Eleganten und effizienten Code schreiben

Jürgen Sieben, ConDeS GmbH & Co. KG

In dieser Folge widmen wir uns dem elegantesten, sichersten und schnellsten denkbaren Code: keinem Code.

Oft wird darauf hingewiesen, dass ein Problem, wenn es ohne Code direkt in SQL gelöst werden kann, dort auch gelöst werden sollte – normalerweise verbunden mit dem Hinweis auf weniger genutzte SQL-Fähigkeiten wie analytische Funktionen, „log error"-Klauseln oder Multi-Table-Inserts. So richtig und gut diese Hinweise sind, so wäre doch aufgrund der langen Verfügbarkeit dieser Optionen ein Upgrade angezeigt. Obwohl, den Autor überkommt schon eine gelinde Verzweiflung, wenn er in einem aktuellen Buch über PL/SQL, das mit griffigen Slogans wie „Advice from the Experts" und „underused features" wirbt, ein Codebeispiel wie in *Listing 1* findet. Er ist geneigt, den Code wie in *Listing 2* umzuschreiben.

```
... where date_column > sysdate - 0.01 -- that's roughly 14 minutes
```

Listing 1

```
... where date_column > sysdate - interval '10' minute -- that's simply
10 minutes
```

Listing 2

ID	Wert	Gueltig_von	Gueltig_bis
123	A	01.01.2017	01.01.2017 23:59:59
123	B	02.01.2017	02.01.2017 23:59:59
123	A	03.01.2017	03.01.2017 23:59:59
123	A	04.01.2017	04.01.2017 23:59:59
234	C	01.01.2017	01.01.2017 23:59:59
234	D	02.01.2017	02.01.2017 23:59:59
234	E	03.01.2017	03.01.2017 23:59:59
234	E	04.01.2017	04.01.2017 23:59:59

Tabelle 1

Aber sei's drum. Die Funktion gibt es erst seit etwa zwanzig Jahren und man muss man ja nicht jede Neuerung mitmachen. Die Erweiterungen, die SQL in den letzten Jahren erfahren hat, lösen natürlich keine Revolution mehr aus, sondern sind Erweiterungen spezieller Probleme, dort aber können sie erhebliches Potenzial entfalten. Nachfolgend ein Beispiel aus einem Projekt.

Das Projekt

In einem Data Warehouse sind Daten zu Grundstücken erfasst. In der ersten Version wurden die Grundstücksdaten täglich neu geladen und zeitlich von den Altdaten abgegrenzt. Da allerdings viele Grundstücke verwaltet werden und die Änderungshäufigkeit sich in engen Grenzen hielt (wovon der Begriff „immobil", der in diesem Zusammenhang verwendet wird, ja auch zeugt), kamen nach drei Jahren, in denen die Zeilenzahl von etwa einer Million auf mehr als eine Milliarde Zeilen angewachsen war, Zweifel an der Sinnhaftigkeit dieses Datenmodells auf. Die Daten sollten konsolidiert und nur noch dann zeitlich abgegrenzt werden, wenn sich ein Attribut auch tatsächlich ändert. Nur, wie macht man das? Die Datensätze könnten sich über die Zeit durchaus wie in *Tabelle 1* entwickelt haben.

Würde man nun dem ersten Reflex folgen und Gruppen-Funktionen einsetzen, um für die Kombination aus ID und Wert das erste beziehungsweise letzte Gültigkeitsdatum zu finden, ginge das für ID 234, nicht aber für ID 123. Man würde falsche Ergebnisse erhalten, weil Wert A vom 01.01.2017 bis zum 04.01.2017 abgegrenzt und Wert B vom 02.01.2017 damit zeitlich überlagert würde.

Analytische Funktionen helfen hier nicht, denn sie benötigen ebenfalls ein klares Kriterium zur Gruppierung, unterliegen also dem gleichen Problem wie Gruppen-Funktionen. Eine Alternative wäre eine hierarchische Abfrage, die bei dem Datensatz vom 01.01. beginnt und so lange hierarchisch zugeordnete Daten sucht, solange alle relevanten Attribute gleich und das untergeordnete Startdatum eine Sekunde hinter dem aktuellen Enddatum liegt.

Das ginge zwar, doch wo beginnt man mit der jeweiligen Suche? Wir könnten bei jeder Zeile beginnen und auf dem Ergebnis dann eine Gruppierung vor-

nehmen, doch ist das bei einer Milliarde Zeilen eine so gute Idee? Also doch PL/SQL? Nein, denn ab Version 12*c* ist diese Fragestellung ein ideales Beispiel für das Row Pattern Matching, die Mustersuche in SQL. Dieses Feature ähnelt syntaktisch sowohl den analytischen Funktionen als auch der Model-Klausel, bietet aber mehr Flexibilität, da es mit dieser Funktion möglich ist, Muster zu erkennen und auszuwerten.

Der Datenbestand wird bei dieser Klausel durch eine „partition by"-Klausel separiert und durch eine „order by"-Klausel innerhalb der Partition sortiert, wie man das auch bei analytischen Funktionen kennt. Nun kommt die Mustererkennung, die letztlich einen trivialen Vergleich ausführen muss: Gruppiere so viele Zeilen zusammen, wie innerhalb der Partition durch gleiche Attributwerte gekennzeichnet sind. Liefere anschließend das erste und letzte Datum der erkannten Gruppierung. *Listing 3* zeigt die Abfrage.

```
select id, wert, gueltig_von, gueltig_bis
      from grundstuecke
  match_recognize(
          partition by id
          order by gueltig_von
          measures wert as wert,
              first(gueltig_von) as gueltig_von,
              last(gueltig_bis) as gueltig_bis
          one row per match
          pattern (strt same*)
          define same as wert = prev(wert) -- hier würden alle Attribute
verglichen
              and numtodsinterval(gueltig_von - prev(gueltig_bis), 'day')
                  = interval '1' second
          );

ID    WERT   GUELTIG_VON           GUELTIG_BIS
----  -----  --------------------  -------------------
123   A      01.01.2017 00:00:00   02.01.2017 23:59:59
123   B      03.01.2017 00:00:00   03.01.2017 23:59:59
123   A      04.01.2017 00:00:00   05.01.2017 23:59:59
234   C      01.01.2017 00:00:00   01.01.2017 23:59:59
234   D      02.01.2017 00:00:00   02.01.2017 23:59:59
234   E      03.01.2017 00:00:00   04.01.2017 23:59:59
```

Listing 3

Fazit

Die Abfrage hat die Daten korrekt abgegrenzt und das Problem in einem Durchlauf gelöst. Es geht gar nicht so sehr um die konkreten syntaktischen Details, sondern der Autor möchte gern eine Einladung aussprechen, sich mit SQL zu beschäftigen. Diese Beschäftigung endet nie, nicht mit einer Datenbank-Version und nicht bei einem vorgegebenen Detail-Kenntnisgrad. Gerade beim Row Pattern Matching ist es ihm zu Beginn schwergefallen, sinnvolle Anwendungsbereiche zu identifizieren. Erst nachdem er sich die Möglichkeiten intensiver angesehen hatte, wurde ihm klar, wie häufig wir in Datenbanken von Mustern umgeben sind, die mit diesem neuen Mittel analysiert werden können.

In einer Schulung bat ein Teilnehmer den Autor, in einem beinahe vollbesetzten Stadion noch mindestens zwei nebeneinander liegende, freie Plätze zu finden: Mustersuche. Ist der Blick erst einmal geschärft, werden immer mehr Anwendungsbereiche offensichtlich und immer weniger Code nötig. Das Row Pattern Matching ist hierfür nur eines von vielen Beispielen.

Hinweis: Der Code zum Artikel steht unter „*www.doag.org/go/redstack/201801/listings*"

Jürgen Sieben
j.sieben@condes.de

Hilfe bei langsamen Apex-Applikationen

Die Stärken von Apex als Rapid-Development-Tool werden in Unternehmen sehr geschätzt, weiß Kai Glittenberg von Apps Associates. Gerade der Zeit- und Kostenfaktor ist dabei oft eine treibende Rolle für Fach- und IT-Abteilungen, um Software und Schnittstellen mit Apex umzusetzen. Das Daten- und Benutzerwachstum und die Nutzungsfrequenz der Software werden dabei im Vorfeld häufig nicht mitbedacht oder unterschätzt. Nach einiger Zeit oder mit steigendem Funktionsumfang können Performance-Probleme in den Anwendungen auftreten. In diesem Vortrag auf der Apex Connect 2017 geht Glittenberg auf wichtige Punkte zur Optimierung ein. Dabei beantwortet er auch folgende Fragen: Wie cached Apex? Wie cached die Middleware? Was muss ich bei der Architektur in Betracht ziehen? Wie nutze ich Collections effektiv? Das Video steht unter „*https://www.doag.org/de/home/news/aufgezeichnet-hilfe-bei-langsamen-apex-applikationen/detail*".