Komfortable Lösung mit dem EXASolution SQL-Preprocessor

Im ersten Teil dieser Beitragsserie haben wir nach einer allein auf SQL basierenden Lösung für das Verteilen von Werten eines Kopfdatensatzes auf zugehörige Positionsdatensätze gesucht. Die Verteilung von eventuell entstehenden Rundungsdifferenzen führte zu einem sehr komplexen SQL-Ausdruck, bestehend aus einer Vielzahl analytischer Funktionen. Die Anwendung dieser Lösung erweist sich zwar durchaus als performant, ist aber gleichzeitig aufwendig, wenig wartungsfreundlich und aufgrund der Komplexität auch fehleranfällig. In diesem Teil der Beitragsserie werden wir nun eine komfortable Lösung für die In-Memory-Datenbank EXASolution vorstellen.
Dabei kommt der SQL-Preprocessor der EXASolution-Datenbank zum Einsatz. Der SQL-Preprocessor erlaubt das Parsen und Verändern der SQL-Befehle, bevor diese von der Datenbank verarbeitet werden. Beliebige Code-Fragmente eines SQL-Befehls können so durch andere Code-Fragmente ersetzt werden. Damit wird es möglich, für komplexere SQL-Ausdrücke einfachere Schreibweisen anzugeben, die dann vom SQL Preprocessor übersetzt werden.

Definition einer eigenen analytischen Funktion DISTRIBUTE_VALUE

Um die Komplexität des SQL-Ausdrucks aus dem ersten Teil der Beitragsserie zu reduzieren, verwenden wir den Code des Ausdrucks als Vorlage für die Definition einer eigenen analytischen Funktion. Dieser Funktion geben wir den Namen DISTRIBUTE_VALUE. Da es sich um eine analytische Funktion handeln soll, sehen wir folgende Syntax vor. *(siehe Listing 1)*

Das erste Argument der Funktion DISTRIBUTE_VALUE gibt dabei den zu verteilenden Wert des Kopfdatensatzes an. Das zweite Argument bestimmt die Anzahl der Nachkommastellen auf die gerundet werden soll. Die Klausel PARTITION BY mit dem dritten Argument definiert die Partitionen der analytischen Funktion. Um das Beispiel übersichtlich zu halten, gehen wir davon aus, dass PARTITION BY stets angegeben wird. Die bei analytischen Funktionen übliche ORDER BY-Klausel sowie eine Window-Klausel sehen wir nicht vor. Eine umfangreichere Lösung ohne diese Einschränkungen kann aber ebenfalls mit dem hier beschriebenen Verfahren realisiert werden.

Parsen des SQL-Statements

Um den SQL-Befehl mit dem SQL-Preprocessor zu parsen, stellt die EXASolution-Datenbank die sqlparsing-Library zur Verfügung. Die bereitgestellten Library-Funktionen ermöglichen das Zerlegen des SQL-Befehls in einzelne Tokens (Funktion sqlparsing.tokenize). Diese Tokenliste kann dann zum Beispiel nach Schlüsselwörtern und Ausdrücken durchsucht werden (Funktion sqlparsing.find), und die zugehörigen Teile des SQL-Text können verändert und neu zusammengesetzt werden. Für das Parsen mit der sqlparsing-Library wird ein eigenes LUA-Skript transform_distribute_value mit einer Funktion process_distribute_value in der Datenbank erstellt. *(siehe Listing 2)*

SQL Preprocessor Skript

Zusätzlich wird ein zweites LUA-Skript erstellt, das später als Hauptskript für den SQL Preprocessor verwendet wird. Nur in diesem Hauptskript sind die Funktionen getsqltext und setsqltext der sqlparsing-Library verfügbar. Die Funktion getsqltext liefert den SQL-Text des aktuellen SQL-Statements während mit setsqltext der SQL-Text des aktuellen SQL-Statements durch einen eigenen Text überschrieben werden kann. Der eigene Text wiederum wird durch den Aufruf der zuvor beschriebenen Parser-Funktion erzeugt. Unser Hauptskript für den SQL-Preprocessor sieht damit wie folgt aus. *(siehe Listing 3)*

Damit dieses Skript als Hauptskript vom SQL Preprocessor verwendet wird, muss es per ALTER SESSION oder per ALTER SYSTEM als solches angegeben werden. *(siehe Listing 4)*

Anwendung der eignen analytischen Funktion

Nun können wir unsere neu definierte analytische Funktion DISTRIBUTE_VALUE verwenden. Das SQL-Statement aus dem ersten Teil dieser Beitragsserie wird nun extrem vereinfacht und damit deutlich übersichtlicher. *(siehe Listing 5)*

Komfortable Lösung mit dem EXASolution SQL-Preprocessor

Fazit

Der SQL-Preprocessor ermöglicht die Definition eigener neuer SQL-Ausdrücke, die von einer Parser-Funktion durch bestehende SQL-Ausdrücke der EXASolution-Datenbank ersetzt werden. Damit können auch komplexe Aufgaben, für die eigentlich unübersichtliche SQL-Ausdrücke erforderlich sind, durch einfache und verständliche Ausdrücke gelöst werden.

Die hier dargestellte analytische Funktion DISTRIBUTE_VALUE für die Verteilung von Werten zeigt exemplarisch, wie komplexe fachspezifische Aufgaben in einer sehr eleganten Lösung gekapselt und einfach wiederverwendet werden können. Sowohl SQL-Abfragen als auch ETL-Prozesse können auf diese Weise stark vereinfacht werden.
Da durch den SQL-Preprocessor der SQL-Text manipuliert wird – also SQL-Fragmente durch andere SQL-Ausdrücke ersetzt werden – ist der Einsatz jedoch auf Aufgaben beschränkt, die sich mit reinem SQL lösen lassen. Diese Beschränkung bringt jedoch gleichzeitig den Vorteil mit sich, dass – sofern eine Lösung möglich ist – diese meist auch sehr performant ist.
Im dritten und letzten Teil dieser Beitragsserie werden wir auf eine reine SQL-Lösung verzichten und unserer Beispielaufgabe mit Hilfe von User-Defined Analytic Functions in der Oracle-Datenbank umsetzen. In einem Vergleich werden wir beide Lösungen gegenüberstellen.

Komfortable Lösung mit dem EXASolution SQL-Preprocessor

Listing 1:

```
DISTRIBUTE_VALUE (expression, expression) OVER (PARTITION BY expression)
```

Listing 2:

```
create or replace script 'my_schema.transform_distribute_value()
AS
function process_distribute_value(sqltext)
while (true) do
  local tokens = sqlparsing.tokenize(sqltext)

  -- "DISTRIBUTE_VALUE (" suchen
  local startDV = sqlparsing.find(tokens, 1, true, false
                                , sqlparsing.iswhitespaceorcomment
                                , 'DISTRIBUTE_VALUE', '(' )
  if (startDV==nil) then
    break;
  end

  -- Passende ")" für "(" von "DISTRIBUTE_VALUE" suchen
  local endDV = sqlparsing.find(tokens, startDV[2], true, true
                              , sqlparsing.iswhitespaceorcomment
                              , ')')
  if (endDV==nil) then
    error("function DISTRIBUTE_VALUE not ended properly, missing ')'")
    break;
  end

  -- "OVER (PARTITION BY" suchen
  local startDVOver = sqlparsing.find(tokens, startDV[2], true, true
                                    , sqlparsing.iswhitespaceorcomment
                                    , ')', 'OVER', '(', 'PARTITION', 'BY')
  if (startDVOver==nil) then
    error("function DISTRIBUTE_VALUE not ended properly"..
          ", missing 'OVER (PARTITION BY'")
    break;
  end
  if (startDVOver[1]~=endDV[1]) then
    error("function DISTRIBUTE_VALUE not ended properly"..
          ", missing 'OVER (PARTITION BY'")
    break;
```


```
end;

-- Zweites Argument von DISTRIBUTE_VALUE suchen
local startPrec = sqlparsing.find(tokens, startDV[2]+1, true, true
                                , sqlparsing.iswhitespaceorcomment
                                , ',' )
if (startPrec==nil) then
  error("Function DISTRIBUTE_VALUE should be called with 2 arguments.")
  break;
end
-- Passende ")" für "(" von "OVER (PARTITION BY" suchen
local endDVOver = sqlparsing.find(tokens, startDVOver[3], true, true
                                , sqlparsing.iswhitespaceorcomment
                                , ')' )
if (endDVOver==nil) then
  error("function DISTRIBUTE_VALUE not ended properly"..
        ", missing ')' for clause 'OVER (PARTITION BY'")
  break;
end
local arg1=table.concat(tokens, '', startDV[2]+1, startPrec[1]-1)
local arg2=table.concat(tokens, '', startPrec[1]+1, startDVOver[1]-1)
local arg3=table.concat(tokens, '', startDVOver[5]+1, endDVOver[1]-1)
local caseStmt='case \
     when row_number () over (partition by '..arg3..' order by 1) \
       <= div (abs ('..arg1..' - (count (1) over (partition by '..arg3..') \
        * cast ('..arg1..' / count (1) over (partition by '..arg3..') \
            as decimal (10,'..arg2..')))), 1 / power (10, '..arg2..')) \
     then cast ('..arg1..' / count (1) over (partition by '..arg3..') \
            as decimal (10,'..arg2..')) \
        + sign ('..arg1..' - count (1) over (partition by '..arg3..') \
        * cast ('..arg1..' / count (1) over (partition by '..arg3..') \
            as decimal (10,'..arg2..'))) \
        * 1 / power (10, '..arg2..') \
     when row_number () over (partition by '..arg3..' order by 1) \
        = 1 + div (abs ('..arg1..' - (count (1) over (partition by '..arg3..') \
        * cast ('..arg1..' / count (1) over (partition by '..arg3..') \
            as decimal (10,'..arg2..')))), 1 / power (10, '..arg2..')) \
     then cast ('..arg1..' / count (1) over (partition by '..arg3..') \
            as decimal (10,'..arg2..')) \
        + mod ('..arg1..' - count (1) over (partition by '..arg3..') \
        * cast ('..arg1..' / count (1) over (partition by '..arg3..') \
```


```
              as decimal (10,'..arg2..')), 1 / power (10, '..arg2..')) \
       else cast ('..arg1..' / count (1) over (partition by '..arg3..') \
              as decimal (10,'..arg2..')) \
       end'


  sqltext=table.concat(tokens, '', 1, startDV[1]-1)..caseStmt..
          table.concat(tokens, '', endDVOver[1]+1)
end
return sqltext
end
/
```

Listing 3:

```
create or replace script 'my_schema.preprocess_distribute_value()
as
import('my_schema.transform_distribute_value', 'transform_distribute_value' )
sqlparsing.setsqltext(
  transform_distribute_value.process_distribute_value(sqlparsing.getsqltext()))
/
```

Listing 4:

```
ALTER SESSION SET sql_preprocessor_script=my_schema.preprocess_distribute_value;
```

Listing 5:

```
select p.*, distribute_value (b.vk, 2) as vk_verteilt
  from bestellung b
     , bestellposition p
 where b.bnr = p.bnr;
```