Meet your Match: Advanced Row Pattern Matching

Stew Ashton
Independent
Paris, France

Keywords:

Oracle Database 12c, SQL, row pattern matching, MATCH_RECOGNIZE

Introduction

The new Row Pattern Matching functionality in Oracle Database 12c enables SQL developers to quickly solve new problems - and old problems in much more efficient and scalable ways.

This session will summarize the MATCH_RECOGNIZE clause, then present general-purpose SQL techniques for bin fitting problems, hierarchical summaries, temporal validity queries and joining or summarizing data by ranges. By decreasing CPU use, these techniques run 50 to 500 times faster than existing solutions and complement the advantages of Exadata or the In-Memory option.

This session is geared primarily to SQL practitioners who are willing to unleash the full power of SQL in the Oracle database. They will learn how to apply row pattern matching concepts to different categories of problems, and will observe these concepts applied to typical situations using repeatable techniques.

Some prior acquaintance with the MATCH_RECOGNIZE clause is desirable.

Reminder of the Basics

To start with, we will explain what "row pattern matching" means and how MATCH_RECOGNIZE defines row patterns. Then we will go over the steps taken to match input rows with the pattern.

A PATTERN is an uninterrupted series of input rows described as list of conditions. The syntax of the conditions resembles classic "regular expressions".

PATTERN (A B*)

Means we want one row that meets the A condition and 0 or more rows that meet the B condition. If there is a choice, we want as many B rows as possible.

We then DEFINE at least one of the row conditions. Any undefined condition is by default TRUE, so any row will meet that condition.

```
DEFINE
[A undefined = TRUE]
B AS page = PREV(page)+1
```

This means we want a row, followed by any rows with consecutive page numbers.

Each series that matches the pattern is a “match”. Within the match, "A" and "B" are now row subsets and identify the rows that meet their conditions.

Note that there cannot be “unmatched” rows within the pattern, but there can be unmatched rows between series

The processing steps can be outlined as follows:

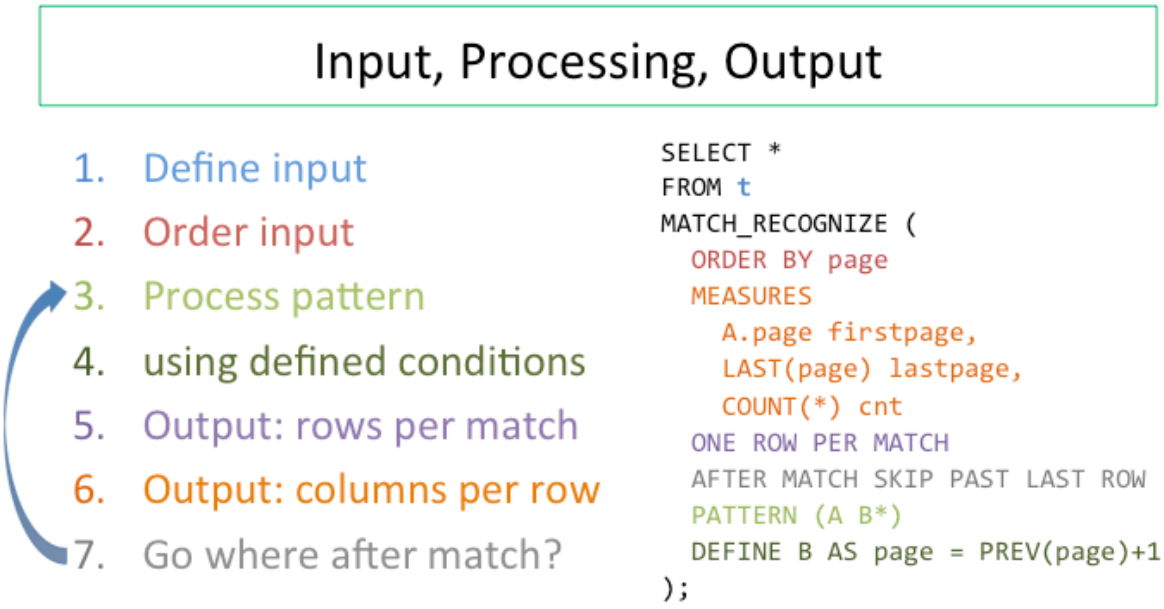

Illustration. 1: MATCH_RECOGNIZE processing logic

Bin fitting and bidirectional sequencing

Using a customer transaction table as an example, I will demonstrate “agile” implementation of a complex requirement. The table contains customer ids, transaction dates and descriptions. Basically, each description indicates whether the transaction was a sale or an inquiry.

First, I will show the simplest possible MATCH_RECOGNIZE clause that will simply return the input dataset.

CUST_ID	TX_DATE	DESCR
C001	2016-01-01	Inquiry
C001	2016-01-01	Inquiry
C001	2016-01-10	Sales
C001	2016-01-21	Repeat Inquiry
C001	2016-02-10	Repeat Inquiry
C001	2016-05-01	Sales
C001	2016-05-06	Sales
C001	2016-06-10	Inquiry 1
C001	2016-09-01	Inquiry 2
C002	2016-02-01	Inquiry 1
C002	2016-02-25	Inquiry 2
C003	2016-02-01	Inquiry 2
C003	2016-02-10	Sales
C003	2016-02-10	Sales
C003	2016-03-10	Inquiry 2
C004	2016-04-15	Sales

```
select * from t match_recognize(
 all rows per match
 pattern (a*)
 define a as 1=1
);
```

Second, I will add a "bin fitting" requirement: number each transaction in sequence, but start over after 40 days. Note that the second 40 day window starts not at day 41, but at whatever date follows the last row of the first window. In this case, the second window starts on 2016-05-01 and includes the next two rows. By adding 4 rows to the SQL, we arrive at the second result.

CUST_ID	TX_DATE	DESCR	SEQ
C001	2016-01-01	Inquiry	1
C001	2016-01-01	Inquiry	2
C001	2016-01-10	Sales	3
C001	2016-01-21	Repeat Inquiry	4
C001	2016-02-10	Repeat Inquiry	5
C001	2016-05-01	Sales	1
C001	2016-05-06	Sales	2
C001	2016-06-10	Inquiry 1	3
C001	2016-09-01	Inquiry 2	1
C002	2016-02-01	Inquiry 1	1
C002	2016-02-25	Inquiry 2	2
C003	2016-02-01	Inquiry 2	1
C003	2016-02-10	Sales	2
C003	2016-02-10	Sales	3
C003	2016-03-10	Inquiry 2	4
C004	2016-04-15	Sales	1

This type of simple "bin fitting" problem arrives frequently. The most efficient pre-12c solution is the MODEL clause, which is generally an order of magnitude slower.

Next, we refine the requirement: the sequencing should always start with a sale, and inquiries outside any 40 day window should have a sequence of 0. Adjusting the solution requires changes to 4 lines again.

CUST_ID	TX_DATE	DESCR	SEQ
C001	2016-01-01	Inquiry	0
C001	2016-01-01	Inquiry	0
C001	2016-01-10	Sales	1
C001	2016-01-21	Repeat Inquiry	2
C001	2016-02-10	Repeat Inquiry	3
C001	2016-05-01	Sales	1
C001	2016-05-06	Sales	2
C001	2016-06-10	Inquiry 1	3
C001	2016-09-01	Inquiry 2	0
C002	2016-02-01	Inquiry 1	0
C002	2016-02-25	Inquiry 2	0
C003	2016-02-01	Inquiry 2	0
C003	2016-02-10	Sales	1
C003	2016-02-10	Sales	2
C003	2016-03-10	Inquiry 2	3
C004	2016-04-15	Sales	1

Finally, the requirement adds a special treatment of inquiries that precede the sale by 10 days or less: these inquiries should have negative sequence numbers. The other sequence numbers are unchanged.

The complete solution introduces the FINAL keyword, which allows us to subtract the total count of inquiries from the cumulative count to produce the negative sequence. The only change is in the method of calculating the sequence.

CUST_ID	TX_DATE	DESCR	SEQ
C001	2016-01-01	Inquiry	-2
C001	2016-01-01	Inquiry	-1
C001	2016-01-10	Sales	1
C001	2016-01-21	Repeat Inquiry	2
C001	2016-02-10	Repeat Inquiry	3
C001	2016-05-01	Sales	1
C001	2016-05-06	Sales	2
C001	2016-06-10	Inquiry 1	3
C001	2016-09-01	Inquiry 2	0
C002	2016-02-01	Inquiry 1	0
C002	2016-02-25	Inquiry 2	0
C003	2016-02-01	Inquiry 2	-1
C003	2016-02-10	Sales	1
C003	2016-02-10	Sales	2
C003	2016-03-10	Inquiry 2	3
C004	2016-04-15	Sales	1

Hierarchical Summary

Once a hierarchy is established, a common requirement is to sum a value from the bottom of the hierarchy to the top. Take for example the venerable SCOTT.EMP table, which implements an employee hierarchy. Suppose we want to calculate the total salary of each manager and all his direct

and indirect reports. Usually, we would need a scalar subquery to recreate part of the hierarchy each time.

MATCH_RECOGNIZE allows us to establish the hierarchy once, then walk through the parts we need. We start with a manager at level N, then match the following rows until we get to the level N again. That gives us the salaries we need, which we can then sum. We then need to start the next match at the very next row, which is possible with the clause AFTER MATCH SKIP TO NEXT ROW. Here is the result:

LVL	ENAME	SAL	SUM_SAL
1	KING	5000	**29025**
2	JONES	2975	**10875**
3	SCOTT	3000	**4100**
4	ADAMS	1100	**1100**
3	FORD	3000	**3800**
4	SMITH	800	**800**
2	BLAKE	2850	**9400**
3	ALLEN	1600	**1600**
3	WARD	1250	**1250**
3	MARTIN	1250	**1250**
3	TURNER	1500	**1500**
3	JAMES	950	**950**
2	CLARK	2450	**3750**
3	MILLER	1300	**1300**

JOIN alternative: comparing tables

If we are comparing tables with primary keys, we can do a FULL JOIN. Unfortunately this may require renaming columns, and comparing columns visually is not easy.

Using MATCH_RECOGNIZE, we can produce output in the classic Change Data Capture format, optionally including the "old" rows so we can see updated values directly beneath the new values.

T1

PK	VAL
1	Same value
2	Delete this
3	Old value

T2

PK	VAL
1	Same value
3	New value
4	Insert this

PK	OP	VAL	OLDRID
2	D	Delete this	AAAkdlAAH...MAAB
3	O	Old value	AAAkdlAAH...MAAC
3	U	New value	AAAkdlAAH...MAAC
4	I	Insert this	

This solution introduces new syntax in the PATTERN clause:
^ indicates the start of a partition
$ indicates the end of a partition
(A | B) says that a row can match either the A condition or the B condition

Avoiding inequality joins

I demonstrate the performance problem with inequality joins: joining two tables of 10,000 lines takes several seconds.

Let us say the join condition involves table T with ranges and table U with single values, we can UNION the tables together and use MATCH_RECOGNIZE. After sorting, we can step through the data, assigning different pattern identifiers: U for the rows from table U and T for the table T rows that fit the U range. We obtain the equivalent of the JOIN, but using a sort step that takes no more time than an equality join.

The presentation ends with some advice on how to identify problems that call for pattern matching solutions.

Contact address:

Stew Ashton
Independent
(no work address)
75017 Paris, France

Phone: (no work phone)
Email stew.ashton@gmail.com
Internet: stewashton.wordpress.com