Wider die CASE-Anweisung

Jürgen Sieben, ConDeS GmbH & Co. KG

Vor einiger Zeit saß der Autor in einem Projekt mit der Fachseite zusammen. In einem Krankenhaus sollten aus Abrechnungsdaten für Privatpatienten Übersichten für die verschiedenen Privat-Ambulanzen erstellt werden. Dabei stellte sich das Problem, die einzelnen Leistungen den richtigen Kliniken und Ambulanzen zuzuordnen und die privaten von den gesetzlichen Leistungen zu trennen.

Das sei kein Problem, beschied die Fachseite, weil die privaten Leistungen ganz einfach an der Aufnahmeart zu erkennen seien: Enthalte diese ein „C“, handele es sich um eine Chefarzt-Leistung und mithin um eine Privatleistung. Auf die Frage, ob dies immer so sei, lautete die Antwort: „Ja – also fast immer jedenfalls“, war die Antwort. Eine genauere Analyse der Situation förderte dann etwas mehr als vierzig verschiedene Fälle, die zu unterscheiden waren, ans Tageslicht.

Die Analyse zeigte, dass insgesamt vier verschiedene Kriterien zu beachten waren, die einen signifikanten Wert enthalten konnten, oder auch nicht; zudem ergab sich, dass die verschiedenen Kriterien sich gegenseitig überstimmten: Sagte Kriterium 1, dass es sich nicht um eine Privatleistung handelt, konnte dies von Kriterium 2 übersteuert werden, analog übersteuerte Kriterium 3 das erste und zweite, Kriterium 4 alle anderen. In der SELECT-Anweisung, die ich nun zu erstellen hatte, würde ich also ein gut getestetes und komplexes Gewebe aufeinander aufbauender CASE-Anweisungen benötigen, sortiert vom Speziellen zum Allgemeinen.

Besonders unangenehm an der Situation war, dass eine Erweiterung der zu unterscheidenden Fälle mich zwingen würde, das austarierte Gewebe erneut anzupassen. Dass sich im Laufe der Zeit neue Fälle ergeben würde, war keine Befürchtung, sondern Gewissheit. Ähnliche Situationen hat jeder von uns sicher auch schon einmal erlebt. Nur – was tun?

Auch, wenn das geschilderte Problem wie ein reines SQL-Problem anmutet, kommen solche Fälle mindestens ebenso häufig auch in PL/SQL vor, wahrscheinlich sogar eher noch häufiger, weil man sich aufgrund der zu erwartenden Komplexität ohnehin bereits entschlossen hat, die Lösung in einem PL/SQL-Package zu suchen. Das ist aber in vielen Fällen unnötig, denn komplexe CASE-Anweisungen deuten häufig darauf hin, dass die Datenbank nicht über die notwendigen Kenntnisse verfügt, um ein Problem in SQL zu lösen.

Aus einem Projekt erinnert sich der Autor beispielsweise an eine ausufernde CASE-Anweisung, in der wegen verschiedenster Informationen Leistungen zu Leistungsgruppen organisiert wurden; in einem anderen Fall musste großer Aufwand betrieben werden, um herauszufinden, welche Vorgänge, die einem Dezernat zugeordnet waren, für einen Abteilungsleiter sichtbar sein sollten. In beiden Fällen fehlten der Datenbank wichtige Informationen: Im ersten Fall müsste in einer Tabelle die Zuordnung von Leistungen zu Leistungsgruppen verwaltet werden, im zweiten Fall gab es keine Tabelle, die das Organigramm des Unternehmens beinhaltete, sodass die Datenbank nicht wusste, welches Dezernat zu welcher Abteilung gehört.

Dinge dieser Art sind oft für die Anwendung selbst nicht von Interesse, werden aber für das Berichtswesen oder für eine Berechtigungsprüfung benötigt. Ge-

	A	B	C	D	E	F
1	**K1**	**K2**	**K3**	**K4**	**Hauptgruppe**	**Nebengruppe**
2	AESTHETIK	ALL	6	ALL	AMBULANT	ÄSTHETIK
3	AESTH-S	ALL	6	ALL	STATIONÄR	ÄSTHETIK
4	ALL	E-PRLIQ-APO	100	ALL	APOTHEKE	STANDORT1
5	ALL	E-PRIVLIQ-S	ALL	PHYSIO_A	STATIONÄR	PHYSIO
6	ALL	E-PRIVLIQ-S	ALL	PHYSIO_B	STATIONÄR	PHYSIO
7	ALL	WLCHPLUS	ALL	ALL	H-PLUS	ALL
8	AMB-PR	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
9	AMB-PR-DUI	E-PRLIQ-APO	100	ALL	APOTHEKE	STANDORT2
10	AMB-PRK	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
11	AMB-PRK-KVB	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
12	AMB-PRK-PB	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
13	AMB-PR-KVB	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
14	AMB-PR-OB	E-PRLIQ-APO	100	ALL	APOTHEKE	STANDORT3
15	AMB-PR-PB	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
16	AMB-PR1	E-PRIVLIQ-A	ALL	ALL	AMBULANT	ALL
17	HKS	E-PRIVLIQ-S	ALL	ALL	STATIONÄR	ALL
18	HKSK	E-PRIVLIQ-S	ALL	ALL	KONSIL	ALL
19	HKSK-KNA	E-PRIVLIQ-S	ALL	ALL	KONSIL	ALL
20	HKSK-KVB	E-PRIVLIQ-S	ALL	ALL	KONSIL	ALL
21	HKS-KNA	E-PRIVLIQ-S	ALL	ALL	STATIONÄR	ALL
22	HKS-KVB	E-PRIVLIQ-S	ALL	ALL	STATIONÄR	ALL

Tabelle 1

nauso war es auch im angesprochenen Beispiel der Privatleistungen im Krankenhaus. Doch es bleibt natürlich die Frage, warum eine einfache Tabelle mit den verschiedenen Fällen nun die CASE-Anweisung obsolet machen kann. Wie organisieren wir zum Beispiel, dass Kriterium 2 immer Kriterium 1 überstimmt? Wie können wir erreichen, dass Kriterien, die nicht gesetzt sind, von SQL ignoriert werden? Betrachten wir uns das Problem an einem Beispiel an. In einer Falltabelle sind die Leistungen hinterlegt, zudem sind dort die Kriterien K1 bis K4 für den jeweiligen Fall mit einem beliebigen Wert belegt. NULL-Werte sind möglich und sollen als nicht bekannt/nicht relevant behandelt werden.

Tabelle 1 zeigt einige der Daten der für die vier Kriterien K1 bis K4 und die aus der jeweiligen Kombination resultierenden Haupt- und Nebengruppen, die im Bericht zur Gruppierung der Ergebnisse verwendet werden. Ein Kriterium, das beliebige Werte enthalten kann, wurde hier mit „ALL" belegt. Nun benötigen wir eine Auswertung, die uns zeigt, welche Zuordnung für eine gegebene Konstellation die passendste ist.

Sollte ein Kriterium der Falltabelle mit einem Kriterium der Kriterien-Tabelle übereinstimmen, werden ein Join zwischen den beiden Tabellen zu „TRUE" evaluiert und die Zeilen verbunden. Nicht übereinstimmende Kriterien (auch „NULL"-Werte) der Fall-Tabelle werden nur dann zu „TRUE" evaluieren, wenn in der entsprechenden Kriterien-Spalte der Wert „ALL" enthalten ist.

Da in der Wertigkeit K4 vor K3 etc. liegt, können wir die passendste Kombination herausfinden, indem wir alle passenden Kriterien mit einem Gewichtungsfaktor versehen, diese aufsummieren und die Zeile mit der höchsten Summe auswählen. Dies leistet die folgende Abfrage, die die Falldaten mit der Kriterien-Tabelle abgleicht *(siehe Listing 1)*. Dabei ist zu beachten, dass die Verwendung der „ROW_LIMITING"-Klausel wegen der Partitionierung nach „PAT" nicht verwendet werden kann.

Die Abfrage bewertet die verschiedenen Kriterien mit 1, 2, 4 oder 8, wenn die jeweiligen Kriterien erfüllt und ungleich „ALL" sind. Über die Summe dieser Bewertung wird eine Rangfolge gebildet und nur der erste Rang gewählt. Diese konkrete Lösung ist natürlich nicht für alle denkbaren Fälle geeignet, sondern eine Lösung für das konkrete Problem, doch ist mir wichtig, den grundsätzlichen Lösungsansatz, CASE-Anweisungen durch Tabellen und SQL zu ersetzen, ins Bewusstsein zu holen.

```
select *
  from (select f.*, k.hauptgruppe, k.nebengruppe,
         rank() over (
         order by
           decode(k.k1, 'ALL', 0, 1)
         + decode(k.k2, 'ALL', 0, 2)
         + decode(k.k3, 'ALL', 0, 4)
         + decode(k.k4, 'ALL', 0, 8) desc) rang
      from fall_tabelle f
      join kriterien k
        on k.k1 in (f.k1, 'ALL')
       and k.k2 in (f.k2, 'ALL')
       and k.k3 in (f.k3, 'ALL')
       and k.k4 in (f.k4, 'ALL'))
  where rang = 1
```

Listing 1

```
  with session_state as(
         select rules_helper.get_firing_item firing_item,
            to_number(v('P1_CHILD'), '9990') P1_CHILD,
            to_number(v('P1_PARENT'), '9990') P1_PARENT
         from dual)
select /*+ NO_MERGE(s) */
         r.rule_id, r.rule_name, r.action_type_id, r.action_attribute
  from sct_bl_rules r
  join session_state s
      on r.firing_item = s.firing_item
 where (r.rule_id = 98 and (bl_pkg.has_parent (p1_child) = 'Y'))
      or (r.rule_id = 99 and (bl_pkghas_children(p1_parent) = 'N'))
 order by r.rule_sort_seq
 fetch first 1 row only
```

Listing 2

Im konkreten Fall hat der Autor mit dieser Abfrage seit mehreren Jahren keinen Wartungsaufwand mehr, denn eine kleine Apex-Anwendungsseite erlaubt der Fachseite die Pflege der Kriterien-Tabelle. Ein kleines Gimmick: Auf dieser Seite kann die Fachseite für die vier Kriterien Beispielwerte einfügen und sich live berechnen lassen, wie die hinterlegte Parametrierung diese Kombination bewertet. Auf diese Weise ist eine einfache Kontrolle möglich, es müssen lediglich die eingefügten Werte im „f.k1" bis „f.k4" zur Verfügung gestellt und die Abfrage ausgeführt werden.

Es gibt aber auch noch eine große Anwendung dieses Prinzips. Diese Idee entstand, als der Autor plante, ein Regelwerk in der Datenbank einzurichten, um Geschäftsregeln dynamisch vorzuhalten. Ihm war klar, dass er auf der Anwendungsseite einen Editor benötigen würde, der verschiedene logische Operatoren, UND- und ODER-Verknüpfungen, Klammern und so weiter implementieren müsste. Die Idee ist angesichts der geschilderten Lösungen dieses Artikels geradezu banal, aber in der Projektsituation muss man eben auch darauf kommen: Wir haben bereits eine Regelauswertungsmaschine, deren Syntax alle Entwickler sicher beherrschen, denn die Regeln stellen nichts anderes als Teile einer WHERE-Klausel einer SQL-Abfrage dar.

Wenn man die Messwerte, die man in Regeln auswerten muss, in der SELECT-Klausel zur Verfügung stellt (eventuell einfach als Variablen, die man gegen DUAL abfragt), lassen sich die Regeln in der WHERE-Bedingung auswerten. Die Regeln selbst sind in einer Tabelle gespeichert und dynamisch zu der Regelabfrage in *Listing 2* zusammengestellt. Den Code-Generator dafür gibt es ja bereits …

Die Idee: Eine Tabelle „SCT_BL_RULES“ sammelt alle Regeln, die für eine gegebene Situation gelten sollen. Alle relevanten Messwerte (hier aus einem Session State einer Apex-Anwendung) werden über eine View gegen „DUAL“ in der WITH-Klausel zur Verfügung gestellt. Die einzelnen Regelbedingungen kommen aus der Regeltabelle und sind unter der Regel-ID in die WHERE-Klausel gewandert („<Regel_ID> = ID and <Regelbedingung>“). Die Regeln verfügen über ein Sortierkriterium, das anzeigt, welche Regel vorrangig vor einer anderen Regel ausgeführt werden soll, falls mehrere Regeln zu „TRUE“ evaluieren. Die SQL-Abfrage ermittelt alle Regeln, deren Bedingung erfüllt sind und wählt die relevanteste Regel aus.

Damit besteht ein unendlich leistungsfähiges Instrumentarium zur Formulierung der Regeln – und das alles ohne Programmieraufwand. Noch besser ist es, wenn man die Regeln selbst in einer Tabelle speichert und dynamisch zu einer Regelabfrage zusammenstellt. Auch hier gilt also wieder einmal: Wer SQL die Arbeit machen lässt, hat nur noch die Hälfte des Programmieraufwands.

Jürgen Sieben
j.sieben@condes.de