQL method for doing this as it offers fast and efficient processing.

The richness of this SQL feature allows developers to define "unusual" and "suspect" patterns of behavior that are unique to their industry or particular circumstances. For example you might need to search for a behavior that seems suspicious when there is a transfer of funds. A typical "suspicious" transfer pattern could be defined as three or more small money transfers (less than $2000) within 30 days followed by a large transfer (over $1,000,000) within 10 days of the last small transfer.

Using the pattern clause it is relatively easy to define a series of statements that firstly identify the small transfers, flag the large transfer that follows, identify if the small transfers occurred within a specified time period (in this case, 30 days) and lastly catch large transfers that then occur occurred within a specified time window of the last small transfer, in this case that window is 10 days.

In simple terms the syntax for this type of pattern would be similar to the following:

```
SELECT userid,
       first_t,
       last_t,
       amount
FROM (SELECT * FROM event_log WHERE event = 'transfer')
MATCH_RECOGNIZE
      (PARTITION BY userid
      ORDER BY time
      MEASURES
            FIRST(x.time) first_t,
            y.time last_t,
            y.amount amount
      PATTERN ( x{3,} y )
      DEFINE
            x AS (event='transfer' AND amount < 2000),
            y AS (event='transfer' AND amount >= 1000000
            AND LAST(x.time) - FIRST(x.time) < 30
            AND y.time - LAST(x.time) < 10)
      );
```

Using this new in-database SQL based pattern matching feature customers have the opportunity to leverage the simplicity, performance and scalability of the Oracle Database to deliver real-time fraud detection processing to support anti-money laundering requirements.

Intruder detection

With hacking and unauthorized access to computer systems on the increase the need for a robust, scalable and high performance pattern matching process has never been greater. Most computer systems are configured to generate error messages and authentication checks that can be analyzed to determine if there are security issues and/or other related problems.

In many cases the log combing process is looking for a given number of consecutive authentication failures ignoring other attributes such as the IP origination address. Using pattern matching it is an easy task to define a search for this type of pattern. For example the code might look something like this:

```
. . .
MATCH_RECOGNIZE(
PARTITION BY error_type ORDER BY time_stamp
MEASURES
      S.time_stamp AS START_T
      W.time_stamp AS END_T
      W.ipaddr AS IPADDR
      COUNT(*) AS CNT
      MATCH_NUMBER() AS MNO
ONE ROW PER MATCH
AFTER MATCH SKIP TO LAST W
PATTERN ( S W{2,} )
DEFINE
      S AS S.message LIKE '%authenticat%',
      W AS W.message = PREV (W.message)
      AND W.ipaddr = PREV (W.ipaddr)
) MR_S3
. . .
```

This simple example can be expanded to search the logs for four or more consecutive authentication failures, regardless of IP origination address. With these types of result sets it would be beneficial to layer additional analytical features on top to provide further levels of insight. By using an IP identification service a spatial map could used to display the originating location of each incident. This would allow security teams to look for other "patterns" within the results.

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. The new MATCH_RECOGNIZE syntax allows them to simplify their application code by leveraging SQL-based in-database processing that can be integrated with existing database developer features such as: PL/SQL table functionality to stream result sets in real-time where sophisticated processing is required and the Oracle Data Cartridge Interface where Java and/or C++ is the preferred development language.

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.

BI users can easily layer their analysis by incorporating pattern matching into their initial query, which can then be sliced, diced and pivoted just like any other result-set. Once the initial pattern has been discovered other layers of analysis such as multi-dimensional, spatial, data mining can then be applied over that initial result set to gain deeper insight.

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.

Patterns everywhere – Find them fast!
SQL pattern matching in Oracle Database 12c

Oracle is committed to developing practices and products that help protect the environment

June 2013
Author: Keith Laker
Senior Principal Product Manager, Data Warehousing and Big Data

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com