# 12c SQL Pattern Matching – wann werde ich das benutzen?

**Andrej Pashchenko**
**Trivadis GmbH**
**Düsseldorf**

**Schlüsselworte**
12c, SQL Pattern Matching, analytische Funktionen, Muster, MATCH_RECOGNIZE

**Einleitung**
Die in der Datenbank-Version 12c eingeführte SQL Pattern Matching Klausel (MATCH_RECOGNIZE) wird oft in Zusammenhang mit Big Data, complex event processing, etc. gebracht. Sollen SQL-Entwickler, die (noch) nicht mit ähnlichen Aufgaben konfrontiert werden, das neue Feature erstmal ignorieren? Auf gar keinen Fall! Die SQL-Erweiterung ist mächtig genug, um viele alltäglichen SQL-Aufgaben auf die neue, einfachere und effizientere Weise lösen zu können. Der Vortrag gibt Einblick in die neue SQL-Syntax und präsentiert einige Beispiele, die einem Entwickler einen Vorgeschmack auf mehr geben.

Schon seit langem bietet Oracle Datenbank uns Entwicklern, in SQL eingebaute Mittel, um verschiedene Analyse-Problemstellungen direkt in SQL effizient zu lösen. Dabei reicht es von echten „Hits“ wie den analytischen Funktionen bis zu eher selten eingesetzten SQL-Model- oder PIVOT/UNPIVOT-Klauseln. Könnte die neue Row Pattern Matching Funktionalität eine ähnlich große Bedeutung haben wie damals die Einführung von den analytischen Funktionen in 8i?

**Einstieg in die Syntax.**

Die Erweiterung der SQL-Funktionalität für Row Pattern Matching wird mit Hilfe einer neuen Klausel – MATCH_RECOGNIZE – gemacht. Worum geht es eigentlich bei dem Begriff Pattern Matching, die Mustererkennung auf Deutsch? Bei der Art  Mustererkennung geht es um die datensatzübergreifende Muster in den Daten.
Man muss schon zugeben, der Einstieg in das Thema ist für einen SQL-Entwickler nicht so leicht. Die Syntax ist umfangreich und auf den ersten Blick kompliziert. Eine Vielzahl zum Teil bekannter Techniken und Konzepte kommen hier auf die neue Art und Weise zum Ansatz. Das

Schauen wir uns folgendes Beispiel an. In einer Log-Tabelle werden Protokoll-Daten der täglichen Lade-Jobs eines Datawarehouse Systems gespeichert.

```
SELECT etl_date, mapping_name, elapsed
FROM   dwh_etl_runs;
...
04-NOV-14 MAP_STG_S_ORDER_ITEM +000000 00:14:54.42738
05-NOV-14 MAP_STG_S_ORDER      +000000 00:10:13.44989
05-NOV-14 MAP_STG_S_ORDER_ITEM +000000 00:15:06.24587
05-NOV-14 MAP_STG_S_ASSET      +000000 00:14:15.22855
06-NOV-14 MAP_STG_S_ASSET      +000000 00:14:00.49513
06-NOV-14 MAP_STG_S_ORDER      +000000 00:11:05.07337
06-NOV-14 MAP_STG_S_ORDER_ITEM +000000 00:10:12.67410
07-NOV-14 MAP_STG_S_ORDER_ITEM +000000 00:19:29.64314
```

```
07-NOV-14 MAP_STG_S_ORDER      +000000 00:14:59.80953
07-NOV-14 MAP_STG_S_ASSET      +000000 00:13:33.80789
08-NOV-14 MAP_STG_S_ASSET      +000000 00:10:14.65652
08-NOV-14 MAP_STG_S_ORDER      +000000 00:13:30.77744
08-NOV-14 MAP_STG_S_ORDER_ITEM +000000 00:17:15.11789
```

Listing 1: ETL-Protokoll-Daten

Wir müssen herauszufinden, welche Prozesse (Mappings) an vier aufeinander folgenden Tagen immer schneller wurden. In unserem Beispiel sind es die vier fettmarkierte Datensätze für das Mapping MAP_STG_S_ASSET.

```
1: SELECT *
2: FROM   dwh_etl_runs
3:        MATCH_RECOGNIZE (
4:           PARTITION BY mapping_name
5:           ORDER BY  etl_date
6:           MEASURES  FIRST (etl_date) AS start_date
7:           ,         LAST (etl_date) AS end_date
8:           ,         FIRST (elapsed) AS first_elapsed
9:           ,         LAST (elapsed) AS last_elapsed
10:          ,         AVG(elapsed) AS avg_elapsed
11:          PATTERN   (STRT DOWN{3})
12:          DEFINE    DOWN AS elapsed < PREV(elapsed) );
```

Listing 2: Abfrage mit MATCH_RECOGNIZE-Klausel

Die neue Klausel kommt nach dem Tabellen- oder Viewnamen oder Inline-Subquery in der FROM-Klausel (Zeile 3). Weitere Parameter kommen in Klammern.

Zeilen 4 und 5:

In der Regel müssen wir die Daten partitionieren und sortieren. Das hier ist bekannt aus dem Bereich analytischer Funktionen. Wir wollen die Mappings finden, die zu unserer Fragestellung passen, also müssen wir die Daten separat für jedes Mapping betrachten – die Partitionierung nach MAPPPING_NAME ist dabei nötig. Ebenso ist es wichtig, dass die Daten in chronologischer Reihenfolge betrachtet werden – daher die Sortierung nach ETL_DATE.

Zeilen 6 bis 10:

Bei MEASURES (Zeile 6 bis 10) beschreiben wir die Rückgabewerte aus der MATCH_RECOGNIZE-Klausel. Auf diese kann man in der Hauptabfrage zurückgreifen. Hier können Felder referenziert werden, evtl. mit Verweis auf die Mustervariable (dazu später), SQL-Ausdrücke, SQL-Funktionen, unter anderem einige Aggregatfunktionen (Zeile 10).

Zeilen 11 und 12:

In der Zeile 11 definieren wir das Suchmuster. Das Muster wird definiert mit der Syntax der regulären Ausdrücke. Diese aus Perl kommende Syntax hat sich mittlerweile über alle Schichten der IT-Welt verbreitet. Im Gegensatz zu den herkömmlichen Einsätzen wenden wir diese Syntax hier nicht auf die Zeichen und Strings an, um Treffer bei Texterkennung zu erzielen, sondern die regulären Ausdrücke werden hier auf die sogenannten Mustervariablen (pattern variables) angewendet. Diese Mustervariablen definieren wir selbst im DEFINE-Abschnitt (Zeile 12). In unserem Beispiel

definieren wir eine Mustervariable DOWN. Die Definition ist ein SQL-Ausdruck mit einem booleschen Rückgabewert, ähnlich wie man diese in der WHERE-Klausel verwendet. Hier sind auch sogenannte Navigationsoperatoren (PREV, NEXT, FIRST, LAST) zulässig, sowie Aggregatfunktionen. Die Bedingung in unserem Beispiel bedeutet, dass die ELAPSED vom aktuellen Datensatz kleiner ist als vom vorherigen. Nun zurück zum Muster (Zeile 11). Wofür steht STRT? Das ist eine zweite Mustervariable, die wir nicht im DEFINE explizit definiert haben. Das bedeutet, sie ist immer TRUE oder zutreffend. Diesen Trick brauchen wir, um einen Ausgangspunkt für die Überprüfung der Bedingung der Variable DOWN zu haben. Schließlich, schauen wir dabei auf den vorherigen Datensatz mit PREV() und dieser vorherige Datensatz muss schon Teil des zu überprüfenden Musters sein. Unser regulärer Ausdruck bedeutet: nach dem STRT muss die Bedingung hinter der Variable DOWN drei Mal zutreffen.

Die Idee, die Datensätze im (Zwischen-)Ergebnis untereinander mit Hilfe regulärer Ausdrücke zu untersuchen, ist sehr ungewöhnlich aber auch flexibel und mächtig. Vor allem als SQL-Entwickler mit Jahren Praxis-Erfahrung braucht man eine gewisse Umstellung in der Denkweise. Man muss anfangen „in Mustern“ zu denken. Schnell merkt man, dass Muster eigentlich überall sind.

Das häufigste Beispiel, das zum Veranschaulichen der Mustererkennung gezeigt wird, sind die Aktienkursanalysen. In der Finanzbranche kann man bestimmte Muster in Transaktionen erkennen, die auf eine Geldwäsche hindeuten. Thema Sicherheit und Fraud-Detektion: finde mehrere misslungenen Login-Versuche in den Log-Dateien. Quality of Service in der Telekommunikation: die Analyse der CDR (Call Data Records) auf bestimmte Muster, die eine Aussage über die Qualität der Telefonverbindungen treffen lässt. In CRM-Systemen lassen sich Vorgänge in der Auftragsbearbeitung oder im Bereich Trouble-Ticketing untersuchen. Wie kann man Vorgänge in einem Webseiten-Log identifizieren, die zu einer Benutzer-Session gehören, wobei die Regel dazu fachlich definiert wird und sich ändern kann?

An sich klingen solche Fragestellungen nicht unbedingt kompliziert, sie haben aber eine Gemeinsamkeit: sie ließen sich bisher nicht optimal mit reinen SQL-Mitteln beantworten. Oft führt die Lösung über mehrere Joins oder Semi-/Anti-Joins, über mehrere Ebenen verschachtelten Abfragen mit Einsatz analytischer Funktionen. Man hat auch die Funktionalität aus SQL in PL/SQL ganz oder teilweise ausgelagert.

Vergleichen wir die Lösungen mit herkömmlichen Mitteln und mit Einsatz des Pattern Matching. Als nächstes betrachten wir die Frage, die sich häufig im Zusammenhang mit Auswertungen für Quality of Service stellt: finde kontinuierliche Bereiche oder Lücken in einem numerischen Datenbestand oder bezogen auf Datum/Zeit.

In unserem Beispiel befüllen wir einfach die Tabelle T_GAPS mit nur einer numerischen Spalte ID mit „lückenhaftem“ Bestand aus ganzen Zahlen. Als Ausgabe brauchen wir den Start und Ende des fortlaufenden Bereichs. Listing 3 zeigt eine mögliche Lösung des Problems

```
1:  WITH groups_marked AS (
2:    SELECT id
3:    ,      CASE
4:             WHEN id != LAG(id,1,id) OVER(ORDER BY id) + 1 THEN 1
5:             ELSE 0
6:           END new_grp
7:    FROM   t_gaps)
8:  ,    sum_grp AS (
9:    SELECT id, SUM(new_grp) OVER(ORDER BY id) grp_sum
10:   FROM groups_marked )
```

```
11: SELECT     MIN(id) start_of_range
12: ,          MAX(id) end_of_range
13: FROM       sum_grp
14: GROUP BY   grp_sum
15: ORDER BY   grp_sum;
```

Listing 3: Fortlaufende Bereiche finden

Im ersten Schritt „markieren“ wir Datensätze, an denen ein „Sprung“ stattfindet mit 1 und tragen bei allen anderen 0 ein. Dazu verwenden wir eine analytische Funktion LAG(), um den Wert der Spalte ID mit dem Wert im vorherigen Datensatz zu vergleichen. Diese Markierung hilft uns noch nicht direkt weiter: wir müssen pro fortlaufenden Bereich nur einen Datensatz zurückgeben, also müssen wir noch eine Gruppierungsoperation durchführen. Aber uns fehlt noch das Gruppierungskriterium. Im zweiten Schritt summieren wir die Werte aus dem Markierungsfeld – und fertig ist unser Gruppierungskriterium. Im dritten Schritt müssen wir nur noch Gruppieren min MIN()/MAX() – Funktionen. So genial die Idee mit Markieren und Summieren ist, man versteht auf Anhieb gar nicht, was diese Abfrage eigentlich macht… Ist die neue MATCH_RECOGNIZE-Klausel in dieser Hinsicht besser? Schauen wir mal auf das Listing 4:

```
SELECT *
FROM   t_gaps MATCH_RECOGNIZE (
             ORDER BY id
             MEASURES FIRST(id) start_of_range
             ,        LAST(id)  end_of_range
             ONE ROW PER MATCH
             PATTERN (strt fortlaufend+)
             DEFINE   fortlaufend AS id = PREV(id) + 1
             );
```

Listing 4: Fortlaufende Bereiche mit Pattern Matching erkennen

Bei der Definition der Mustervariable ist hierbei sofort zu erkennen, dass wir solche Datensatze suchen, bei denen sich die Werte um eins unterscheiden. Auch das Muster ist trivial: nach dem Start muss noch mindestens ein Datensatz da sein, bei dem diese Bedingung zutrifft. Die Anweisung ONE ROW PER MATCH führt zur gewünschten Gruppierung.

**Trouble-Ticket-System**

Im nächsten Beispiel betrachten wir ein Trouble-Ticketing-System. Die Tickets werden im Rahmen vom Bearbeitungsprozess mehreren Bearbeitern zugewiesen. Die Zuordnungen werden in der Tabelle TROUBLE_TICKET mit ganz einfachen Struktur (s. Abbildung 1) festgehalten. Wir suchen Tickets, die über eine oder mehrere Stationen zurück an den Bearbeiter gehen, der diese schon mal in Bearbeitung hatte. Auf der Abbildung 1 kann man erkennen: die Tickets Nr. 1, 3, und 4 passen zu unserem Muster.

Wie würde man dieses Problem normalerweise angehen? Zum Beispiel mit einem dreifachen Self-Join (Listing 5):

```
SELECT DISTINCT t1.ticket_id
,      t1.assignee AS first_assignee
,      t3.change_date AS last_change
FROM   trouble_ticket t1, trouble_ticket t2, trouble_ticket t3
WHERE  t1.ticket_id = t2.ticket_id
AND    t1.assignee != t2.assignee
```

```
AND    t2.change_date > t1.change_date
AND    t3.assignee = t1.assignee
AND    t3.ticket_id = t1.ticket_id
AND    t3.change_date > t2.change_date
ORDER BY ticket_id;
```

Listing 5: Mustererkennung mit einem Self-Join

| ID | Assignee | Datum |
|---|---|---|
| 1 | SCOTT | 01.02.2015 |
| 1 | SCOTT | 02.02.2015 |
| 1 | ADAMS | 03.02.2015 |
| 1 | SCOTT | 04.02.2015 |
| 2 | ADAMS | 01.02.2015 |
| 2 | ADAMS | 02.02.2015 |
| 2 | SCOTT | 03.02.2015 |
| 3 | KING | 01.02.2015 |
| 3 | ADAMS | 02.02.2015 |
| 3 | ADAMS | 03.02.2015 |
| 3 | KING | 04.02.2015 |
| 3 | ADAMS | 05.02.2015 |
| 4 | KING | 01.02.2015 |
| 4 | ADAMS | 02.02.2015 |
| 4 | SCOTT | 03.02.2015 |
| 4 | KING | 05.02.2015 |

*Abb. 1: Trouble-Ticket Prozess-Log-Tabelle*

Es ist unschwer zu erkennen, wie unflexibel dieser Ansatz ist. Das Muster ist in der Abfragestruktur wie im Stein gemeißelt. Würden wir die Fragestellung ändern und längere Bearbeitungsketten erkennen wollen, wird sich die Abfrage möglicherweise deutlich ändern. Für jede Bearbeitungsstation muss die Tabelle einmal zusätzlich „gejoined“ werden.

Wie formulieren wir die Abfrage mit Hilfe der Pattern Matching Klausel? Listing 6 zeigt, wie einfach es geht. Wir definieren Zwei Mustervariablen SAME und ANOTHER für denselben und abweichenden Bearbeiter, die sich relativ zu einer Dummy-Variable STRT sehr einfach ausformulieren lassen (Zeile 10,11). Das Suchmuster ist auch keine Hexenwerk: nach dem Start kann noch derselbe Bearbeiter vorkommen, muss aber nicht (* steht für 0 bis N Vorkommnisse). Danach kommt mindestens einmal ein anderer Bearbeiter (+ steht für 1 bis N) und danach wieder der Ursprüngliche (Zeile 9).

```
1: SELECT *
2: FROM   trouble_ticket
3:        MATCH_RECOGNIZE(
4:          PARTITION BY ticket_id
5:          ORDER BY  change_date
6:          MEASURES  strt.assignee as first_assignee
7:          ,         LAST(same.change_date) as letzte_bearbeitung
8:          AFTER MATCH SKIP TO FIRST another
```

```
9:          PATTERN (strt same* another+ same+)
10:          DEFINE same AS same.assignee = strt.assignee,
11:                 another AS another.assignee != strt.assignee );
```

Listing 6: Trouble-Ticketing mit Pattern Matching

Neu ist in dieser Abfrage die Klausel zum Wiederaufsetzen nach dem Finden des Musters (Zeile 8). Diese definiert, wo die Suche nach einem Treffer weiter gehen soll und sorgt z.B. dafür, dass beim Ticket mit der ID=3 zwei Muster gefunden werden.

**Gruppieren über unscharfe Kriterien**

Das Problem stellt sich dar, wenn man Datensätze gruppieren möchte und dabei kein eindeutiges Kriterium in den Daten hat. Unter Grupperein versteht man hier sowohl das Reduzieren der Datenmenge auf ein Datensatz pro Gruppe mit oder ohne das Aggregieren als auch die Gruppenzuordnung beim Beibehalten der ursprünglichen Datenmenge.

Denken wir an die Zuordnung der protokollierten Benutzeraktionen aus einer Logging-Tabelle einer bestimmten Benutzer-Session. Manchmal wird der Begriff „Session“ statt rein technischen eher einer fachlichen Definition unterliegen, z.B. alle Aktionen, zwischen denen weniger als zehn Minuten vergehen, gehören zu einer Session. Eine SQL-Abfrage mit Einsatz analytischer Funktionen wird zwar möglich, aber alles andere als trivial sein. Die Definition vom Muster für eine SQL Pattern Matching Abfrage ist nach Ansicht des Autors einfach verständlich und folgt im Wesentlichen der Beschreibung in menschlicher Sprache:

```
PATTERN (STRT SESS+)
DEFINE SESS AS SESS.ins_date – PREV(SESS.ins_date)<= 10/24/60
```

Nun definieren wir den Begriff der Session anders: zu einer Session gehören alle Aktionen im Zeitraum von 30 Minuten. Wird der Zeitraum überschritten, fängt eine neue Session an. Hier ist eine Art der Rekursion im Spiel, denn wir werden erst wissen, ab welchem Zeitpunkt weitere 30 Minuten zu messen sind, nachdem wir die erste Session identifiziert haben. Hier würden Bestimmt viele Entwickler zu PL/SQL greifen. Eine mögliche reine SQL-Lösung, die neben analytischen Funktion noch einen Self-Join und eine hierarchische Abfrage kombiniert, gibt zwar die Antwort, kann aber nicht als gut lesbar und verständlich bezeichnet werden. Eine Definition des Musters sieht dagegen kaum komplizierter als im vorherigen Fall:

```
PATTERN (SESS+)
DEFINE SESS AS ins_date < FIRST(ins_date) + 30/24/60
```

Ein anderes Beispiel aus dieser Kategorie ist die Gruppierung über laufende Aggregate. Wir sollen z.B. den Kundenbestand nach Alter sortiert so in einzelne Gruppen verteilen, dass der Gesamtumsatz in jeder Gruppe die 200000$ nicht übersteigt. Wir können zwar eine analytische Funktion zur Ermittlung des laufenden Umsatzes einsetzen. Aber, um das Ergebnis dieser Funktion weiter als Kriterium für den Anfang einer neuen Gruppe zu verwenden, werden wir die Abfragen verschachteln müssen.

```
WITH q AS (SELECT  c.cust_id, c.cust_year_of_birth
           ,       SUM(s.amount_sold) cust_amount_sold
           FROM    customers c JOIN sales s ON s.cust_id = c.cust_id
           GROUP BY c.cust_id, c.cust_year_of_birth
           )
SELECT *
FROM   q
```

```
MATCH_RECOGNIZE(
ORDER BY cust_year_of_birth
MEASURES MATCH_NUMBER() gruppe
,        SUM(cust_amount_sold) running_sum
,        FINAL SUM(cust_amount_sold) final_sum
ALL ROWS PER MATCH
PATTERN (gr*)
DEFINE gr AS SUM(cust_amount_sold)<=200000 );
```

Listing 6: Aggregatfunktionen in der Mustervariablendefinitionen

Listing 6 zeigt, die Aggregatfunktionen können einfach in der Definition der Mustervariablen eingesetzt werden. Somit wird das Ausformulieren der Bedingung zum Kinderspiel: wir schreiben die Fachanforderung fast einfach nur auf. Eine Gruppe bleibt solange bestehen bis die laufende Summe kleiner 200000 ist. Stimmt die Bedingung nicht, fangen wir mit der neuen Gruppe an.

Aggregatfunktionen können nicht nur in der Variablendefinition verwendet werden, sondern auch in der MEASURES-Klausel (s. Listing 6). Hier sind zwei Varianten möglich: sogenannte RUNNING (Default) oder FINAL-Semantik. Die erste stellt eine laufende Aggregation dar und die letztere die Aggregation über das gesamte Muster. In der DEFINE-Klausel sind logischerweise nur RUNNING-Aggregationen zulässig, denn beim Überprüfen der Bedingungen für Mustervariablen weiß die Abfrage noch nicht, wie das endgültig erkannte Muster aussehen wird.

**Fazit**

Die neue Syntax mag auf den ersten Blick etwas ungewöhnlich sein. Es lohnt sich aber für einen SQL-Entwickler, einen ersten Schritt zu wagen und die neue Syntax kennenzulernen. Wer sich noch unsicher mit regulären Ausdrücken fühlt, ist es ein Anlass mehr das zu überwinden. Übrigens, wird es keine Oracle-proprietäre SQL-Erweiterung sein, sondern die Syntax ist seit 2007 als Vorschlag für den ANSI-SQL-Standard aufgenommen worden. Es ist ein Tool mehr, das ein Entwickler in seiner Toolbox bereit haben sollte. Ich werde es in Zukunft für viele Problemstellungen in Betracht ziehen. Handelt sich hierbei um ein Muster? Diese Frage werde ich mir stellen. Wenn ich als Lösung über Self-Joins, Verschachtelung der Abfragen mit analytischen Funktionen nachdenke, soll es ein Hinweis sein, dem SQL Pattern Matching eine Chance zu geben. Natürlich sollte wie immer das Testen der Alternativen nicht zu kurz kommen. Auch, was die Performance angeht. Generell kann man sagen, dass Pattern Matching CPU-intensiv ist. Aber wenn man dadurch z.B. einige Self-Joins eliminieren kann, hat man unter dem Strich einen Performancegewinn.

**Kontaktadresse:**
Andrej Pashchenko
Trivadis GmbH
Werdener Str., 4
D-40227 Düsseldorf

| | |
|---|---|
| Telefon: | +49 (0) 211-58 66 64 70 |
| Fax: | +49 (0) 211-58 66 64 71 |
| E-Mail | andrej.pashchenko@trivadis.com |
| Internet: | www.trivadis.com |