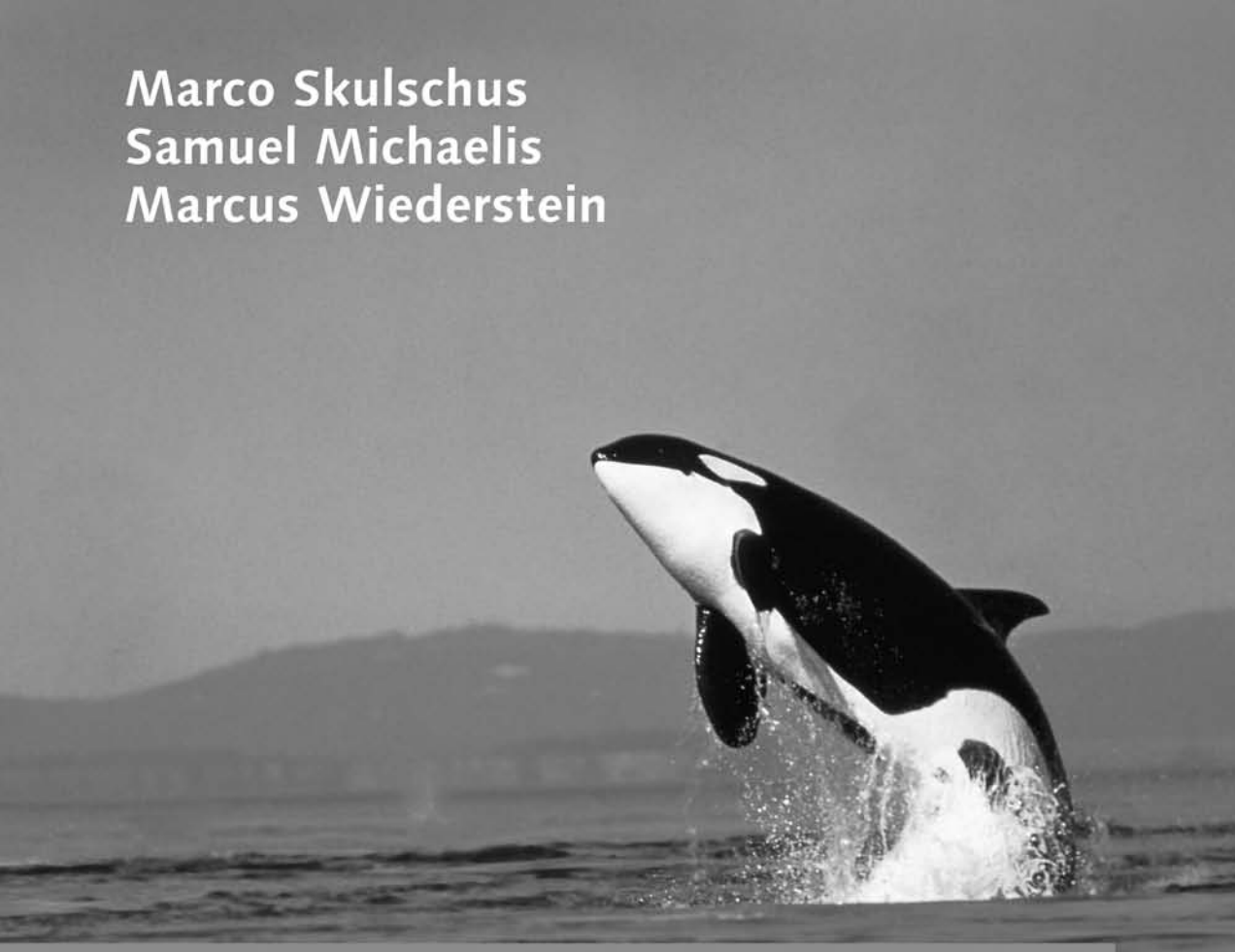
Marco Skulschus
Samuel Michaelis
Marcus Wiederstein



Oracle 10g

Programmierhandbuch

Galileo Computing

Auf einen Blick

Inhalt

```
  Steigung Datenmenge y-Schnittpunkt Bestimmtheitsmaß
---------- ---------- -------------- ----------------
  206701,1          4     -413189419        ,580345963
y-Durchschnitt x-Durchschnitt
-------------- --------------
      522832,25         2001,5
1 Zeile wurde ausgewählt.
```

3.5 Analytische bzw. Data-Warehouse-Abfragen

Dieser Abschnitt führt nach der Vorstellung der eingebauten SQL-Funktionen wieder zu den Abfragen zurück, wobei dieses Mal keine eigentlichen Standard-Abfragen behandelt werden, sondern das, was man unter fortgeschrittenen Abfragetechniken versteht. Dies bezieht sich zum einen auf die Tatsache, dass nur große Datenbanksysteme die gleich folgenden Erweiterungen und Konzepte unterstützen. Dies bezieht sich aber auch zum anderen darauf, dass mit Blick auf die im zweiten Abschnitt dargestellten analytischen Möglichkeiten verschiedene Oracle-Spezifika zu Sprache kommen, die daher nicht in anderen Datenbanksystemen so einsetzbar sind. Dort mag es andere, ähnliche Konstruktionen geben, oder eventuell auch gar keinen Ersatz, sodass man die Analyseschritte selbst durchführen oder andere Hilfswerkzeuge zum Einsatz bringen muss, um die gleichen Ergebnisse zu erhalten.

3.5.1 Erweiterte Gruppierungen

In den vorangegangenen Abschnitten haben wir bereits mehrfach unterschiedliche Bezeichnungen für die drei Erweiterungen `GROUPING SETS`, `CUBE` und `ROLLUP` verwendet. Sie alle stellen Erweiterungen und zusätzliche Klauseln dar, die auf dem bereits bekannten `GROUP BY`-Befehl aufbauen, verschiedene Berechnungen vereinfachen und eher Berichte statt Abfrageergebnisse ausgeben. Da diese Klauseln mit dem Begriff der Dimension arbeiten und die Ergebnisse auch in Form von Dimensionen betrachtet werden können, bezeichnet man diese Abfrageklauseln auch als *Data-Warehouse-Funktionen*. Dies hat nichts damit zu tun, dass eine vorhandene Datenbank als Data Warehouse mit seinen spezifischen Eigenarten betrieben werden muss. Vielmehr sind diese Abfragen dort besonders häufig und werden mit weiteren Methoden und Abfragetechniken kombiniert, um tatsächlich aus dem gesammelten Datenmaterial neues, meist auch strategisches Wissen für eine Organisation herauszufiltern. Man kann diese Funktionen allerdings auch ohne ein Data Warehouse in Abfragen einsetzen, weswegen wir mit diesem Thema auch das Abfrage- und Analyse-Kapitel beenden wollen.

Sinn und Zweck von erweiterten Gruppierungen

Wie gerade schon gesagt wurde, handelt es sich bei diesen Klauseln um keine gewöhnlichen weiteren Einschränkungen, sondern um Möglichkeiten, komplexe Berichte zu erzeugen. Ihnen fehlt natürlich ein nettes Format und eine farbenfrohe Darstellung, doch vom eigentlichen Inhalt her sind sie durchaus ausgereift genug, um als Bericht gelten zu können. Dies bedeutet, dass keine relationalen Abfragemengen entstehen, sondern Datenstrukturen mit einer fallweisen Logik, die nur schwer in allgemeinen Programmen zu verwerten ist. Dies ist einer der Gründe, warum auch verschiedene Autoren und Verfechter eines strengeren relationalen Ansatzes in Datenbanken diese gesamte Konstruktion ablehnen. Wir möchten uns dieser Meinung nicht anschließen, müssen allerdings darauf hinweisen, dass man mit den Ergebnissen definitiv die bisher getätigten Abfragen vollkommen verlässt.

Die Problematik und die Besonderheit dieser Abfragen versteht man am ehesten mit dem Begriff der Dimension. Dieser ist einer der zentralen Begriffe des Data-Warehouse-Konzepts wie auch dieser Erweiterungen zu `GROUP BY`, weil sie ebenfalls Ergebnisse in Dimensionen abbilden. Eine Dimension stellt dabei in Reinsprache eine Datenstruktur dar, die man bereits bei der Vorstellung des einfachen `GROUP BY` mit dem kleinen Partikel »pro« ableiten konnte. Möchte man z. B. die Anzahl der Kurse pro Bereich haben, so erhält man in einer einzigen Dimension eine Tabelle mit einer Liste aller Kursbereiche und einer Liste von Kurszahlen, die in diesen einzelnen Bereichen vorhanden sind. Möchte man diese Abfrage erweitern und eine weitere Dimension aufspannen, wie z. B. die Dimension »Teilnehmerzahlen pro Bereich«, so erhält man eine weitere Spalte mit den zugehörigen Daten. Es ist nun allerdings nicht mehr möglich, die Dimensionen und Datenstrukturen wie die Anzahl aller Kurse und die Anzahl aller Teilnehmer in einer Tabelle sinnvoll unterzubringen. Dies ist keine Schande für das relationale Modell, weil es nicht für Berichte und derartige Analysen entwickelt worden ist, sondern andere Ansätze verfolgt.

```
SELECT K_Bereich                 AS "Bereich",
       COUNT(DISTINCT kurs.K_Nr) AS "Anzahl Kurse",
       COUNT(B_Nr)               AS "Anzahl Buchungen"
  FROM kurs INNER JOIN termin
    ON kurs.K_Nr = termin.K_Nr
          INNER JOIN buchung
    ON termin.T_Nr = buchung.T_Nr
 GROUP BY K_Bereich;
```

Listing 3.138 351_01.sql: Einfache Übersicht in zwei Dimensionen

Wie nicht anders zu erwarten, erhält man genau das gewünschte Ergebnis, da beide Dimensionen sich auf die gleichen Größe – nämlich den Kursbereich – beziehen.

```
Bereich                        Anzahl Kurse Anzahl Buchungen
------------------------------ ------------ ---------------
Datenbanken                              13             557
Grafik                                    8             384
Office                                    6             146
Programmierung                           41            1341
Programmierung(Office)                    3              76
Server                                    3              91
Webdesign                                 5             240
7 Zeilen ausgewählt.
```

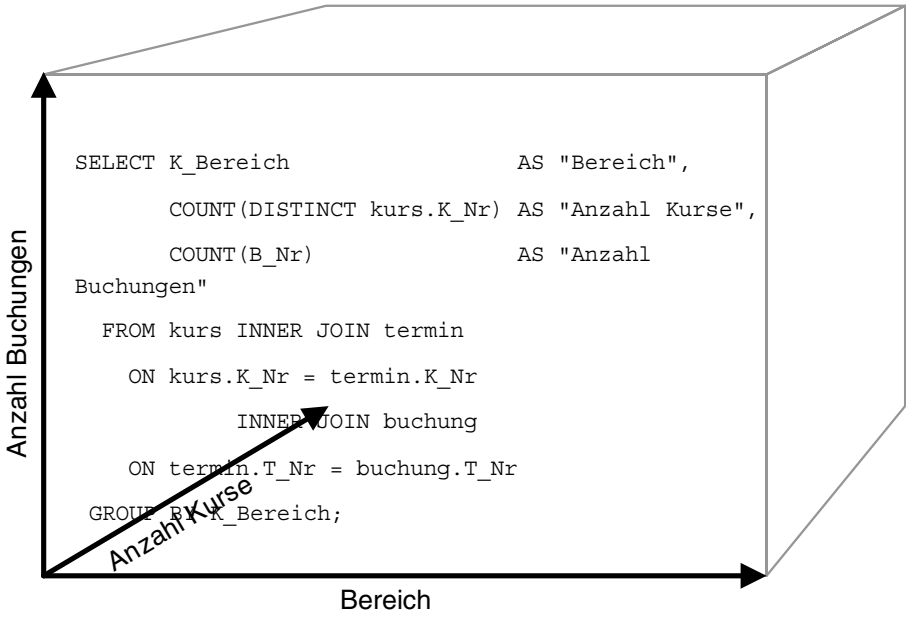


Abbildung 3.18 Würfel mit drei Dimensionen

Erweitert man diese Abfrage um eine Zeile, in der man die Dimension der Kursbereiche um die Titel ergänzt, so erhält man diese natürlich in einer wesentlich längeren Ergebnismenge. Allerdings werden die Gruppierungen in diesem Fall nur noch für die Titel ausgeführt und nicht mehr für die Bereiche. Diese erscheinen natürlich noch in der Ergebnismenge, allerdings nur noch als weitere Information, zu welchem Bereich der entsprechende Titel gehört. Dies ist relational völlig korrekt, aber insoweit schade, als dass man nun die Anzahl

der Kurse pro Bereich wie auch die Anzahl der Buchungen pro Bereich nicht mehr in einer einzigen Ergebnismenge erhält.

```
Bereich         Titel          Anzahl Kurse Anzahl Buchungen
--------------- -------------- ------------ ---------------
Grafik          Flash                     1              97
Grafik          Director                  1               6
Grafik          Freehand                  1              38
Grafik          Fireworks                 1             103
```

Listing 3.139 351_02.sql: Ergebnis einer Unterdimension

Mit Hilfe der `GROUP-BY`-Erweiterungen ist es jedoch möglich, sowohl die oben ermittelten Werte »pro Bereich« als auch »pro Titel« innerhalb desselben Ergebnisses zu erhalten. Dabei entstehen nämlich an passenden Stellen `NULL`-Felder. Dies ist letztlich die einzige Funktion aller drei Erweiterungen, die in den folgenden Abschnitten vorgestellt werden. Lediglich die Definition und Bestimmung der einzelnen Dimensionen und Aggregationsstufen wird durch eine unterschiedliche Syntax eingerichtet.

Zum letzten Mal: Es sind übrigens die `NULL`-Felder, die die Ergebnismenge ihres relationalen Charakters berauben, da hier zwar unbekannte Werte erzeugt werden, die auch durch keinen Inhalt zu füllen wären. Ihre Existenz verdanken sie ausschließlich dem Format oder dem Aussehen des entstehenden Berichts, sie lassen sich aber nicht nach 1 × 1 des relationalen Datenmodells erklären. Problematisch wird dies besonders, wenn man solche Mengen unbesehen verarbeiten möchte und daher wissen muss, an welchen Stellen `NULL`-Felder aus diesen Gründen entstehen, wo interessante Werte stecken, welche inhaltliche Aussage diese Datenstrukturen beinhalten und in welche Bereiche der Bericht insgesamt eingeteilt ist.

GROUPING SETS

Die Lösung des oben gestellten Problems lässt sich ganz einfach über die explizite Angabe der Achsen bzw. der Dimensionen lösen, die benötigt werden. Der Würfel ist neben der Dimension ein weiterer konzeptioneller Begriff im Data-Warehouse-Konzept. Auch wenn er bei multidimensionalen Abfragen regelmäßig nicht mehr grafisch, sondern nur noch strukturell zu erfassen ist, da mehr als drei Dimensionen zum Einsatz kommen, hilft er dabei, erweiterte Abfragen zu planen. Mit Hilfe der `GROUPING SETS` bestimmt man die zu verwendenden Achsen selbst und explizit, während beim später vorgestellten `CUBE` sämtliche Dimensionen erzeugt werden. Dies ist nicht immer wünschenswert, da dementsprechend mehr Daten entstehen, die man eventuell gar nicht benötigt.

Für das obige Problem, in dem man neben den Werten für die Titel auf einer niedrigeren Aggregationsstufe auch noch die Titel mit ihren Kurs- und Buchungszahlen benötigte, lässt sich diese Achse mit `(K_Bereich, K_Titel)` angeben. Dies schließt man an die `GROUP BY`-Klausel mit Hilfe der `GROUPING SETS`-Klausel an.

```
SELECT SUBSTR(K_Bereich,1,12)    AS "Bereich",
       SUBSTR(K_Titel,1,12)      AS "Titel",
       COUNT(DISTINCT kurs.K_Nr) AS "Anzahl Kurse",
       COUNT(B_Nr)               AS "Anzahl Buchungen"
  FROM kurs INNER JOIN termin
    ON kurs.K_Nr = termin.K_Nr
          INNER JOIN buchung
    ON termin.T_Nr = buchung.T_Nr
 GROUP BY GROUPING SETS (K_Bereich, K_Titel);
```

Listing 3.140 351_03.sql: Verschiedene Aggregationsstufen

Man erhält – wie oben gewünscht und indirekt angekündigt – eine berichtähnliche Ergebnismenge zurück, in der an oberster Stelle die Bereichsliste mit den zugehörigen Werten steht. Sie befinden sich deswegen an oberster Stelle, da dies ja auch so in der Achsendefinition zuvor vorgegeben wurde. Das komprimierte Ergebnis hat also prinzipiell folgende Gestalt:

```
Bereich      Titel        Anzahl Kurse Anzahl Buchungen
------------ ------------ ------------ ----------------
Datenbanken                         13              557
Server                               3               91
Webdesign                            5              240
             ASP.NET                 1               30
             Access                  3              133
```

Verwendung und Formulierung

Die `GROUPING SETS`-Klausel ist dann einzusetzen, wenn nicht alle Dimensionsachsen oder Bezüge des entstehenden Würfels benötigt werden oder wenn man genau bestimmen möchte, welche Bezüge berechnet werden sollen. Dabei verwendet man so genannte zusammengesetzte Spalten, die den jeweiligen Bezug darstellen. Im Vergleich zu den anderen Klauseln gibt es immer Möglichkeiten, entweder die eine oder die andere syntaktische Formulierung zu wählen, um zum gleichen Ergebnis zu gelangen.

GROUPING SETS	Alternative
`GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())`	`CUBE(a, b, c)`
`GROUPING SETS ((a, b, c), (a, b), ())`	`ROLLUP(a, b, c)`

Tabelle 3.6 Vergleich mit anderen Erweiterungen

Die Klausel `GROUPING SETS` lässt sich auch über andere syntaktische Mittel einrichten wie z. B. über `UNION ALL` berechnete Abfragen, wie die folgende Tabelle zeigt.

GROUPING SETS	Alternative
`GROUP BY GROUPING SETS(a, b, c)`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c`
`GROUP BY GROUPING SETS(a, b, (b, c))`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c`
`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`
`GROUP BY GROUPING SETS(a, (b), ())`	`GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()`
`GROUP BY GROUPING SETS(a, ROLLUP(b, c))`	`GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)`

Tabelle 3.7 Vergleich der GROUPING SETS mit GROUP BY

Zusätzlich ist es möglich, mehrere `GROUPING SET`-Klauseln zu kombinieren, um so Kreuzverknüpfungen einzurichten, bei denen jede Dimension mit jeder anderen zusätzlich kombiniert wird. Dies bedeutet, dass auch `GROUP BY GROUPING SETS (a, b), GROUPING SETS (c, d)` die zur Form `GROUP BY GROUPING SETS (a, c), (a, d), (b, c), (b, d)` wird.

Im folgenden Beispiel untersucht man die Summe bzw. den Umsatz, der durch die Kursanmeldungen hervorgerufen wurde, pro Ort und Bereich mit den Bezügen Jahr und Monat. Daher erhält man insgesamt vier Achsen, die man als zusammengesetzte Spalten mit jeweils zwei Ausdrücken in die `GROUPING SETS`-Klausel platziert. Sie sehen bereits wieder, dass es auch im Zusammenhang mit dieser Klausel sowie mit den anderen Erweiterungen stets darum geht, gültige Ausdrücke zu finden. Um also den Monat ausfindig zu

machen, kann man entweder die TO_CHAR- oder EXTRACT-Funktion verwenden, die genauso in der bisherigen Abfrage erscheinen kann wie in den zusammengesetzten Spalten in der GROUPING SETS-Klausel.

```
SELECT SUBSTR(T_Ort,1,10)        AS Ort,
       TO_CHAR(T_Beginn, 'YYYY') AS Jahr,
       TO_CHAR(T_Beginn, 'MM')   AS Monat,
       SUBSTR(K_Bereich,1,10)    AS Bereich,
       SUM(B_Preis)
  FROM termin NATURAL JOIN buchung
              NATURAL JOIN kurs
  GROUP BY GROUPING SETS
              ((TO_CHAR(T_Beginn, 'YYYY'), T_Ort),
               (TO_CHAR(T_Beginn, 'YYYY'), K_Bereich),
               (TO_CHAR(T_Beginn, 'MM'), T_Ort),
               (TO_CHAR(T_Beginn, 'MM'), K_Bereich),
               (T_Ort, K_Bereich));
```

Listing 3.141 351_04.sql: Mehrere Dimensionen

Man erhält ein umfangreiches Ergebnis, das weniger seiner Werte wegen für uns interessant ist, sondern wegen der Formatierung derselben:

- Die Reihenfolge der Spalten in der Spaltenliste ändert sich gegenüber den bisherigen Abfragen nicht. Lediglich Felder, die keine passenden Werte in der gewählten Dimension ausgeben können, bleiben als Novum leer.

```
ORT        JAHR MO BEREICH    SUM(B_PREIS)
---------- ---- -- ---------- ------------
```

- Die erste Dimension ist Ort und Jahr, wie in der GROUPING SETS-Klausel auch angegeben. Daher erhält man zunächst eine Liste aller Städte pro Jahr mit ihrem Umsatz. Dies bedeutet auch, dass man einige Städte doppelt erhält, wenn sie in verschiedenen Jahren Umsatz erwirtschafteten:

```
Bochum     2000                    413543
Bochum     2001                    209712
```

- Die zweite Dimension untersucht den Umsatz pro Bereich pro Jahr, sodass also hier auch doppelte Zahlwerte und doppelte Bereichsnamen erscheinen können. Innerhalb dieser Gruppen jedoch bilden sie immer ein Paar, da nur das Aggregat ausgegeben wird: pro Jahr und pro Bereich.

```
           2000    Grafik           53070
           2000    Office           13210
```

- Die dritte Dimension ist der Umsatz pro Stadt pro Monat mit doppelten Nennungen der Stadtnamen, wenn in diesen Städten in verschiedenen Monaten Umsatz erwirtschaftet wurde. Hierbei bleiben die Jahreszahlen unberücksichtigt, d.h., man erhält nur die Werte für alle Geschäftsjahre als Summe:

```
Köln            09                   5742
Köln            11                  11264
Berlin          02                   7404
```

- Die vierte Dimension ist der Umsatz pro Monat pro Bereich, wobei auch hier für alle Geschäftsjahre ein Gesamtumsatz pro Monat ausgegeben wird.

```
                10 Office            4110
                01 Server           17066
```

- Die fünfte Dimension ist schließlich der Umsatz pro Stadt pro Bereich mit einer Summenbildung für alle Jahre.

```
Köln              Programmie        26964
Berlin            Datenbanke        17984
Bochum            Grafik           234144
```

Formulierung von zusammengesetzten Spalten

Als zusammengesetzte Spalten lassen sich alle gültigen Spaltenausdrücke verwenden. Dies bezieht sowohl die Zahl der für eine Dimension benötigten Spalten ein als auch Spaltenfunktionen, die z.B. Werte aus den gespeicherten Daten extrahieren.

Im folgenden ersten Beispiel finden Sie eine Variation der obigen Abfrage, wobei dieses Mal nicht nur Dimensionen aus zwei Spalten gebildet werden, sondern auch übersichtliche Angaben zu einer einzigen Spalte (wie z.B. für den Umsatz pro Ort und für den Umsatz pro Bereich) oder zu drei Spalten (wie z.B. für den Umsatz pro Ort pro Bereich pro Jahr) verlangt werden.

```
SELECT SUBSTR(EXTRACT(YEAR FROM T_Beginn),1,5) AS Jahr,
       SUBSTR(T_Ort,1,10)                      AS Ort,
       SUBSTR(K_Bereich,1,20)                  AS Bereich,
       TO_CHAR(SUM(B_Preis), '9G999G999U')     AS Summe
  FROM termin NATURAL JOIN buchung
              NATURAL JOIN kurs
  GROUP BY GROUPING SETS (
            (T_Ort, K_Bereich, EXTRACT(YEAR FROM T_Beginn)),
            (K_Bereich, EXTRACT(YEAR FROM T_Beginn)),
```

```
(T_Ort),
(K_Bereich),
(EXTRACT(YEAR FROM T_Beginn)));
```

Listing 3.142 351_05.sql: Typen von zusammengesetzten Spalten

Dies erzeugt strukturell die folgende Ausgabe, wobei wir das Original sehr verkürzt haben und aus jeder Dimension zwei Reihen abdrucken:

```
JAHR  ORT        BEREICH               SUMME
----- ---------- -------------------- --------------------
2000  Bochum     Grafik                     53.070_
2001  Bochum     Grafik                     25.509_
2000             Grafik                     53.070_
2001             Grafik                     25.509_
                 Programmierung          1.138.180_
                 Programmierung(Offic       47.994_
      Berlin                                17.984_
      Bochum                             1.778.365_
2000                                       413.543_
2001                                       209.712_
```

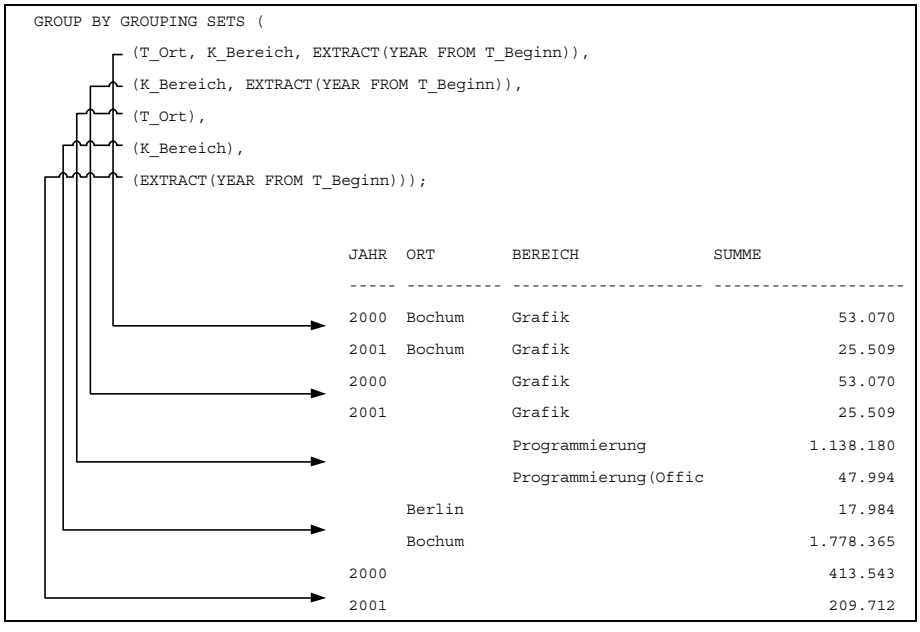


Abbildung 3.19 Dimensionsbestimmung bei GROUPING SETS

Die Spaltenliste kann nicht nur einzelne Spaltennennungen enthalten, sondern auch komplexe Strukturen in Form von durch einen Klammerausdruck kombinierten Spalten. Im folgenden Beispiel erzeugt man aus nur zwei in der Spaltenliste definierten Spalten drei Dimensionen, indem in der `GROUPING SETS`-Klausel die beiden Spalten `K_Bereich` und `T_Beginn` als eine Spalte zusammengefasst werden. So erhält man die drei Dimensionen Teilnehmerzahl *pro Bereich*, *pro Monat* und durch die Zusammenfassung auch *pro Bereich pro Monat*.

```
SELECT TO_CHAR(T_Beginn, 'MM') AS Monat,
       K_Bereich AS Bereich,
       COUNT(B_Nr) AS Zahl
  FROM buchung INNER JOIN termin
    ON buchung.T_Nr = termin.T_Nr
               INNER JOIN kurs
    ON termin.K_Nr = kurs.K_Nr
 GROUP BY GROUPING SETS(TO_CHAR(T_Beginn,'MM'),K_Bereich,
                        (TO_CHAR(T_Beginn,'MM'),K_Bereich));
```

Listing 3.143 351_06.sql: Komplexe Spaltenstrukturen

Diese Struktur ist gleichbedeutend und damit alternativ zu folgender Formulierung, in der die drei Dimensionen direkt bestimmt werden und auch unmittelbar als drei erkennbar sind:

```
GROUP BY GROUPING SETS (TO_CHAR(T_Beginn, 'MM'),
                        (K_Bereich),
                        (TO_CHAR(T_Beginn,'MM'),K_Bereich));
```

Wie oben erwähnt, erhält man drei Dimensionen:

```
MO BEREICH                              ZAHL
-- ------------------------------ ----------
01 Grafik                                4
03 Grafik                               42
   Webdesign                           240
   Datenbanken                         557
06                                     275
07                                     193
```

Es kann allerdings auch Situationen geben, in denen diese Kombination notwendig wird, um unerwünschte Ergebnisse zu vermeiden. Unerwünscht kann ein Ergebnis dabei nicht nur deswegen sein, weil man seine Informationen nicht benötigt, sondern weil es auch keine sinnvollen Informationen liefert. In

der nächsten, sehr umfangreichen Abfrage ermittelt man zunächst die Anzahl der Teilnehmer pro Unternehmen und pro Bereich, den Umatz und die Anzahl der gebuchten Termine. Durch die Abfragestruktur muss man im GROUP BY-Ausdruck in jedem Fall die Unternehmensnummer und den Beginn gruppieren. Dies führt aber zu unsinnigen bzw. nicht verwertbaren Informationen, da dies als Dimensionsangabe verstanden wird.

```
SELECT SUBSTR(EXTRACT(YEAR FROM T_Beginn),1,5) AS Jahr,
       SUBSTR(U_Name, 1, 15) AS Unternehmen,
       SUBSTR(K_Bereich,1,5) AS Bereich,
       (SELECT COUNT(buchung.TN_Nr)
          FROM buchung INNER JOIN teilnehmer
            ON buchung.TN_Nr = teilnehmer.TN_Nr
                       INNER JOIN unternehmen b
            ON teilnehmer.U_Nr = b.U_Nr
         WHERE b.U_Nr = a.U_Nr
         GROUP BY U_Nr) AS TNZahl,
       (SELECT COUNT(DISTINCT T_Nr)
          FROM buchung INNER JOIN teilnehmer
            ON buchung.TN_Nr = teilnehmer.TN_Nr
                       INNER JOIN unternehmen b
            ON teilnehmer.U_Nr = b.U_Nr
         WHERE b.U_Nr = a.U_Nr
         GROUP BY U_Nr)            AS Termine,
       SUM(B_Preis)                AS Summe
  FROM buchung INNER JOIN teilnehmer
    ON buchung.TN_Nr = teilnehmer.TN_Nr
               INNER JOIN unternehmen a
    ON teilnehmer.U_Nr = a.U_Nr
               INNER JOIN termin
    ON buchung.T_Nr = termin.T_Nr
               INNER JOIN kurs
    ON termin.K_Nr = kurs.K_Nr
  GROUP BY GROUPING SETS (U_Name, K_Bereich,
                          (U_Nr,EXTRACT(YEAR FROM T_Beginn)))
  HAVING SUM(B_Preis) > 15000;
```

Listing 3.144 351_07.sql: Notwendige Verknüpfung von Spalten

Die so formulierte Abfrage liefert folgendes Ergebnis in den drei angegebenen Dimensionen, die auch sinnvolle Datenstrukturen darstellen:

```
JAHR  UNTERNEHMEN     BEREI     TNZAHL    TERMINE      SUMME
----- --------------- ----- ---------- ---------- ----------
2002                                31          6      16536
2003                                30          5      19392
                      Serve                              69856
                      Webde                             132856
       KVZ Bank Essen                                    15150
      Bildungsforum g                                    16140
```

Verzichtet man auf die oben angegebene Klammerung der beiden notwendigerweise zu gruppierenden Spalten `Unternehmensnummer` und `Beginn`, erhält man diese beiden Spalten auch als Dimension mit folgenden wertlosen Informationen:

```
JAHR  UNTERNEHMEN     BEREI     TNZAHL    TERMINE      SUMME
----- --------------- ----- ---------- ---------- ----------
                                    30          5      24030
                                    24          4      20748
                                    24          4      19026
```

Gruppensuchbedingungen und Sortierungen

Da es sich bei allen drei Klauseln um Erweiterungen der `GROUP BY`-Klausel handelt, kann man wie bisher Gruppensuchbedingungen und Sortierungen an diese Klausel anschließen. Dabei gilt die Gruppensuchbedingung nicht für die Datenverarbeitung, sondern ausschließlich für die Anzeigen. Dies zeigt folgendes Beispiel, in dem die Umsätze pro Unternehmen anhand des Unternehmensnamens und des Bereichsnamens ausgegeben und ermittelt werden. Dabei sollen nur die Datensätze ausgegeben werden, die größer als 5 000 sind. Mit der `LIKE`-Bedingung wird die Funktionsweise anhand eines speziell ausgewählten Unternehmens gezeigt.

```
SELECT SUBSTR(U_Name,1,25)    AS Unternehmen,
       SUBSTR(K_Bereich,1,11) AS Bereich,
       SUM(B_Preis)           AS Summe
  FROM buchung NATURAL JOIN teilnehmer
               NATURAL JOIN unternehmen
               NATURAL JOIN termin
               NATURAL JOIN kurs
  GROUP BY GROUPING SETS ((U_Name, K_Bereich),
                          (U_Name))
  HAVING SUM(B_Preis) > 5000
```

```
      OR U_Name LIKE '5D%'
  ORDER BY U_Name, K_Bereich;
```

Listing 3.145 351_09.sql: Gruppensuchbedingungen

Man erhält im Ergebnis wie sonst auch die Summen pro Unternehmen, sofern sie die 5 000-Euro-Grenze überschreiten, und auch sämtliche Datensätze für das über die LIKE-Bedingung ausgewählte Unternehmen. Im Vergleich zum zweiten abgedruckten Unternehmen sieht man, dass die Gesamtsumme sich auf alle Bereiche bezieht, wenn auch nicht alle Datensätze für das zweite Unternehmen ausgegeben werden. Es hat 8 920 Euro umgesetzt, wovon 6 500 auf den Bereich Programmierung entfallen und die übrigen 2 420 auf einen anderen Bereich, der nicht ausgegeben werden kann, da er 5 000 Euro nicht übersteigt.

```
5D Trading Computer + Com Datenbanken       8520
5D Trading Computer + Com Webdesign         3450
5D Trading Computer + Com                  11970
90 Products Analysentechn Programmier       6500
90 Products Analysentechn                   8920
```

ROLLUP

Für die Erzeugung von Zwischensummen lassen sich vollständige oder auch teilweise Spaltengruppen definieren, die mit der Erweiterung ROLLUP als Summenspalten deklariert werden.

Verwendung für Spaltensummen

Der Einsatz von ROLLUP ist überaus einfach, da seine einzige Funktion nur in der Erzeugung von Spaltensummen besteht. Diese Klausel schließt man ebenfalls an den GROUP BY-Befehl an und nennt in einem Klammerausdruck die zu summierenden Spalten. Dabei werden die gewöhnlichen, auch von einem einfachen GROUP BY-Befehl erzeugten Spalten und Ergebnisse erzeugt, ergänzt um Unterspaltensummen und eine große Spaltensumme am Ende der gesamten Tabelle. Die folgende Abfrage zählt die Umsätze pro Bereich und ermittelt für die einzelnen Jahre jeweils eine Jahressumme und für die gesamte Tabelle eine Gesamtsumme.

```
SELECT K_Bereich,
       TO_CHAR(T_Beginn, 'YYYY') AS Jahr,
       SUM(B_Preis)
  FROM teilnehmer INNER JOIN buchung
    ON buchung.TN_Nr = teilnehmer.TN_Nr
```

```
                INNER JOIN termin
    ON buchung.T_Nr = termin.T_Nr
                INNER JOIN kurs
    ON kurs.K_Nr = termin.K_Nr
  GROUP BY ROLLUP (TO_CHAR(T_Beginn, 'YYYY'), K_Bereich);
```

Listing 3.146 351_10.sql: Erzeugung von Spaltensummen

Das Ergebnis erhält die angekündigten Summen für jedes Jahr mit ausgewiesener Jahreszahl und einer Gesamtsumme am Ende der Tabelle.

```
K_BEREICH                      JAHR SUM(B_PREIS)
------------------------------ ---- ------------
Datenbanken                    2003       213220
Programmierung                 2003       566631
Programmierung(Office)         2003        17811
                               2003      1024639
                                         2091329
```

Vollständige und teilweise Summenbildung

Die Art und Weise der Summierung kann man beeinflussen, indem man die Spalten, die gruppiert werden müssen, weil man für sie Daten zählt, und die Spalten, die für die Summenbildung herangezogen werden sollen, getrennt erfasst. Im folgenden Beispiel erzeugt man was für den Ort und das Jahr sowie die gewöhnlichen Summen aus dem `GROUP BY`-Befehl in Kombination mit der `ROLLUP`-Klausel. resultiert. Allgemein gesagt: Bei einer Formulierung wie `GROUP BY spalte1, ROLLUP(spalte2, spalte3)` erhält man Summen für die Bereiche `(spalte1, spalte2, spalte3)`, `(spalte1, spalte2)` und `(spalte1)`.

```
SELECT TO_CHAR(T_Beginn, 'YYYY') AS Zeit,
       SUBSTR(T_Ort,1,10)        AS Ort,
       SUBSTR(K_Bereich,1,10)    AS Bereich,
       COUNT(DISTINCT T_Nr)      AS Termine,
       SUM(B_Preis)              AS Summe
  FROM termin NATURAL JOIN buchung
              NATURAL JOIN kurs
  GROUP BY TO_CHAR(T_Beginn, 'YYYY'),
           ROLLUP(T_Ort, K_Bereich);
```

Listing 3.147 351_11.sql: Teilweise Summenbildung

Man erhält die angekündigten Summen:

```
ZEIT ORT        BEREICH       TERMINE      SUMME
---- ---------- ---------- ---------- ----------
2003 Münster    Datenbanke          2       7536
2003 Münster    Programmie          5      24891
2000 Bochum                       121     413543
2001 Bochum                        59     209712
2000                              121     413543
2001                               59     209712
```

CUBE

Die `CUBE`-Erweiterung ist im Vergleich zur `GROUPING SETS`-Klausel eine große Vereinfachung, da sie alle möglichen Dimensionen automatisch erzeugt. Dies lässt schon ihr Name erkennen: Sie berechnet den größtmöglichen Würfel, wohingegen bei der `GROUPING SETS`-Klausel die einzelnen Achsen extra definiert werden müssen. Der einzige Nachteil besteht darin, dass damit in den meisten Fällen auch Datenstrukturen generiert werden, die nicht benötigt werden.

```
SELECT SUBSTR(K_Bereich,1,12)    AS "Bereich",
       SUBSTR(K_Titel,1,12)      AS "Titel",
       COUNT(DISTINCT kurs.K_Nr) AS "Anzahl Kurse",
       COUNT(B_Nr)               AS "Anzahl Buchungen"
  FROM kurs INNER JOIN termin
    ON kurs.K_Nr = termin.K_Nr
          INNER JOIN buchung
    ON termin.T_Nr = buchung.T_Nr
 GROUP BY CUBE (K_Bereich, K_Titel);
```

Listing 3.148 351_12.sql: Erzeugung eines Würfels

Man erhält im Ergebnis sämtliche verfügbaren Dimensionen:

- Die Spaltenliste wird aus der in der Abfrage erzeugten Spaltenliste übernommen.
- Dimension Anzahl Buchungen, Anzahl Kurse pro Titel pro Bereich:

```
Bereich      Titel        Anzahl Kurse Anzahl Buchungen
------------ ------------ ------------ ----------------
Datenbanken  Access                  3              133
Datenbanken  MS SQL-Serve            2               21
```

- Dimension Anzahl Buchungen, Anzahl Kurse pro Bereich:

```
Datenbanken                    13              557
Grafik                          8              384
```

- Dimension Anzahl Buchungen, Anzahl Kurse pro Titel:

```
                ASP.NET         1               30
                Access          3              133
```

- Spaltensummen für Anzahl der Kurse und Anzahl der Buchungen:

```
                                79             2835
```

GROUPING-Funktionen

Ganz zu Anfang haben wir darauf hingewiesen, dass man sich gewaltige Wortgefechte darüber liefern kann, ob die vorgestellten Erweiterungen Verschlimmbesserungen beim Einsatz des relationalen Modells darstellen oder ob es sich um nützliche Konstruktionen handelt. Die Grundproblematik besteht in der Vermischung einer Formatierung, die notwendigerweise beim Einsatz von Unter- und Gesamtsummen sowie bei der Dimensionsbildung entsteht, mit den eigentlichen Daten, wie sie direkt aus einer Berechnung und aus den Tabellenwerten entnommen werden. Möchte man nun solche Datenstrukturen verarbeiten, bei denen – im Gegensatz zum relationalen Modell, bei dem nur die horizontale Spaltenverteilung inhaltlich bedeutsam ist – auch die vertikale Position eine Bedeutung hat, muss man eine Möglichkeit haben herauszufinden, welche Zeile zu welcher Struktur gehört. Diese Problematik wird durch unterschiedliche Zusatzfunktionen gemildert, die als Spaltenfunktionen verwendet werden können. Sie liefern sehr einfache Informationen, an denen der Zeilentyp ermittelt werden kann.

Aggregationsstufen und Untersummen bei GROUPING

Mit `GROUPING` werden die ärgsten Probleme bei der Verwendung mit den von `GROUP BY`-Erweiterungen auf Zeilenebene gelöst. Diese Funktion wird als Spaltenfunktion für die Dimensionsspalten genutzt, wobei sie nicht bei der einfachen Verwendung der Spalte zusätzlich zum Tragen kommt, sondern in einem erneuten Aufruf jeder Dimensionsspalte eingerichtet wird. Dies bedeutet für das nächste Beispiel, das eine Variation des vorherigen darstellt, dass die beiden Dimensionen `Bereich` und `Titel` noch einmal mit dieser Spaltenfunktion aufgerufen werden. Die Funktion besitzt die folgenden zwei Rückgabewerte:

- 1, wenn die Zeile eine `NULL` enthält, die durch `ROLLUP` oder `CUBE` hervorgerufen wurde. Dies ist genau bei Untersummen und Summen der Fall.

- 0, wenn ein anderer Wert oder ein in der Datenbank gespeicherter NULL-Wert in der Zeile erscheint.

```
SELECT SUBSTR(K_Bereich,1,5)     AS Bereich,
       SUBSTR(K_Titel,1,10)      AS Titel,
       COUNT(DISTINCT kurs.K_Nr) AS Kurse,
       COUNT(B_Nr)               AS Buchungen,
       GROUPING(K_Bereich)       AS B,
       GROUPING(K_Titel)         AS T
  FROM kurs INNER JOIN termin
    ON kurs.K_Nr = termin.K_Nr
          INNER JOIN buchung
    ON termin.T_Nr = buchung.T_Nr
 GROUP BY CUBE (K_Bereich, K_Titel);
```

Listing 3.149 351_13.sql: Verwendung von GROUPING bei CUBE

Im Ergebnis erhält man zwei weitere Spalten, die die Werte für die beiden Dimensionen Bereich und Titel individuell charakterisieren. Überall dort, wo ein Wert aus der Tabelle verwendet wird, wie dies in den ersten beiden Zeilen des verkürzten Ergebnisses der Fall ist, erhält man den Zahlwert 0. Überall dort hingegen, wo Berechnungen zu NULL-Werten führen, wie dies in den beiden folgenden Zeilen für die Kurse pro Bereich für die Spalte K_Titel der Fall ist, erhält man den Zahlwert 1. Während in dieser Dimension wie auch für ihr Pendant im Titel-Bereich jeweils ein Wert – Titel oder Bereich – direkt aus der Datenbank stammt, erhält man für eine Spalte eine 1 und für die andere Spalte eine 0. Dies ändert sich in der letzten Zeile, in der nur noch die Gesamtsumme ausgegeben wird. Dies bewirkt gleichermaßen für die Titel- und Bereichsspalte NULL-Werte.

```
BEREI TITEL           KURSE  BUCHUNGEN          B          T
----- ---------- ---------- ---------- ---------- ----------
Daten Access              3        133          0          0
Daten MS SQL-Ser          2         21          0          0
Serve                     3         91          0          1
Webde                     5        240          0          1
      ASP.NET             1         30          1          0
      Access              3        133          1          0
                         79       2835          1          1
```

Für die ROLLUP-Erweiterung erhält man bei der Verwendung dieser Funktion notwendigerweise eine bestimmte Struktur, da hier nach den definierten Grup-

pen Untersummen gebildet werden und daher stets eine Reihe mit 0-Werten für die Dimensionen erscheint.

```
SELECT K_Bereich                              AS Bereich,
       TO_CHAR(T_Beginn, 'YYYY')              AS Jahr,
       SUM(B_Preis)                           AS Summe,
       GROUPING(K_Bereich)                    AS K,
       GROUPING(TO_CHAR(T_Beginn, 'YYYY')) AS T
  FROM teilnehmer INNER JOIN buchung
    ON buchung.TN_Nr = teilnehmer.TN_Nr
                  INNER JOIN termin
    ON buchung.T_Nr = termin.T_Nr
                  INNER JOIN kurs
    ON kurs.K_Nr = termin.K_Nr
  GROUP BY ROLLUP (TO_CHAR(T_Beginn, 'YYYY'), K_Bereich);
```

Listing 3.150 351_14.sql: GROUPING bei ROLLUP

Man erhält also folgendes typisches Ergebnis mit 0-Werten in den Untersummen und in der Gesamtsumme am Ende der Ausgabe:

```
BEREICH                        JAHR      SUMME          K  T
------------------------------ ---- ---------- ---------- --
Programmierung                 2003     566631          0  0
Programmierung(Office)         2003      17811          0  0
                               2003    1024639          1  0
                                       2091329          1  1
```

Eine weitere Variante bietet sich bei der Verwendung der `GROUPING`-Funktion in der Gruppensuchbedingung mit `HAVING` an. Hier eröffnet sich die Möglichkeit, explizit Summen- und/oder Untersummenzeilen auszuwählen und damit Unterergebnisse sowie Rohdaten auszublenden. Im folgenden Beispiel ersetzen wir nur den letzten Teil der obigen Abfrage, sodass nur sämtliche Summenzeilen ausgegeben werden:

```
...
  GROUP BY ROLLUP (TO_CHAR(T_Beginn, 'YYYY'), K_Bereich)
  HAVING GROUPING(TO_CHAR(T_Beginn, 'YYYY')) = 1
      OR GROUPING(K_Bereich) = 1;
```

Listing 3.151 351_15.sql: Zugriff auf GROUPING-Ergebnisse

Man erhält folgendes Ergebnis, das nur die Summenzeilen ausgibt:

```
B JAHR       SUMME           K           T
- ---- ---------- ---------- ----------
  2000     413543           1           0
  2001     209712           1           0
  2002     443435           1           0
  2003    1024639           1           0
          2091329           1           1
5 Zeilen ausgewählt.
```

Gruppierungsebenen mit GROUPING_ID

Um zu ermitteln, zu welcher Gruppierungsebene eine spezielle Zeile gehört, verwendet man die Funktion `GROUPING_ID`. Diese Funktion erwartet mindestens eine Spalte bzw. eine Dimension. Möchte man – was wohl der Regelfall ist – alle Dimensionen analysieren und den einzelnen Ergebniszeilen einen Ebenenwert zuweisen, listet man diese in dem Klammerausdruck der Funktion auf. Das Ergebnis bemisst sich beispielsweise für zwei Dimensionen wie in der folgenden Übersicht. Dabei entspricht die erste Zeile mit dem Wert 0 für die `GROUPING_ID`-Funktion einer normalen, gewöhnlichen Gruppierung. Erst die Betrachtung der einzelnen Achsen sowie der Gesamtsumme verlangt weitere Gruppierungen und damit auch eine höhere Gruppierungsebene.

Gruppierung	Bit-Vektor	GROUPING_ID-Wert
a,b	00	0
a	01	1
b	10	2
Gesamtsumme	11	3

Tabelle 3.8 Werte der GROUPING_ID-Funktion

Im folgenden Beispiel lässt man sich für die in den `GROUPING SETS` exakt benannten drei Achsen und die Gesamtsumme auch die Gruppierungsebene ausgeben.

```
SELECT TO_CHAR(T_Beginn, 'MM')     AS Monat,
       K_Bereich                   AS Bereich,
       COUNT(B_Nr)                 AS Zahl,
       GROUPING_ID(K_Bereich,
          TO_CHAR(T_Beginn, 'MM')) AS Ebene
  FROM buchung INNER JOIN termin
    ON buchung.T_Nr = termin.T_Nr
```

```
                INNER JOIN kurs
    ON termin.K_Nr = kurs.K_Nr
 GROUP BY GROUPING SETS (ROLLUP(TO_CHAR(T_Beginn,'MM')),
                         K_Bereich,(TO_CHAR(T_Beginn,'MM'),
                         K_Bereich));
```

Listing 3.152 351_16.sql: Gruppierungsebenen bestimmen

Im folgenden Ergebnis sind die oben in den `GROUPING SETS` definierten Achsen und die mit `ROLLUP` definierte Gesamtsumme angegeben:

```
MO BEREICH                              ZAHL      EBENE
-- ------------------------------ ---------- ----------
01 Grafik                                  4          0
   Programmierung(Office)                 76          1
01                                        70          2
                                        2835          3
```

Duplikatsuche mit GROUP_ID

Die komplexen Ergebnisse können in einigen Situationen Duplikate hervorbringen, die manchmal natürlich durch andere Schreibweisen zu vermeiden wären, aber manchmal auch unumgänglich sind. In solchen Fällen ist es durchaus keine Kosmetik, solche Duplikate auszublenden, da sie – anders als bei gewöhnlichen Abfragen, in denen Duplikate oft sofort ins Auge fallen – Ergebnisbewertungen oder die weitere Verarbeitung komplett verfälschen würden. Mit der `GROUP_ID`-Funktion, die ohne Spaltenaufruf, also ohne Parameter, zum Einsatz kommt, lassen sich solche Duplikate mit dem Zahlwert 1 in der `GROUP_ID`-Spalte kennzeichnen. Dabei erhält die erste Ausgabe eines Datensatzes den Zahlwert 0, die zweite den Zahlwert 1 und die folgende eine 2 usw. Das gesamte Abfrageergebnis wird also von oben nach unten verarbeitet und zeilenweise miteinander verglichen.

In der folgenden Abfrage möchte man explizit noch eine Liste aller Bereiche mit ihren Teilnehmerzahlen erhalten, wobei die Spalte `K_Bereich` doppelt in der `CUBE`-Klausel erscheint. Dies führt im Ergebnis dazu, dass zwar die gewünschten Daten zusätzlich ausgegeben werden, dies allerdings mit Duplikaten für die anderen beiden Achsen verbunden wird.

```
SELECT TO_CHAR(T_Beginn, 'YYYY') AS Zeit,
       SUBSTR(T_Ort,1,10)        AS Ort,
       SUBSTR(K_Bereich,1,10)    AS Bereich,
       COUNT(DISTINCT T_Nr)      AS Termine,
       COUNT(TN_Nr)              AS Summe,
```

```
      GROUP_ID()
  FROM termin NATURAL JOIN buchung
              NATURAL JOIN kurs
  GROUP BY CUBE(TO_CHAR(T_Beginn, 'YYYY'),
                (T_Ort, K_Bereich),
                 K_Bereich);
```

Listing 3.153 351_17.sql: Duplikate herausfinden

Im Ergebnis erhält man, wie oben erwähnt, die zusätzlichen Informationen, aber auch Duplikate:

```
ZEIT ORT        BEREICH       TERMINE      SUMME GROUP_ID()
---- ---------- ---------- ---------- ---------- ----------
2003 Berlin     Datenbanke          5         22          0
2003 Berlin     Datenbanke          5         22          1
     Bochum     Programmie        223       1105          1
                Datenbanke        109        557          0
                Grafik             74        384          0
```

Um die Vorteile einer komplexen Definition von Dimensionen zu nutzen, kann man über eine Gruppensuchbedingung in der `HAVING`-Klausel die Duplikate ausblenden:

```
  GROUP BY CUBE(TO_CHAR(T_Beginn, 'YYYY'),
                (T_Ort, K_Bereich), K_Bereich)
  HAVING GROUP_ID() < 1;
```

Listing 3.154 351_18.sql: Ausblenden von Duplikaten

3.5.2 Rangordnungen erstellen

Für die Ermittlung von Rangfolgen aller Art stellt Oracle fast zwei Hand voll Funktionen bereit, mit denen unterschiedliche Hitparaden wie auch Häufigkeitsverteilungen eingerichtet werden können. Da ihre Verwendung sehr einfach ist, werden wir die Darstellung auf einige Beispiele beschränken.

Rangfolgen einrichten

Als Ergänzung – wenn auch nicht zur syntaktischen Erweiterung – der allgemeinen `ORDER BY`-Klausel gibt es zwei weitere Funktionen, mit denen sich Sortierungen unter Verwendung einer Hitparadennummer vornehmen lassen. Dabei unterscheiden sich beide Funktionen nur in der Art und Weise der Zuordnung von Hitparadenplätzen, nicht aber in der allgemeinen Syntax:

- `all_method_results`: Rückgabewerte von Methoden von aufrufbaren Objekttypen
- `dba_method_results`: Rückgabewerte von Methoden aller Objektypen
- `user_method_results`: Rückgabewerte von Methoden von Objekttypen des aktuellen Benutzers

- **Trigger**
  - `all_internal_triggers`: Trigger auf aufrufbare Tabellen
  - `dba_internal_triggers`: alle Trigger auf allen Tabellen
  - `user_internal_triggers`: benutzereigene Trigger
  - `all_triggers`: alle aufrufbaren Trigger
  - `dba_triggers`: alle Trigger
  - `user_triggers`: alle Benutzer-Trigger
  - `all_trigger_cols`: alle Spalten von aufrufbaren Triggern
  - `dba_trigger_cols`: Spalten in allen Triggern
  - `user_trigger_cols`: Spalten in Benutzer-Triggern

5.2 Funktionen und Prozeduren erstellen

Das Erstellen einer Funktion ist ebenso wie die Definition einer Prozedur überaus simpel. Die allgemeine Syntax bietet zwar eine Menge an Möglichkeiten für die Einstellungen und Verhaltensweisen von Funktionen an, doch letztendlich handelt es sich um ein Programm, das nur einen einzigen Wert zurückliefert und um einige syntaktische Details ergänzt wurde, die es zu einer Funktion machen.

5.2.1 Allgemeine Syntax für Funktionen

Die allgemeine Syntax hat also folgende Form:

```
[CREATE [OR REPLACE ] ]
FUNCTION name [ ( parameter [modus] datentyp
              [ , parameter [modus] datentyp]... ) ]
 RETURN datentyp
[ AUTHID { DEFINER | CURRENT_USER } ]
[ PARALLEL_ENABLE
[ { [CLUSTER parameter BY (spaltenname [,spaltenname]... ) ] |
[ORDER parameter BY (spaltenname [ , spaltenname]... ) ] } ]
[ ( PARTITION parameter BY
{ [ {RANGE | HASH } (spaltenname [,spaltenname]...)] | ANY }
```

```
) ]
]
[DETERMINISTIC] [ PIPELINED [ USING implementationstyp ] ]
[ AGGREGATE [UPDATE VALUE] [WITH EXTERNAL CONTEXT]
USING implementationstyp] {IS | AS}
[ PRAGMA AUTONOMOUS_TRANSACTION; ]
[ lokale anweisungen ]
BEGIN
 anweisungen
[ EXCEPTION
 ausnahmebehandlung ]
END [ name ];
```

Da eine Funktion in der Datenbank als Schema-Objekt gespeichert wird, gilt auch hier für die Definition der `CREATE`- Befehl und für das Löschen der `DROP`-Befehl. Die Funktionsweise von Funktionen können Sie – wie zuvor beschrieben – in der *Oracle Enterprise Manager Konsole* bearbeiten, allerdings nicht in *SQL*Plus*. Hier können Sie nur eine komplett neue Funktion durch die Zusatzklausel `REPLACE` speichern, die die alte, gleichnamige Funktion überschreibt.

Für die dann folgende Parameterliste gilt, dass sie die einzelnen Parameter sowie Ihre Modi (Standardwert ist `IN`) und Datentypen durch Komma getrennt und in zwei runden Klammern zusammengefasst aufzählen. Der Rückgabewert wird lediglich mit dem Schlüsselwort `RETURN` und seinem Datentyp gekennzeichnet. Für alle diese Datentypen gilt die eherne Regel, dass keine Beschränkungen in Länge, Genauigkeit oder Größe vorgenommen werden dürfen. Innerhalb der Funktion wird zur Laufzeit die Beschränkung des übergebenen Parameters – falls vorhanden – berücksichtigt. Alternativ können Sie allerdings eine Typableitung mit `%TYPE` vornehmen. Dies impliziert eine Beschränkung, wobei diese allerdings auch erst zur Laufzeit ermittelt wird und daher analog zur Beschränkungsübernahme bei der Parameterübergabe abläuft.

Auf das Schlüsselwort `IS` (alternativ `AS`) folgt dann der Anweisungsblock, der durch `BEGIN` und `END funktionsname` umschlossen wird. (Der Funktionsname ist optional, dient allerdings einer schnelleren Fehlerüberprüfung.) Innerhalb des Anweisungsblocks befindet sich dann an letzter Stelle der Rückgabewert der gesamten Funktion, der erneut mit dem Schlüsselwort `RETURN` gekennzeichnet wird und der verschiedene Datentypen enthalten kann: skalare Datentypen wie `VARCHAR2` oder `NUMBER`, natürlich auch `BOOLEAN`, dann auch Sammlungen (verschachtelte Tabellen, Varrays), Datensätze, Objekttypen und große Objekte. Um einen Cursor zurückzugeben, muss man eine eine Cursor-Referenz nutzen.

Neben diesen Standard- oder Minimalanforderungen an eine Funktion finden Sie in der allgemeinen Syntax noch folgende Alternativen und Zusatzangaben wieder:

- `AUTHID` legt fest, ob die Funktion sich lediglich vom Besitzer (Standardwert) ausführen läst oder durch den aktuellen Benutzer (`CURRENT_USER`).
- `PARALLEL_ENABLE` kennzeichnet eine Funktion als sicher, wenn sie in Parallelanweisungen von DML-Befehlen verwendet wird.
- `DETERMINISTIC` ist ein Kompilierungshinweis, der nicht zwangsläufig berücksichtigt werden muss. Er unterstützt allerdings die Kompilierung insoweit, als dass frühere Funktionsaufrufe und ihre Ergebnisse verwendet werden können, bevor redundante bzw. wiederholte Aufrufe stattfinden.
- Das Pragma `AUTONOMOUS_TRANSACTION` legt eine Funktion als unabhängig fest, sodass sie eine eigene Transaktion darstellt und solche Befehle wie `COMMIT` oder `ROLLBACK` verwenden kann und erst nach Abarbeitung die Kontrolle an das auslösende Programm zurückgibt.
- Als eigenständiger Bereich ist auch die Definition einer Ausnahmebehandlung optional, der mit dem Schlüsselwort `EXCEPTION` beginnt.

Sie finden in der allgemeinen Syntax noch verschiedene weitere Angaben, die den Speicherort der Funktion festlegen und eher in die Administration statt in die Programmierung gehören.

5.2.2 Blockstruktur von Funktionen

In der *Oracle Enterprise Manager Konsole* haben Sie bereits gesehen, dass die Funktionsdefinition bzw. der Quelltext, der sie bildet, in zwei Hälften geteilt wurde: Der Name befand sich in einem eigenen Bereich der grafischen Oberfläche, während der Rest im Editierbereich als einfacher Text bereitstand. Theoretisch lässt sich die Blockstruktur von Funktionen noch weiter aufgliedern, was im folgenden Beispiel dargestellt werden soll.

Wir werden an diesem Beispiel verschiedene Themen erarbeiten, sodass wir nur an dieser Stelle kurz auf die Funktionalität eingehen. Bis jetzt haben wir nur in einem einzigen Beispiel auf die Problematik Rücksicht genommen, dass der Preis für einen Kurs von der Zahl der Teilnehmer abhängt, die ein Unternehmen anmeldet. Wenn also eine Firma oder ein Verein drei Teilnehmer für ein Seminar anmeldet, verringert sich der Preis um einige Prozentpunkte, was anhand von Preisstufen in der wenig benutzten Tabelle `PREIS` zu bewundern ist. Ohne dynamisches SQL ist es nicht möglich, nur einen einzigen Preis zu erhalten, ohne komplizierte Fallunterscheidungen für die Auswahl des passenden Feldes durchführen zu müssen. Natürlich ist für die Anmeldung von Teil-

nehmern wie auch für die Stornierung von Buchungen jedes Mal eine Ermittlung des passenden Preises für die Teilnehmer der Kleingruppe nötig, sodass dies – wie viele andere Dinge auch – ein hervorragender Kandidat für die Verwendung einer Funktion ist. Eine Funktion bietet sich deswegen an, weil genau ein Wert erwartet wird, der letztendlich nur über die Kursnummer und die Anzahl der Teilnehmer zu bestimmen ist.

Zunächst könnte man einen anonymen Block mit entsprechender Funktionalität in der folgenden Form verwenden. Für die Konstruktion sind zwei Dinge bedeutsam: Zum einen wird die gesamte Preisermittlung in zwei verschachtelten Blöcken untergebracht, damit Ausnahmen, die im Ausführungs- und Deklarationsabschnitt (des untergeordneten Blocks) auftreten, insoweit verarbeitet werden, als dass dem Preis in diesem Fall der Wert 0 zugewiesen wird. Alternativ könnte in diesem Zusammenhang (und das gilt auch für die Funktion weiter unten) eine Fehlernummer mit auf den Weg gegeben werden, die dann in der umliegenden Anwendung ausgewertet werden könnte. Zum anderen haben Sie hier noch einmal ein interessantes Beispiel für die Verwendung von nativem dynamischen SQL, da die Abfrage so in einer Zeichenketten-Variable zusammengesetzt wird, dass anhand der zugeordneten oder übermittelten Teilnehmeranzahl eine passende Spalte ausgewählt wird. (Auch hier werden schon mögliche Fehler oder unerwünschte Werte berücksichtigt.)

In diesem anonymen Block und in den zwei folgenden Funktionen, die auf dem anonymen Block aufbauen, finden Sie eine Fallunterscheidung für die Teilnehmerzahl. Diese soll nur sicherstellen, dass maximal fünf Teilnehmer mit einem speziellen Preisnachlass rechnen können und auch höhere Teilnehmerzahlen immer mit dem TN5-Preis berücksichtigt werden und nicht etwa mit weiteren Nachlässen. Andererseits soll auch bei Fehleingaben wie negativen Zahlen oder 0 mindestens der TN1-Preis ermittelt werden. Im Standardfall verwendet man einfach die vorhandene Teilnehmerzahl.

```
<<Berechnung>>
DECLARE
 v_KPreis preis.P_TN1%TYPE;      -- Kurspreis
 v_SQL    VARCHAR2(300);         -- Abfrage
 BEGIN
 <<Ermittlung>>
 DECLARE
  b_KNr    kurs.K_Nr%TYPE := 1025051; -- Kursnr
  b_TNZahl buchung.B_TZahl%TYPE := 9; -- TN-Zahl
 BEGIN
  IF b_TNZahl > 5
```

```
   THEN b_TNZahl := 5;
  ELSIF b_TNZahl <= 0
   THEN b_TNZahl := 1;
  END IF;
  v_SQL := 'SELECT P_TN'
          || b_TNZahl
          || ' FROM kurs NATURAL JOIN preis'
          || ' WHERE K_Nr='
          || b_KNr;
  EXECUTE IMMEDIATE v_SQL
  INTO v_KPreis;
  DBMS_OUTPUT.PUT_LINE(v_KPreis);
 END Ermittlung;
 EXCEPTION
 WHEN VALUE_ERROR OR NO_DATA_FOUND
  THEN v_KPreis := 0;
END Berechnung;
```

Listing 5.6 522_01.sql: Ermittlung von Preisen

Die folgende Funktion ist eine von zwei möglichen Funktionen, die die obige Anwendung strukturell übernehmen. Sie bietet im Gegensatz zur nächsten Implementierung einen Parameter `Teilnehmerzahl` an, der mit dem Parametermodus `IN OUT` gekennzeichnet ist. So ist es möglich, dass gleichzeitig eine Teilnehmerzahl über diesen Parameter in die Funktion einfließt und dort sowohl gelesen als auch mit neuen Werten – wie in der Fallunterscheidung für die Wertuntersuchung – gefüllt werden kann.

```
CREATE OR REPLACE FUNCTION BerechnePreis (
 Kursnummer     IN     kurs.K_Nr%TYPE,
 Teilnehmerzahl IN OUT buchung.B_TZahl%TYPE)
RETURN NUMBER
IS
 v_KPreis preis.P_TN1%TYPE;     -- Kurspreis
 v_SQL    VARCHAR2(300);        -- Abfrage
BEGIN
  IF Teilnehmerzahl > 5
   THEN Teilnehmerzahl := 5;
  ELSIF Teilnehmerzahl <= 0
   THEN Teilnehmerzahl := 1;
  ELSE Teilnehmerzahl := Teilnehmerzahl;
  END IF;
```

```
  v_SQL := 'SELECT P_TN'
           || Teilnehmerzahl
           || ' FROM kurs NATURAL JOIN preis'
           || ' WHERE K_Nr='
           || Kursnummer;
 EXECUTE IMMEDIATE v_SQL
 INTO v_KPreis;
 RETURN v_KPreis;
 EXCEPTION
 WHEN VALUE_ERROR
  THEN v_KPreis := 0;
END BerechnePreis;
```

Listing 5.7 522_02.sql: BerechnePreis mit IN- und IN OUT-Parametern

In Abbildung 5.9 wurde für die in diesem Abschnitt erstellte Funktion die Blockstruktur herausgehoben:

- Der **Kopf-Abschnitt** enthält den Funktionsnamen, so wie er in der Datenbank gespeichert wird. Dieser enthält die Parameterliste und den Datentyp des `RETURN`-Werts.
- Der **Deklarationsabschnitt** enthält weitere Variablen oder andere syntaktische Konstrukte wie Cursor und Typen, die nur innerhalb der Funktion gültig und sichtbar sind. Dieser Abschnitt entspricht prinzipiell dem Deklarationsabschnitt in anonymen Blöcken.
- Der **Ausführungsabschnitt** enthält die eigentliche Funktionalität der Funktion wie in einem anonymen Block. Er umschließt einen möglichen Ausnahmenabschnitt und endet mit der Rückgabe durch die `RETURN`-Klausel.
- Der **Ausnahmeabschnitt** wird am Ende des Ausführungsabschnitts platziert. Er verarbeitet zunächst Ausnahmen, die innerhalb des Ausführungsabschnitts der aktuellen Funktion oder dann auch im Deklarationsabschnitt einer weiteren aufgerufenen (oder verschachtelt aufgerufenen) modularen Struktur aufgetreten sind.

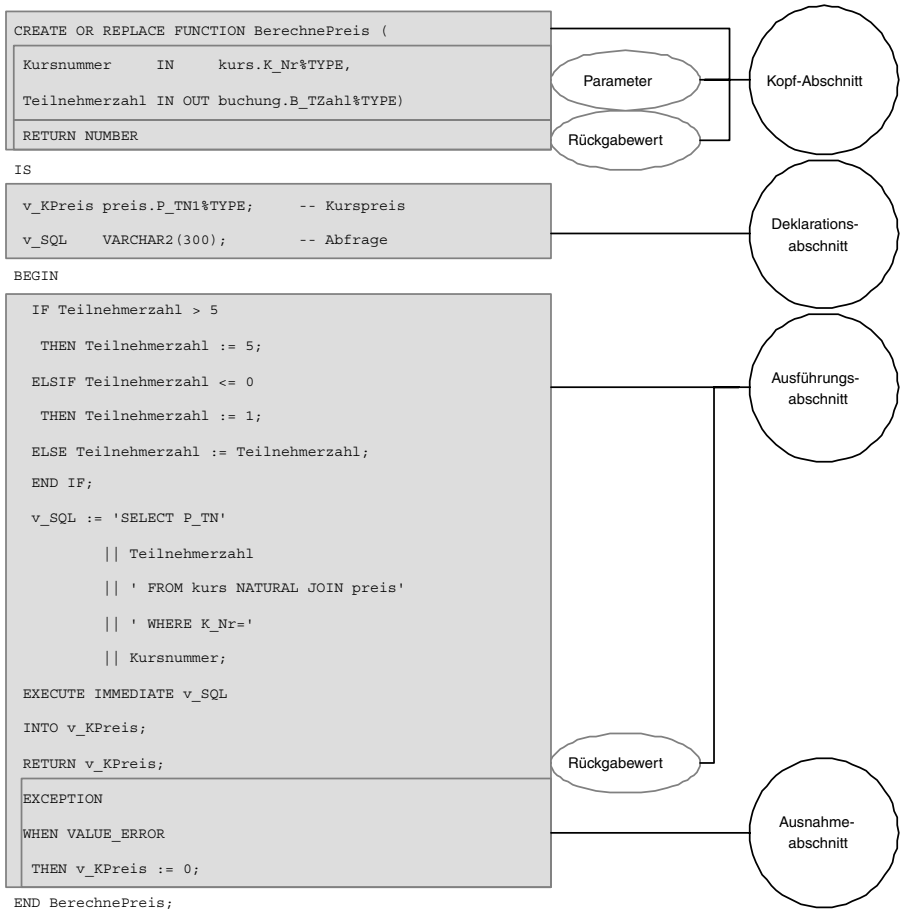


Abbildung 5.9 Blockstruktur von Funktionen

5.2.3 Entscheidungen für Parameter-Modi

Beschränkungen in Programmiersprachen dienen wenn ihre Struktur gut konzeptioniert ist dazu, dass Anwendungen leichter verständlich und besser wartbar bleiben. Manchmal sind Beschränkungen allerdings lästig, sodass man geneigt ist, Abkürzungen zu wählen, die sich über Standardwerte oder Joker-Elemente mit multifunktionalen Eigenschaften anbieten. In diesen Zusammenhang gehören auch die drei Parameter-Modi, deren Eigenschaften wir zuvor vorgestellt haben. Sobald man einen Parameter als `IN` deklariert hat, ist der Parameter nicht nur für den Eingang in eine Funktion zuständig und benutzbar, sondern ist auch automatisch für die Veränderung von Werten innerhalb höchst unzugänglich. Er wirkt also wie eine Konstante und löst bei Wertzuweisungen linksseitig eine Fehlermeldung aus, die ihn als unpassenden Zuwei-

sungsort charakterisiert. Ein mit `OUT` deklarierter Parameter lässt sich dagegen erst innerhalb der Funktion ansprechen und kann keine Werte in die Funktion übernehmen. Sie werden bei der Vorstellung der Ausnahmebehandlung sogar sehen, dass eine unschöne Fehlermeldung ausgelöst werden kann, wenn man es dennoch versucht, nämlich der Hinweis auf den unpassenden Zuweisungsort.

Im aktuellen Beispiel ist rein zufällig der Parameter `Teilnehmerzahl` ein typischer Kandidat, der für einen kürzeren Quelltext am besten mit `IN OUT` deklariert wird, da in Abhängigkeit von wünschenswerten (1 bis 5) und nicht wünschenswerten Werten (kleiner 1 und größer 5) erneut Wertzuweisungen anfallen, damit die Abfrage auch ein sinnvolles Ergebnis zurückliefert – und nicht Fehlermeldungen wie `NO_DATA_FOUND` oder dass keine passende Spalte gefunden wurde.

Selbstverständlich arbeitet man am ordentlichsten, wenn man gerade keine Joker oder Standardwerte (in diesem Fall `IN`) benutzt, sondern lieber – wie in der folgenden Variante der Preisermittlungsfunktion – eine eigene Variable im Deklarationsabschnitt versteckt, die dann passende Werte aus dem `IN`-Parameter übernimmt und ggf. neue Werte aufgrund von Fallunterscheidungen annimmt.

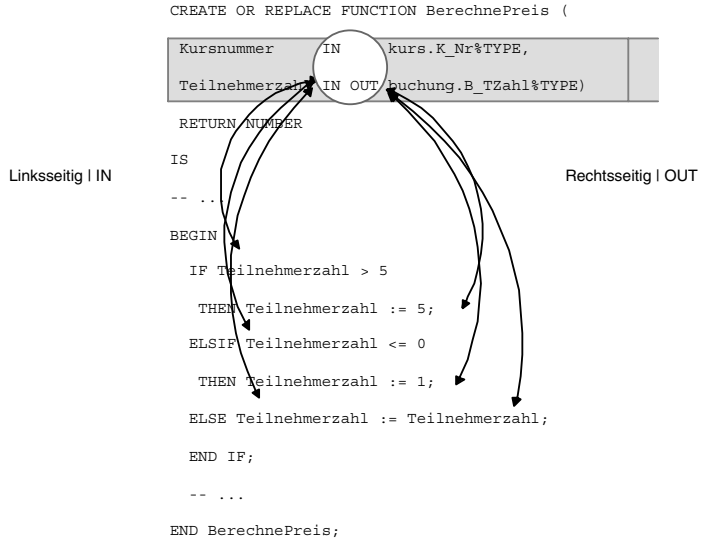


Abbildung 5.10 Parameter-Modi und Verhaltensweisen im Vergleich

Der dritte Parameter-Modus ist im Zusammenhang mit Funktionen nicht so nützlich, wie das folgende Beispiel zeigt. Sie können ihn für Fälle einsetzen, in

denen Sie speziell den Ausgabeparameter innerhalb des Programms nur schreibend ansprechen wollen und dadurch den RETURN-Wert erzeugen möchten. Dies werden Sie auch im folgenden Beispiel sehen. Allerdings wird über ihn kein Wert zurückgeliefert, wie es für Prozeduren der Fall ist, sondern tatsächlich nur über den RETURN-Wert. Bei Prozeduren stellen die Ausgabewerte die Rückgabewerte dar, entsprechen also quasi dem RETURN-Wert von Funktionen, wobei keine Einschränkung für ihre Zahl gilt.

Im folgenden Beispiel variieren wir die obige Funktion so, dass an erster Stelle noch ein Ausgabeparameter bzw. ein Parameter mit dem Parametermodus OUT verzeichnet ist. Ansonsten entspricht sie der vorherigen Version dieser Funktion.

```
CREATE OR REPLACE FUNCTION BerechnePreis3  (
 Preis          OUT     buchung.B_Preis%TYPE,
 Kursnummer     IN      kurs.K_Nr%TYPE,
 Teilnehmerzahl IN  OUT buchung.B_TZahl%TYPE )
RETURN NUMBER
IS
 v_SQL    VARCHAR2(300);        -- Abfrage
BEGIN
  IF Teilnehmerzahl > 5
   THEN Teilnehmerzahl := 5;
  ELSIF Teilnehmerzahl <= 0
   THEN Teilnehmerzahl := 1;
  ELSE Teilnehmerzahl := Teilnehmerzahl;
  END IF;
  v_SQL := 'SELECT P_TN'
           || Teilnehmerzahl
           || ' FROM kurs NATURAL JOIN preis'
           || ' WHERE K_Nr='
           || Kursnummer;
 EXECUTE IMMEDIATE v_SQL
 INTO Preis;
 RETURN Preis;
EXCEPTION
 WHEN VALUE_ERROR
  THEN Preis := 0;
END BerechnePreis;
```

Listing 5.8 523_01.sql: Preisberechnung

Beim Aufruf soll nun ein Eintrag in die Tabelle BUCHUNG anhand diverser Parameter wie Kursnummer, Teilnehmernummer und Terminnummer erfolgen. Dabei ermitteln die Funktionen BerechnePreis3() und ZaehleReihen() automatisch die benötigten Werte für den Preis und die Buchungsnummer. Achten Sie hierbei darauf, dass Sie auch bei verwendetem OUT-Parameter die Funktion als Ausdruck einer Variablen zuweisen müssen und dass Sie die Funktion nicht wie eine Prozedur quasi aus dem Nichts aufrufen können. Dies sind Fähigkeiten, die nur von Prozeduren bereitgestellt werden.

```
DECLARE
 v_BPreis buchung.B_Preis%TYPE;
 v_KNr    kurs.K_Nr%TYPE        := 1015067;
 v_MaxBNr buchung.B_Nr%TYPE;
 v_TNNr   teilnehmer.TN_Nr%TYPE := 521;
 v_TNr    termin.T_Nr%TYPE      := 321;
 v_TZahl  buchung.B_TZahl%TYPE  := 1;
BEGIN
 -- Ermittlung der Buchungsnummer
 v_MaxBNr := ZaehleReihen(v_MaxBNr, 'buchung', 'B_Nr');
 -- Preisermittlung
 v_BPreis := BerechnePreis3(v_BPreis, v_KNr, v_TZahl);
 -- Erfassung der Buchung
 INSERT INTO buchung
 VALUES (v_MaxBNr+1, v_TNNr, v_TNr, SYSDATE,
         v_TZahl, v_BPreis, NULL, NULL, NULL);
END;
```

Listing 5.9 523_01.sql: Erfasssung einer Buchung

Mit der Abfrage `SELECT B_Nr, TN_Nr, T_Nr, B_Datum, B_TZahl, B_Preis FROM buchung;` erhalten Sie dann folgendes Ergebnis, das den erfolgreichen Eintrag in der Tabelle dokumentiert.

```
      B_NR       TN_NR        T_NR    B_TZAHL    B_PREIS
---------- ---------- ---------- ---------- ----------
         1        521        321          1       1430
1 Zeile wurde ausgewählt.
```

Sie können PL/SQL-Funktionen auch in SQL verwenden, sodass Sie gewissermaßen dennoch Funktionen – wie oben formuliert – aus dem Nichts heraus aufrufen können. Dies entspricht aber vielmehr dem Umstand, dass in SQL-Befehlen überall dort Funktionen eingesetzt werden können, wo genau ein Wert benötigt wird. In der Datei *624_01.sql* finden Sie eine weitere und letzte

Variante dieser Preisermittlungsfunktion, `BerechnePreis2()`. In dieser Variante gibt es nur `IN`-Parameter. Eine solche Funktion eignet sich z.B. auch für den Einsatz in einem SQL-Befehl als quasi selbst geschaffene (PL/)SQL-Funktion. Sowohl Parameter mit dem Modus `OUT` als auch Parameter mit dem Modus `IN OUT` liefern folgende Fehlermeldung zurück, was Sie mit Hilfe des anonymen Blocks in der Datei *523_02.sql* verifizieren können.


```
        BerechnePreis(v_KNr, v_TZahl), NULL, NULL, NULL);
        *
FEHLER in Zeile 14:
ORA-06550: Zeile 14, Spalte 9:
PL/SQL: ORA-06572: Funktion BERECHNEPREIS hat Out-Argumente
ORA-06550: Zeile 12, Spalte 2:
PL/SQL: SQL Statement ignored
```

Eine SQL-taugliche Funktion hat dagegen – neben anderen Erfordernissen, die später noch dargestellt werden – folgendes Format im Kopf-Abschnitt:

```
CREATE OR REPLACE FUNCTION BerechnePreis2  (
 Kursnummer     IN kurs.K_Nr%TYPE,
 Teilnehmerzahl IN buchung.B_TZahl%TYPE)
```

Diese Funktion lässt sich dann leicht in einem SQL-Befehl aufrufen. Den kompletten anonymen Block finden Sie dazu in der Datei *532_03.sql*, die zusätzlich die Variablendeklaration aus dem letzten Aufruf besitzt.

```
BEGIN
 -- Ermittlung der Buchungsnummer
 v_MaxBNr := ZaehleReihen(v_MaxBNr, 'buchung', 'B_Nr');
 -- Erfassung der Buchung
 INSERT INTO buchung
 VALUES (v_MaxBNr+1, v_TNNr, v_TNr, SYSDATE, v_TZahl,
        BerechnePreis2(v_KNr, v_TZahl), NULL, NULL, NULL);
END;
```

Da eine Funktion nur einen einzigen Wert zurückliefert und da der Mindeststandard der `IN`-Parameter erfüllt ist, trägt auch dieser sehr einfache und elegante Aufruf einen neuen Datensatz in die Tabelle `BUCHUNG` ein.

5.2.4 Ausnahmebehandlung

Die Aussage sein, dass auch eine Ausnahmebehandlung in Funktionen möglich ist, dürfte nach den vorherigen Ausführungen zur allgemeinen Syntax und zur Blockstruktur von Modulen (bzw. in diesem Fall von Funktionen) relativ unspektakulär sein. Die Regeln der Ausnahmebehandlung und die Struktur

ihres Einsatzes entsprechen komplett denen der üblichen Syntax, d.h., die Ausführungen im Abschnitt zur Ausnahmebehandlung in anonymen Blöcken gelten hier ebenso.

Bei Funktionen ist die Tatsache interessant, dass man innerhalb der Ausnahmebehandlung einen passenden Rückgabewert kreieren muss, damit die Funktion auch eine Rückgabe durchführt, sobald es Komplikationen mit der Ausführung gibt. Dies wird in der Variation für die Preisermittlung dadurch gewährleistet, dass bei fehlenden Werten (`NO_DATA_FOUND`) oder zu vielen Treffern (`TOO_MANY_ROWS`) der Preis auf 0 gesetzt und diese Information dann über die Rückgabe an das aufrufende Programm zurückgegeben wird.

```
CREATE OR REPLACE FUNCTION BerechnePreis2  (
 Kursnummer     IN kurs.K_Nr%TYPE,
 Teilnehmerzahl IN buchung.B_TZahl%TYPE)
RETURN NUMBER
IS
 v_KPreis preis.P_TN1%TYPE;     -- Kurspreis
 v_SQL    VARCHAR2(300);        -- Abfrage
 v_TNZahl buchung.B_TZahl%TYPE; -- TN-Zahl
BEGIN
  IF Teilnehmerzahl > 5
   THEN v_TNZahl := 5;
  ELSIF Teilnehmerzahl <= 0
   THEN v_TNZahl := 1;
  ELSE v_TNZahl := Teilnehmerzahl;
  END IF;
  v_SQL := 'SELECT P_TN'
           || v_TNZahl
           || ' FROM kurs NATURAL JOIN preis'
           || ' WHERE K_Nr='
           || Kursnummer;
 EXECUTE IMMEDIATE v_SQL
 INTO v_KPreis;
 RETURN v_KPreis;
 EXCEPTION
 WHEN NO_DATA_FOUND OR TOO_MANY_ROWS
  THEN v_KPreis := 0;
       RETURN v_KPreis;
END BerechnePreis2;
```

Listing 5.10 524_01.sql: Ausnahmebehandlung in Modulen

Die Regelungen der Ausnahmebehandlung werden bei einem Aufruf von Modulen um eine Variation der dritten Regel erweitert, sodass folgendes Regelwerk entsteht:

1. Ausnahmen im Ausführungsabschnitt des Moduls und des aufrufenden Programms werden im jeweiligen Ausnahmeabschnitt behandelt. Wird dort keine passende Behandlung gefunden, wird Regel 2 angewandt.
2. Ausnahmen im Ausführungsabschnitt des Moduls und des aufrufenden Programms, die keine passende Behandlung im jeweiligen Ausnahmeabschnitt finden, werden an die Umgebung abgegeben. Damit gibt das Modul seine Ausnahme an das aufrufende Programm weiter.
3. Ausnahmen im Deklarationsabschnitt und/oder Kopf-Abschnitt des aufrufenden Programms oder des Moduls werden stets an die Umgebung weitergegeben. Damit geben Module ihre Ausnahmen in den beiden ersten Abschnitten stets an das aufrufende Programm weiter.

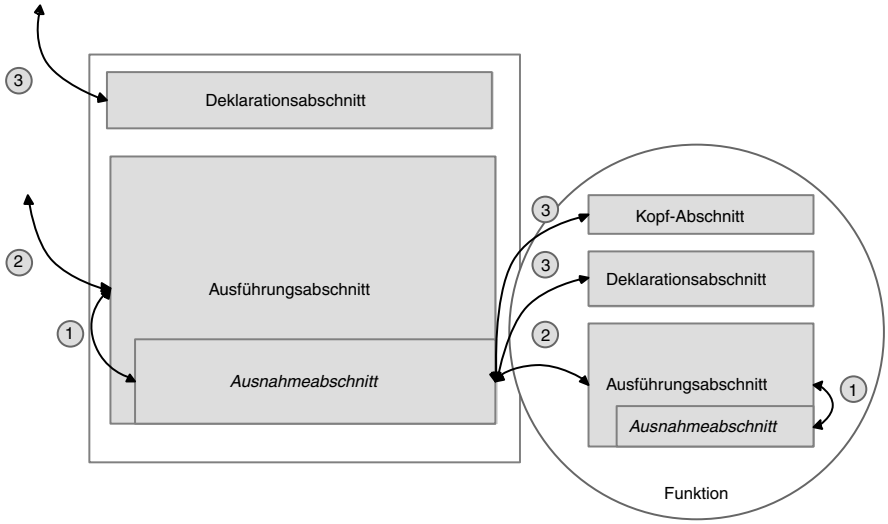


Abbildung 5.11 Ausnahmebehandlung in Modulen

Die Funktionsweise dieser Regelungen lässt sich sehr schön anhand eines einfachen anonymen Blocks zeigen, der mit unterschiedlichen Werten die gerade erstellten Funktionen falsch aufruft. Im nicht auskommentierten, ersten Fall übergibt man eine Zeichenkette statt der erwarteten Teilnehmerzahl, was einen `VALUE_ERROR` im Kopf-Abschnitt auslöst, da dieser Wert nicht erwartet wird. Hier tritt sofort Regel 3 in Kraft, und die Ausnahme des anonymen Blocks als aufrufendes Programm setzt den Preis auf 0. Im zweiten, auskommentierten Fall gibt man eine nicht existierende Kursnummer vor. Dies löst für die SQL-

Abfrage die Ausnahme NO_DATA_FOUND aus. Da dies im Ausführungsabschnitt der Funktion geschieht, kann dort bereits der Preis auf 0 gesetzt und an die aufrufende Umgebung zurückgeliefert werden. Würde hier eine Ausnahmebehandlung fehlen, dann müsste spätestens in der aufrufenden Umgebung diese Ausnahme berücksichtigt sein, damit keine endgültige Fehlermeldung entsteht.

```
SET SERVEROUTPUT ON;
DECLARE
 v_TNZahl buchung.B_TZahl%TYPE := 3;
 v_KPreis buchung.B_Preis%TYPE;
BEGIN
 -- Ausnahme im Kopf-Abschnitt: VALUE_ERROR
 DBMS_OUTPUT.PUT_LINE(BerechnePreis2(1025051,'a'));
 -- Ausnahme im Ausführungsabschnitt Funktion: NO_DATA_FOUND
 --DBMS_OUTPUT.PUT_LINE(BerechnePreis2(0025051, 3));
EXCEPTION
 -- Kopfabschnitt der Funktion
 WHEN VALUE_ERROR
  THEN v_KPreis := 0;
       DBMS_OUTPUT.PUT_LINE('Kopf-Abschnitt: ' || v_KPreis);
END;
```

Listing 5.11 524_02.sql: Ausnahmebehandlung und Regelungsstruktur

5.2.5 Übergabe von Werten bei IN und IN OUT

Wie Parameter überhaupt an Module übergeben werden, haben Sie bereits in den einleitenden Absätzen zur Kenntnis genommen und auch schon sicherlich mit der ähnlich strukturierten Übergabe in anderen Sprachen verglichen. An dieser Stelle soll noch einmal auf diesen Punkt eingegangen werden, wobei dieses Mal die beiden unterschiedlichen Übergabekonzepte bei den Parameter-Modi IN und IN OUT untersucht werden.

Standardmäßig erfolgt die Zuweisung über die Namens- oder Positionsnotation per Referenz in die Funktion. Dabei können nur IN-Parameter direkt Wertzugaben akzeptieren. Dies ist der Fall bei BerechnePreis2() im folgenden einfachen anonymen Block, wo die Anzahl der Teilnehmer direkt als Zahl beim Funktionsaufruf an die passende Position eingetragen wird. Bei einem IN OUT-Parameter, wie er in BerechnePreis() vorliegt, verhält es sich dagegen etwas anders. Hier ist nur eine Übergabe mit Hilfe von Variablen möglich.

```
DECLARE
 v_TNZahl buchung.B_TZahl%TYPE := 3;
```

```
 v_KPreis buchung.B_Preis%TYPE;
BEGIN
 -- Verwendung von IN OUT-Parametern nur über Variablen
 DBMS_OUTPUT.PUT_LINE(BerechnePreis(1025051,v_TNZahl));
 -- DBMS_OUTPUT.PUT_LINE(BerechnePreis(1025051,3)); -- Falsch!
 -- Verwendung von IN-Parametern auch über direkten Wert
 DBMS_OUTPUT.PUT_LINE(BerechnePreis2(1025051, v_TNZahl));
 DBMS_OUTPUT.PUT_LINE(BerechnePreis2(1025051, 3));
END;
```

Listing 5.12 525_01.sql: Aufruf von Funktionen und Wertübergabe

Im korrekten Fall erhält man folgende wenig interessante Ausgabe:

```
880
880
880
PL/SQL-Prozedur wurde erfolgreich abgeschlossen.
```

Dagegen taucht unmittelbar eine Fehlermeldung auf, sobald man versucht, dem `IN OUT`-Parameter einen Wert direkt zuzuweisen.

```
  DBMS_OUTPUT.PUT_LINE(BerechnePreis(1025051,3)); -- Falsch!
                                                  *
FEHLER in Zeile 7:
ORA-06550: Zeile 7, Spalte 46:
PLS-
00363: Ausdruck '3' kann nicht als Zuweisungsziel benutzt werden
ORA-06550: Zeile 7, Spalte 3:
PL/SQL: Statement ignored
```

5.2.6 Rückgaben mit RETURN

Eine Funktion macht nur Sinn, wenn sie auch eine `RETURN`-Anweisung beinhaltet. Sie kann nur eine einzige Einschränkung zurückgeben, allerdings ist dies nur auf die tatsächliche Rückgabe bezogen und hängt nicht davon ab, wie oft die `RETURN`-Klausel in einer Funktion steht. Wie Sie oben bereits bei der allgemeinen Syntax gesehen haben, muss zunächst im Kopf-Abschnitt einer Funktion der unbeschränkte Datentyp einer Funktion definiert werden. Der eigentliche Aufruf erfolgt dann über die `RETURN`-Klausel, die einen Ausdruck anschließt. In den zurückliegenden Beispielen bestand diese Klausel aus Variablenwerten, was eventuell den Eindruck erzeugt hat, dass nur die Variablen als Rückgabewert fungieren können. Tatsächlich aber wird ja nicht die Variable

zurückgegeben, sondern ihr Wert, da eine Variable genauso ein Ausdruck ist wie eine Zeichenkette. In diesem Abschnitt wollen wir uns die Rückgabewerte von Funktionen genauer ansehen.

Beachten Sie zusätzlich die Hinweise zu Anfang des Kapitels bezüglich der Parameter-Modi, die z.B. für den Einsatz von PL/SQL-Funktionen direkt in SQL-Befehlen von Bedeutung sind.

Gültige Rückgabewerte bzw. -ausdrücke

`RETURN ausdruck` meint, dass man sämtliche gültigen Ausdrücke mit Hilfe einer `RETURN`-Klausel zurückliefern kann. Dies beschränkt sich gerade nicht auf Variablen, sondern erlaubt auch die Verwendung von Zeichenketten oder sogar den Einsatz von SQL-Funktionen. Dies zeigt das nächste Beispiel, in dem der Tagessatz des teuersten Dozenten zurückgeliefert wird. Sollte ein Thema nicht vergeben sein oder auch eine Kursnummer falsch eingegeben werden, so erhält man natürlich keinen passenden Tagessatz zurück, sondern den Wert `NULL`. Dies wird in einer Ausnahme vermerkt, die dann eine Fehlermeldung zurückgibt. Da der Rückgabewert den Datentyp `VARCHAR2` hat, können sowohl Zahlen als auch Zeichenketten übermittelt werden. Darauf muss das aufrufende Programm dann natürlich in jedem Fall vorbereitet sein, um keine Wertefehler zu erzeugen.

```
CREATE OR REPLACE FUNCTION FindeTeuerstenDozenten  (
 Kursnummer IN kurs.K_Nr%TYPE )
RETURN VARCHAR2
IS
 Tagessatz NUMBER;
 e_KeinDozent EXCEPTION;
BEGIN
 SELECT MAX(TH_Tagessatz)
   INTO Tagessatz
   FROM dozent NATURAL JOIN themenverteilung
  WHERE K_Nr= Kursnummer;
 IF Tagessatz IS NULL
  THEN RAISE e_KeinDozent;
 END IF;
 RETURN Tagessatz;
EXCEPTION
 WHEN e_KeinDozent
  THEN RETURN UPPER('Kein') || ' Dozent vorhanden!';
END FindeTeuerstenDozenten;
```

Listing 5.13 526_01.sql: Mögliche Ausdrücke in der RETURN-Klausel

Ein Aufruf mit einem gültigen und einem ungültigen Kurs (siehe *626_01.sql*) liefert dann folgende Ausgabe. Für den gültigen Kurs liefert die Funktion den passenden Tagessatz zurück, während sie bei einem ungültigen Kurs auf den Wert NULL stößt und dadurch die Ausnahme auslöst, die folgende Warnung anzeigt:


```
400
KEIN Dozent vorhanden!
PL/SQL-Prozedur wurde erfolgreich abgeschlossen.
```

Mehrere Rückgabewerte durch Fallunterscheidungen

Insbesondere bei der Verwendung von Ausnahmen haben Sie bereits gesehen, dass nicht die genaue Anzahl von RETURN-Klauseln für die Bestimmung wichtig ist, ob die Funktion korrekt oder nicht korrekt geschrieben ist. Vielmehr ist wichtig, dass nur eine einzige RETURN-Klausel möglich ist. Sobald eine Ausnahme ausgelöst wird bzw. sobald eine Funktion überhaupt eine Ausnahme inklusive RETURN-Klausel besitzt, besteht kein Zweifel daran, dass tatsächlich nur eine von beiden ausgewählt wird, wenn sowohl im Ausführungsabschnitt als auch im Ausnahmeabschnitt sichergestellt ist, dass durch die Fallkonstruktionen nur immer genau ein Pfad zu einer Rückgabeanweisung möglich ist. Innerhalb von beiden Abschnitten kann es dann über geeignete Fallunterscheidungen jeweils mehrere Rückgaben geben, wie folgendes Beispiel für den Ausführungsabschnitt zeigt.

Das Beispiel zählt die Dozenten, die einen Kurs unterrichten, wobei hier nicht die Zahl interessant ist (dies wäre ja auch nur eine einzige Rückgabe über die Zahl inklusive 0), sondern eine Klassifizierung der Themenverteilung in drei Gruppen.

```
CREATE OR REPLACE FUNCTION ZaehleDozenten  (
 Kursnummer IN kurs.K_Nr%TYPE )
RETURN VARCHAR2
IS
 TYPE DNummern
  IS TABLE OF dozent.D_Nr%TYPE;       -- Typ D-Nr
 t_DNr   DNummern;                    -- Tabelle D-Nr
 CURSOR c_DDaten IS
  SELECT D_Nr
    FROM dozent NATURAL JOIN themenverteilung
   WHERE K_Nr= Kursnummer;
BEGIN
 OPEN c_DDaten;
```

```
 FETCH c_DDaten BULK COLLECT INTO t_DNr;
  -- Fallunterscheidung und Rückgabe
  CASE
   WHEN t_DNr.COUNT = 0
    THEN RETURN '0';
   WHEN t_DNr.COUNT BETWEEN 1 AND 3
    THEN RETURN 'Wenig';
   WHEN t_DNr.COUNT > 3
    THEN RETURN 'Genug';
  END CASE;
 CLOSE c_DDaten;
END ZaehleDOzenten;
```

Listing 5.14 526_02.sql: Mehrere Rückgaben mit Hilfe einer Fallunterscheidung

Für drei geeignete Kurse (siehe hier *626_02.sql*) erhält man z. B. bei dreifachem Aufruf dieser Funktion folgendes Ergebnis:

```
0
Wenig
Genug
PL/SQL-Prozedur wurde erfolgreich abgeschlossen.
```

Mehrere Rückgabewerte als fehlerhafte Deklaration

Unter Umständen merkt man es gar nicht, wenn man trotz aller Warnungen dennoch mehrere Rückgabewerte konstruiert oder Fallunterscheidungen trifft, die sich überschneidende Fälle haben und daher nur in einer bestimmten Reihenfolge komplett funktionstüchtig sind. Beim Speichervorgang einer Funktion überprüft niemand außer Ihnen selbst, ob eine Funktion eventuell mehrere Rückgabewerte hat. Dies hat auch ein gewisses System, denn die Rückgabe bezieht sich immer auf das erste passende Vorkommen einer `RETURN`-Klausel. In einer korrekten Fallunterscheidung wird – wie ansonsten beim Erreichen einer `RETURN`-Klausel auch – die Abarbeitung des Ausführungsabschnitts beendet. Sollte also im vorherigen Beispiel mit der Dozentenanzahl noch ein bedeutsamer Arbeitsprozess ablaufen, so wird dieser niemals ausgeführt, wenn dieser Bearbeitungsprozess nach der `CASE`-Anweisung platziert wurde. Funktionen können praktisch gar nicht schnell genug mit ihrer Arbeit zu Ende sein. Sie sind in diesem Sinne also sehr faul und arbeitsscheu, was bei der Anwendungsentwicklung natürlich berücksichtigt werden muss. Daher sollte auch – sofern keine Fallunterscheidungen integriert werden – eine `RETURN`-Anweisung die letzte Anweisung in einer Funktion sein.

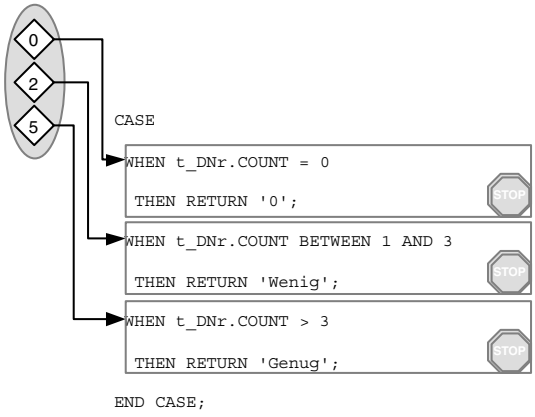


Abbildung 5.12 Singuläre Pfade bei Fallunterscheidungen

Die folgenden zwei kurzen Beispiele sollen zeigen, wie mit Situationen umgegangen wird, in denen theoretisch oder auf den ersten Blick im Quelltext mehrere Rückgabeanweisungen verzeichnet sind. Dazu konstruiert man zunächst eine Schleife, die zwar nicht unendlich läuft, die aber die Funktion schon kräftig durcheinander wirbeln sollte.

```
CREATE OR REPLACE FUNCTION EndlosRETURN
RETURN NUMBER
IS
BEGIN
 FOR i IN 1..10 LOOP
  RETURN i;
 END LOOP;
END EndlosRETURN;
```

Listing 5.15 526_03.sql: Mehrere RETURN-Anweisungen durch Schleife

Im zweiten Beispiel ist man besonders dreist und setzt eiskalt zwei Rückgabeanweisungen auf gleicher Verschachtelungshöhe im Quelltext fest.

```
CREATE OR REPLACE FUNCTION MehrereRETURN
RETURN NUMBER
IS
BEGIN
 RETURN '1';
 RETURN '2';
END MehrereRETURN;
```

Listing 5.16 526_04.sql: Mehrere Rückgaben auf gleicher Höhe

Da im Zusammenhang mit Rückgabeanweisungen der alte Spruch »Wer zuerst kommt, mahlt zuerst« gilt, erhält man – zunächst interessanterweise, dann logischerweise – jedes Mal die gleiche Ausgabe. Da die Funktion sofort bei der ersten Iteration (siehe *626_03.sql*) bzw. beim ersten Auftreten der Rückgabeanweisung (siehe *626_04.sql*), die beide jeweils eine `1` auswerfen, verlassen wird, erhält man das gleiche Ergebnis:

```
1
PL/SQL-Prozedur wurde erfolgreich abgeschlossen.
```

Fehlender Rückgabewert

Typischerweise ist man während der Arbeit nicht so dusselig, eine Funktion zu planen und dann ihren Rückgabewert zu vergessen. Aber es kann durchaus der Fall eintreten, dass kein Fall eintritt, nämlich keiner aus der entscheidenden Fallunterscheidung. In diesem Fall, der dann eintritt, handelt es sich nämlich um einen ohne Fall und damit ohne Rückgabeanweisung. Im Beispiel wollen wir es gar nicht so kompliziert machen, sondern verzichten komplett auf die `RETURN`-Anweisung, was definitiv verhindert, dass sie einen Wert zurückliefern kann.

```
CREATE OR REPLACE FUNCTION KeinRETURN
RETURN NUMBER
IS
BEGIN
 NULL;
END KeinRETURN;
```

Listing 5.17 526_05.sql: Fehlende RETURN-Anweisung

Wenn auch diese unnütze Funktion ohne Schwierigkeiten in der Datenbank gespeichert wird, erhält man doch bei ihrem Aufruf eine von den Fehlermeldungen, die länger sind als das fehlerhafte, kurze Programm. Es handelt sich dabei um den Fehlerwert 06503, der genau für diesen Fall – kein Rückgabewert vorhanden – ausgeworfen wird.

```
DECLARE
*
FEHLER in Zeile 1:
ORA-06503: PL/SQL: Funktion hat keinen Wert zurückgegeben
ORA-06512: in "SCOTT.KEINRETURN", Zeile 5
ORA-06512: in Zeile 3
```

5.2.7 PL/SQL-Funktionen in SQL

Ganz zu Anfang des Kapitels haben wir im Zusammenhang mit den Parameter-Modi für Funktionen herausgefunden, dass in SQL die Möglichkeit besteht, PL/SQL-Funktionen aufrufen, wenn sie gewissen Bedingungen genügen. Eine der ersten Bedingungen stellte die Wahl der Parameter-Modi dar. Es waren nur `IN`-Parameter erlaubt, um eine Fehlermeldung zu vermeiden, die auf die `OUT`-Parameter verwies. Dies ist eine relativ neue Eigenschaft von Oracle, die es ermöglicht, Geschäftsregeln direkt in Funktionen unterzubringen und diese dann in SQL aufzurufen. Bisher hatten wir uns stets damit gequält, für den Eintrag in eine Tabelle zunächst den höchsten Wert der Primärschlüsselspalte zu ermitteln (nur bei aufsteigenden, durchnummerierten Schlüsselwerten möglich) und dann passend über Zählervariablen oder PL/SQL-Tabellen neue, sich an den höchsten Wert anschließende Schlüsselwerte zu errechnen. Alternativ kann man hier auch eine Sequenz benutzen, doch lässt sich dieses Problem ebenso einfach über eine Funktion durchführen, die jeweils den höchsten Schlüsselwert ermittelt. Der Vorteil dieser Technik besteht also ganz einfach darin, dass man sich Arbeit in SQL spart, da man sie bereits in PL/SQL hinter sich gebracht hat.

Bedingungen für verwendbare PL/SQL-Funktionen

Folgende Eigenschaften müssen PL/SQL-Funktionen erfüllen, damit sie in SQL zum Einsatz kommen können:

- Die Funktion muss in der Datenbank und nicht in anderen Oracle-Produkten gespeichert sein. Sie können also nicht auf Funktionen speichern, die Sie nur in Anwendungen von *Oracle Forms* verwenden und die nur dort gespeichert sind.
- Die Parameter-Modi können nur `IN` sein.
- Die Datentypen, die von der Funktion zurückgeliefert werden, müssen den Datenbank-Datentypen entsprechen und dürfen nicht (wie z.B. `BOOLEAN`, `BINARY_INTEGER`, Datensätze, Sammlungen oder spezielle Subtypen) nur in PL/SQL existieren.
- Die Ergebnisse der Funktion müssen sich auf Zeilen und nicht auf ganze Spalten oder Aggregate auswirken. Es kann sich also nicht um gruppenbezogene Funktionen handeln.

Einschränkungen und Gefahren

Aus den Erfahrungen des letzten Kapitels haben Sie sicherlich auch schon gesehen, dass nahezu alle Aktivitäten der autonomen Blöcke, mit denen wir die

Syntax von PL/SQL vorgestellt haben, auch in Module umsetzbar sind. Dies ist ja auch logisch bzw. sinnvoll, da sie nur ständig und für größere Anwendungen zur Verfügung stehen. Dies bedeutet allerdings auch, dass innerhalb einer Funktion eine Menge Quelltext platziert werden kann, der für die Rückgabe eines Wertes neben diversen Abfragen auch Datenänderungen vornimmt.

So könnte eine Funktion, die innerhalb einer Schleife aufgerufen wird und für die Verarbeitung eines Cursors Daten verändern, die in der Abfrage auftauchen und damit entsprechende Inkonsistenzen, Anomalien oder Schwierigkeiten bei der Transaktionsverwaltung entstehen lassen. Um solche, so genannte »Seiteneffekte« zu vermeiden, lassen sich folgende Einschränkungen für die Aktivitäten von Funktionen festlegen:

- Es dürfen keine Datenänderungen über `INSERT`, `UPDATE` oder `DELETE` ausgeführt werden.
- Package-Variablen dürfen nicht angesprochen werden, solange dies nicht über die Befehle und Klauseln `SELECT`, `VALUES` oder `SET` geschieht.
- Nicht alle mitgelieferten Pakete lassen sich für die Anweisungen eine Funktion verwenden.
- Die Funktion kann nur andere Funktionen aufrufen, die die Einschränkungen berücksichtigen.

Geeignete Funktionen und Anwendungsgebiete

Die Verwendung von PL/SQL-Funktionen ist – soweit die obigen Hinweise bei der Planung der Funktion und ihrer Einbettung in SQL Berücksichtigung finden – nicht schwerer als bei herkömmlichen SQL-Funktionen auch. Einige Beispiele sollen dies verdeutlichen, wobei auch unterschiedliche Anwendungsgebiete dieses Themas gezeigt werden.

Berechnung von Aggregatwerten

Mit gewöhnlichem SQL kommt man schnell an die Grenzen des Machbaren, sobald aggregierte Werte in Abfragen verwendet werden sollen oder sobald Übersichtsergebnisse ausgegeben werden sollen. Dazu gibt es zwar auch in vielen Fällen die Möglichkeit, sich mit den Analyse-Befehlen aus dem Data-Warehouse-Bereich zu behelfen, aber man kann auch eigene Funktionen verwenden und leichtere Abfragen erstellen. Nehmen wir an, wir möchten (müssen) herausfinden, welche Kurse eigentlich über- oder unterdurchschnittlich lange dauern. Dabei möchten wir nicht nur; wegen »sondern auch gleich« weiter unten wissen, welche Kurse überhaupt die `AVG(K_Dauer)`-Grenze über- oder unterschreiten, sondern auch gleich noch eine Berechnung der Differenz erhalten. Während die gerade ausgeschlossene Abfrage einfach mit Hilfe einer einfachen

WHERE-Bedingung die Durchschnittsdauer mit den diversen Vergleichsoperatoren zu formulieren ist, ermittelt eine Abfrage der Form

```
SELECT K_Titel, K_Untertitel, K_Dauer-AVG(K_Dauer)
 FROM kurs
 GROUP BY K_Titel, K_Untertitel, K_Dauer;
```


keine akzeptablen Ergebnisse, da durch die Gruppierung und die Bildung des Aggregats leider für jede Zeile die durchschnittliche Kursdauer mit ihrer eigenen ermittelt wird und dadurch im Ergebnis

```
K_TITEL           K_UNTERTITEL                    K_DAUER-AVG(K_DAUER)
------------- ---------------------------------- --------------
ASP               XML-Konzepte                                    0
C++               COM/ATL                                         0
C++               Multithreading                                  0
C++               Syntax + Konzepte                               0
C++               MFC (Windows-Programmierung)                    0
```

Listing 5.18 527.01.sql: Abfrage und Ergebnis

überall der Wert 0 für die angebliche Abweichung zwischen allgemeinem Durchschnitt und spezieller Kursdauer auftritt.

Eine allgemeine Funktion für die Berechnung des Durchschnitts für alle möglichen Tabellen und Spalten könnte hier über dynamisches SQL als Eingangsparameter eine Zeichenkette für Tabellen- und Spaltennamen erwarten. Diese setzt man dann in einer geeigneten Verkettung zu einer einfachen Durchschnittsermittlung zusammen und ermittelt mit ihr für die gegebenen Parameter den gesuchten Durchschnittswert.

```
CREATE OR REPLACE FUNCTION BerechneDurchschnitt
 ( Tabelle IN  VARCHAR2,
   Spalte  IN  VARCHAR2 )
 RETURN NUMBER
IS
 v_SQL    VARCHAR2(300);
 v_Return NUMBER(5);
```

```
BEGIN
  v_SQL := 'SELECT AVG('
           || Spalte
           || ') FROM '
           || Tabelle;
 EXECUTE IMMEDIATE v_SQL
 INTO v_Return;
 RETURN v_Return;
END BerechneDurchschnitt;
```

Listing 5.19 527_01.sql: Funktion zur Berechnung von Durchschnitten

Diese Funktion webt man nun in die entsprechende Abfrage ein, wobei in *SQL*Plus* nicht einmal ein anonymer Block erwartet wird, da ja auch alle anderen (SQL-)Funktionen in Ad-hoc-Abfragen hervorragend funktionieren.

```
SELECT K_Titel,
       K_Untertitel,
       K_Dauer-BerechneDurchschnitt('kurs','K_Dauer') AS AVG
  FROM kurs;
```

Dadurch erhält man dann die benötigte, interessante Übersicht:

```
K_TITEL             K_UNTERTITEL                        AVG
------------------ ----------------------------------- ------
Perl               Module programmieren in Perl         -1
Perl               Programmierung von GUIs mit Perl/Tk  -2
Perl               Netzwerkprogrammierung               -1
Perl               Systemadministration                 -1
Perl               Datenbankprogrammierung              -2
Photoshop          Bildbearbeitung                       1
Photoshop          Korrektur + Verfremdung               0
PHP                Syntax + Konzepte                     0
```

Listing 5.20 527_01.sql: Abfrage mit einer PL/SQL-Funktion in SQL und Ergebnis

Wiederholende Berechnungen

Zwar ändert sich die Mehrwertsteuer (aus dem Schulbeispiel zur Demonstration der Rechenkünste von SQL auf Preisspalten) nur selten innerhalb eines Landes, aber in Anwendungen mit internationaler Ausrichtung können verschiedene Mehrwertsteuersätze auftreten, die landestypisch inklusive der passenden Wechselkurse als Abfrageergebnis in verschiedenen Anwendungen benötigt werden. Wenn man sich die Mühe macht, eine Anwendung so weit aufzubauen, dass Größen vorkommen, die sich irgendwann (bzw. beim Wech-

selkurs täglich), sollte man sie in Variablen auslagern, um sie nur an einem Ort pflegen zu müssen. In solchen Fällen lohnt es sich, gleich die gesamte Berechnung in eine Funktion auszulagern und nicht in SQL doppelt zu implementieren.

Eine solche Abfrage könnte daraus bestehen, aus vorhandenen Dollar-Preisen richtig gerundete Euro-Preise zu machen – inklusive der Berücksichtigung des tagesaktuellen Wechselkurses und der deutschen Mehrwertsteuer. Zusätzlich (es soll ja als Beispiel eine abschreckende Wirkung auf Sie haben) sollen noch Dezimalzahlen, Dezimalkommas und die Währungsbezeichnungen ausgegeben werden. Dies schafft SQL für die fünf Preisspalten in der Tabelle `PREIS` über eine verschachtelte Funktion, die für jede einzelne Spalte aufgerufen werden muss:

```
SELECT K_Nr,
       TO_CHAR(ROUND(P_TN1*0.95*1.16,0),'C99999.99') AS TN1,
       TO_CHAR(ROUND(P_TN2*0.95*1.16,0),'C99999.99') AS TN2,
       TO_CHAR(ROUND(P_TN3*0.95*1.16,0),'C99999.99') AS TN3,
       TO_CHAR(ROUND(P_TN4*0.95*1.16,0),'C99999.99') AS TN4,
       TO_CHAR(ROUND(P_TN5*0.95*1.16,0),'C99999.99') AS TN5
  FROM kurs NATURAL JOIN preis
 WHERE K_Titel = 'PHP';
```

Dies ergibt folgendes Ergebnis für die ersten vier Spalten, da leider die fünfte nicht mehr korrekt in eine Zeile passte:

```
   K_NR     TN1          TN2         TN3         TN4
-------- ----------- ---------- ---------- ----------
 1015024 EUR1014.00  EUR963.00  EUR915.00  EUR869.00
 1015043 EUR992.00   EUR942.00  EUR895.00  EUR850.00
 1015044 EUR992.00   EUR942.00  EUR895.00  EUR850.00
 1015068 EUR1295.00  EUR1230.00 EUR1168.00 EUR1110.00
```

Listing 5.21 527_02.sql: Abfrage und Ergebnis einer Berechnung

Eine gleichartige Funktion, die für eine eingehende Spalte den umgerechneten Währungsbetrag zurückliefert und diesen in die Ergebnismenge überträgt, sähe z.B. folgendermaßen aus. Dabei sind die eingehenden Parameter der Wechselkurs, die Mehrwertsteuer (alternativ auch nur als Prozentsatz, wobei dann innerhalb der Funktion eine passende Berücksichtigung der Prozentzahl erfolgt) und natürlich der Spaltenname.

```
CREATE OR REPLACE FUNCTION BerechnePreisMWST
 (
```

```
 Wechselkurs     IN NUMBER,
 Mehrwertsteuer  IN NUMBER,
 Originalwert    IN NUMBER )
 RETURN VARCHAR2
IS
BEGIN
 RETURN TO_CHAR(
         ROUND(Originalwert*Wechselkurs*Mehrwertsteuer,0)
                 ,'C99999.99');
END BerechnePreisMWST;
```

Listing 5.22 527_02.sql: Berechnung von Preisen innerhalb einer Funktion

Der Aufruf erfolgt dann mit einer kleinen Variation im Gegensatz zum ersten Beispiel, da man hier die Spalte direkt vorgibt. Dies sieht auf den ersten Blick fehlerträchtig aus, da ja hier eine Zeichenkette im Funktionsaufruf zu finden ist, diese aber weder wegen des unpassenden Datentyps erwartet wird noch durch Anführungszeichen als solche gekennzeichnet wird. Dies ist allerdings nicht notwendig, da sie hier als Bezeichner für eine tatsächlich vorhandene Spalte fungiert und damit nur die entsprechenden Werte im passenden Datentyp referenziert und in die Funktion übermittelt. Zwar haben Sie nun beim Funktionsaufruf die beiden Werte für Wechselkurs und Mehrwertsteuer von Hand eingetragen, aber Sie können sich vorstellen, dass Sie stattdessen innerhalb eines Cursors solche Werte über Binde-Variablen vorgeben, die für die gesamte Anwendung global zur Verfügung stehen.

```
SELECT K_Nr,
       BerechnePreisMWST(0.95, 1.16, P_TN1) AS TN1,
       BerechnePreisMWST(0.95, 1.16, P_TN2) AS TN2,
       BerechnePreisMWST(0.95, 1.16, P_TN3) AS TN3,
       BerechnePreisMWST(0.95, 1.16, P_TN4) AS TN4,
       BerechnePreisMWST(0.95, 1.16, P_TN5) AS TN5
  FROM kurs NATURAL JOIN preis
 WHERE K_Titel = 'PHP';
```

Listing 5.23 527_02.sql: Aufruf der Berechnungsfunktion

Korrelierte Unterabfragen

SQL ist eine durchaus komplexe Sprache, und man kann wirklich beeindruckende Ergebnisse und Abfragen ausdrücken. Dazu kann man sich entweder auf diverse Funktionen verlassen, die im jeweiligen System vorhanden sind, oder man setzt die Sprachkonstrukte von SQL umfassend ein. Eine Möglichkeit dieser Sprachkonstrukte ist die korrelierte Unterabfrage, die Mengenvergleiche

über eine umschließende und eingeschlossene Abfragemenge ausführt. Dies steht im Gegensatz zur gewöhnlichen Unterabfrage, die z.B. eine Reihe von Werten (Kursnummern, Namen etc.) für eine `IN`-Klausel besorgt und keine Rücksicht auf umgebende Werte hat, sondern lediglich dynamisch passende und aktuelle Werte in die Klausel zurückliefert.

Möchte man allerdings wissen, welche Dozenten für jedem Kurs den geringsten Tagespreis haben, beschafft man sich eine Liste der günstigsten Dozenten. Dies läuft darauf hinaus, dass die Abfrage pro Kursnummer eine eigene Abfrage absenden muss, die den günstigsten Dozenten anhand des geringsten Tagessatzes pro Kursnummer berücksichtigt. Die Formulierung »pro Kursnummer« meint nun die Korrelation zwischen beiden Abfragen, was man auch mit zwei verschachtelten Schleifen beschreiben kann.

Die Abfrage für diese interessante Liste, die auch gleichzeitig die Frage beantwortet, welche Kurse überhaupt mit Dozenten versorgt sind, lautet daher mit der Korrelationsbedingung `d.K_Nr=a.K_Nr`:

```
SELECT K_Titel, K_Untertitel, D_Nachname, TH_Tagessatz
  FROM kurs a, dozent b , themenverteilung c
 WHERE a.K_Nr=c.K_Nr
   AND b.D_Nr=c.D_Nr
   AND TH_Tagessatz = (SELECT MIN(d.TH_Tagessatz)
                       FROM themenverteilung d,
                            themenverteilung  e
                      WHERE d.TH_Nr=e.TH_Nr
                        AND d.K_Nr=e.K_Nr
                        AND d.K_Nr=a.K_Nr)
  ORDER BY D_Nachname, K_Titel;
```

Sie liefert für die ersten drei Datensätze von insgesamt 86 folgende Werte:

```
K_TITEL     K_UNTERTITEL        D_NACHNAME       H_TAGESSATZ
---------- ------------------- --------------- -------------
Acrobat     Satztechnik         Bärchen              350
Freehand    Grafikerstellung    Bärchen              350
Illustrator Grafikerstellung    Bärchen              350
```

Listing 5.24 527_03.sql: Korrelierte Unterabfrage und Ergebnis zu dem günstigsten Dozenten

Eine solche Abfrage mit weiteren Primärschlüsselfeldern, weiteren Tabellen und – im schlimmsten Fall – weiteren Korrelationen kann man auch in einer allgemeinen oder einer speziellen Funktion verbergen, die passend für die Korrelationsbedingung `d.K_Nr=a.K_Nr` als `IN`-Parameter den benötigten Wert

beschafft. Eine solche Funktion erwartet also einfach nur den passenden Wert, den man – wie im vorherigen Beispiel – als qualifizierten Spaltenwert beim Funktionsaufruf vorgibt.

```
CREATE OR REPLACE FUNCTION MINTagessatz (
 Kursnummer IN kurs.K_Nr%TYPE)
 RETURN themenverteilung.TH_Tagessatz%TYPE
IS
 v_MINTagessatz themenverteilung.TH_Tagessatz%TYPE;
BEGIN
 SELECT MIN(a.TH_Tagessatz)
   INTO v_MINTagessatz
   FROM themenverteilung a, themenverteilung b
  WHERE a.TH_Nr=b.TH_Nr
    AND a.K_Nr=b.K_Nr
    AND a.K_Nr=Kursnummer;
 RETURN v_MINTagessatz;
END MINTagessatz;
```

Listing 5.25 527_03.sql: Funktion für korrelierte Werte

Der Aufruf gestaltet sich dann folgendermaßen, wobei wie zuvor der Wert für jede Zeile mit dem Spaltenwert übergeben wird:

```
SELECT K_Titel, K_Untertitel, D_Nachname, TH_Tagessatz
  FROM kurs a, dozent b , themenverteilung c
 WHERE a.K_Nr=c.K_Nr
   AND b.D_Nr=c.D_Nr
   AND TH_Tagessatz = MINTagessatz(a.K_Nr)
 ORDER BY D_Nachname, K_Titel;
```

Aufruf von Spalten in PL/SQL-Funktionen

Im Beispiel *527_02.sql* haben Sie gesehen, wie schön man auch Spaltennamen in einem Funktionsaufruf unterbringen kann, um den benötigten Wert aus der Spalte zu erhalten. Dies funktioniert, weil der SQL-Statement-Executor zunächst in den Schema-Objekten sucht, die von einem SQL-Befehl angesprochen werden können bzw. die über den Tabellenaufruf in der `FROM`-Klausel angesprochen werden.

Folgende Fehlermeldungen geben einen Einblick in diesen Mechanismus:

- Falscher Spaltenname einer aufgerufenen Tabelle:

```
       BerechnePreisMWST(0.95, 1.16, P_TN0) AS TN1,
                                     *
FEHLER in Zeile 2:
ORA-00904: Ungültiger Spaltenname
```


- Falsche Kennzeichnung als Zeichenkette bei erwarteter Zahl:

```
       BerechnePreisMWST(0.95, 1.16, 'P_TN1') AS TN1,
                                     *
FEHLER in Zeile 2:
ORA-01722: Ungültige Zahl
```

- Eventuell falsche Berechnung bei hartkodiertem Wert, sofern nicht nur dieser als Wert ermittelt werden sollte. Im Rahmen einer Abfrage ergibt dies in jeder Ergebnisspalte den gleichen berechneten Wert.

```
BerechnePreisMWST(0.95, 1.16, 1000) AS TN1
```

- Sollten Sie zufällig in die Situation geraten, dass ein Funktionsname mit einem Spaltennamen übereinstimmt, so wird immer der Spaltenname für die Verarbeitung der Abfrage verwendet. Wenn Sie allerdings genau die Funktion ausführen wollen, dann verwenden Sie den qualifizierten Namen der Funktion, die sich zusätzlich aus dem Schema-Namen bildet, in dem die Funktion erstellt wurde: `schema.funktionsname()`.
- Für den Fall, dass – was eventuell häufiger vorkommt als die letzte Fehlerquelle – Parameternamen und Spaltennamen gleich sind, können Sie sich ebenfalls mit einem qualifizieten Namen behelfen wie z.B.: `schema.prozedurname.parameter`.

Das folgende Beispiel zeigt den gerade erwähnten Fall für eine Prozedur mit einem Ausnahmeabschnitt. Sobald ein unqualifizierter Name für den Parameter `D_Nr` verwendet wird, ist die Bedingung für alle Zeilen der Tabelle erfüllt. Dies erfordert einen Cursor und löst zunächst einmal die Ausnahme `TOO_MANY_ROWS` aus:

```
CREATE OR REPLACE PROCEDURE Dozentendaten (
  D_Nr   IN dozent.D_Nr%TYPE) -- ungünstig!
IS
  v_DDaten dozent%ROWTYPE;
BEGIN
 SELECT *
   INTO v_DDaten
```

```
   FROM dozent
  WHERE D_Nr = Scott.Dozentendaten.D_Nr; -- qualifiziert
  DBMS_OUTPUT.PUT(v_DDaten.D_Nr || ': ');
  DBMS_OUTPUT.PUT(v_DDaten.D_Anrede || ' ');
  DBMS_OUTPUT.PUT_LINE(v_DDaten.D_Nachname);
EXCEPTION
 WHEN TOO_MANY_ROWS           -- knapp vorbei
 THEN DBMS_OUTPUT.PUT_LINE('D_Nr missverständlich.');
END Dozentendaten;
```

Listing 5.26 527_04.sql: Ungünstige Parameternamen und Auswirkungen

Beim Aufruf des korrekten Programms (für den Test des inkorrekten testen Sie einfach die Abfrage mit `WHERE D_Nr = D_Nr` oder löschen die Namensqualifikation) mit

```
DECLARE
BEGIN
 Dozentendaten(12);
END;
```

Listing 5.27 527_04.sql: Aufruf der Prozedur mit qualifiziertem Namen

erhält man dann tatsächlich:

```
12: Herr Still
PL/SQL-Prozedur wurde erfolgreich abgeschlossen.
```

5.3 Eigene Prozeduren verwenden

Eine weitere Modulkonstruktion stellen Prozeduren dar, die einige Gemeinsamkeiten mit Funktionen haben, was z.B. die Parameterübergabe betrifft, die aber auch vollkommen andere Strukturen aufweisen. Während Funktionen im Rahmen eines Ausdrucks angesprochen werden, wie wir es auch gerade sehr deutlich über SQL-Befehle gesehen haben, so lassen sich Prozeduren direkt aufrufen bzw. in dynamischem SQL mit einer eigenen `CALL`-Anweisung starten. Des Weiteren führt eine Prozedur im Wesentlichen eine oder mehrere Aktionen durch und liefert einen oder mehrere »Rückgabewerte« an das aufrufende Programm zurück. Damit stellen Prozeduren im Vergleich zu Funktionen Unterprogramme im engeren Sinne des Wortes dar, weil sie einen kompletten Teil der Anwendungslogik übernehmen können. Teilweise lassen sich Funktionen und Prozeduren mit ähnlichen Eigenschaften bauen, doch bieten Prozeduren für komplexe Anweisungen eindeutig umfangreichere Möglichkeiten.

6.2 Java Stored Procedures

Die Idee ist bestechend einfach: Wenn man gespeicherte Prozeduren in PL/SQL schreiben kann, warum dann nicht auch in Java? Und so bieten die aktuellen Versionen der Oracle-Datenbank heute eine integrierte *Java Virtual Machine* (JVM), die *Oracle JVM*, über die man in Java geschriebene Programme direkt auf dem Server ausführen kann. Dabei hat man in gleicher Weise wie in PL/SQL Zugriff auf die Daten und Funktionen der Oracle-Datenbank.

Warum Java Stored Procedures?

Viele seiner Eigenschaften machen Java zu einer optimalen Sprache für Serverapplikationen.

Zunächst ist Java eine streng objektorientierte und einfache Sprache, die eine hohe Wiederverwertbarkeit von in Java geschriebenen Komponenten garantiert. Immer mehr fertige Produkte und Komponenten können leicht in eigene Anwendungen eingebunden werden. Durch die automatische Speicherverwaltung, die robuste Ausnahmebehandlung und die konfigurierbare Sicherheitsarchitektur ist es möglich, in Java geschriebene Komponenten eng mit dem Datenbankserver zu verkoppeln, ohne dessen Stabilität, Sicherheit und Zuverlässigkeit preiszugeben. Und genau diese enge Kopplung von Datenbankserver und Java Virtual Machine erlaubt den sehr effizienten Zugriff auf die Ressourcen des Datenbankservers. Darüber hinaus ermöglicht Java die transparente Einbindung von verteilten Komponenten über RMI, CORBA und ähnliche Architekturen für verteilte Systeme und verfügt über eine ständig wachsende Klassenbibliothek mit vielen nützlichen Werkzeugen und Schnittstellen.

PL/SQL vs. Java Stored Procedures

Am Anfang war SQL. Aber SQL war nicht gut genug. Denn SQL ist eine rein deskriptive Sprache zur Beschreibung von gewünschten Ergebnismengen, die dann vom Datenbankserver geliefert werden sollen. Leider sind die Probleme, für die heute Lösungen gesucht werden, beileibe nicht so gut klassifizierbar, als dass wir mit einer einfachen Beschreibung der Problematik ein fertiges Programm definieren können. Wer heute Lösungen programmiert, muss wohl oder übel prozedural oder objektorientiert programmieren und nicht deskriptiv.

Sehr bald gab es dann auch Erweiterungen des Standards SQL, die prozedurale Elemente in den Sprachumfang von SQL übernahmen. In diesem Prozess entwickelte Oracle PL/SQL zu einer umfangreichen prozeduralen Programmiersprache mit objektorientierten Merkmalen, die sich nahtlos in deskriptives SQL

integriert. Weil aber PL/SQL von einer sehr speziellen Umgebung ausgeht, und zwar von einer Oracle-Datenbank, ist PL/SQL eine Art Nischensprache geblieben. Es gibt keine Spiele, Office-Pakete oder Webserver, die in PL/SQL programmiert sind. (Zumindest sind den Autoren dieses Buches noch keine bekannt geworden.)

Ein Entwickler, der serverseitige Komponenten in Oracle entwickeln möchte, muss also erst einmal PL/SQL lernen. Das ist zunächst unproduktiv, denn alles, was man mit PL/SQL machen kann, kann man in der Regel mit einer normalen Programmiersprache auch, nur die Syntax ist eben anders. Nach ein paar Monaten, vielleicht nach Abschluss des Projekts, stellt man dann fest, dass PL/SQL tatsächlich besser geeignet ist, um gespeicherte Prozeduren für Oracle zu implementieren. Aber vielleicht auch nicht. Dazu gibt es dann auch den viel zitierten Spruch: Zeit ist Geld.

In den letzten Jahren kam dementsprechend der Trend auf, zur Programmierung serverseitiger Komponenten gängige Programmiersprachen zu nutzen. So gibt es einige Datenbankserver, die Perl oder Python zur Programmierung von gespeicherten Prozeduren unterstützen. Doch erst Java verhalf diesem Trend zum Durchbruch. Zwar ist kaum eine andere Sprache so langsam wie Java, aber Sun hat von Anfang an Java für den Einsatz im Server- und Netzwerkbereich entwickelt und nicht als die Lösung aller Performance-Probleme. Doch selbst dieses Argument ist nicht wirklich hieb- und stichfest, denn inzwischen liegen die Geschwindigkeitsunterschiede zwischen dem PL/SQL-Interpreter und der Oracle Java Virtual Machine im Bereich des statistischen Rauschens.

Ein weiterer Vorteil beim Einsatz von Java ist die Tatsache, dass sich die in Java geschriebenen Komponenten mit wenig Aufwand portieren lassen. Sie werden sehen, dass die Beispiele in diesem Kapitel in gleicher Weise sowohl als Client- als auch als Serverapplikation funktionieren. Wenn man in den Java-Programmen eine Abstraktionsschicht wie JDBC nutzt, kann sogar der Datenbankserver, der die gespeicherte Prozedur zur Verfügung stellt, mit minimalem Portierungsaufwand ausgetauscht werden.

Sind bereits einige Anwendungen in Java geschrieben? Binden Sie den geschriebenen Code einfach in Ihre serverseitige Komponente ein! Der wesentliche Unterschied zwischen einer lokalen JVM und der JVM des Oracle-Servers ist, dass die Ausgaben in der Trace-Datei der Oracle-Datenbank landen. Und darüber hinaus ein paar vom Datenbankadministrator konfigurierbare Sicherheitsrichtlinien, wie zum Beispiel ein beschränkter Zugriff auf das Dateisystem, oder dem Verbot Verbindungen zu anderen Datenbanken und Systemen herzustellen, an die sich der serverseitig ausgeführte Java-Bytecode halten muss. Dafür wird Ihr Java-Code direkt auf dem Oracle-Server ausgeführt. Der Zugriff

auf die Daten Ihrer Datenbank erfolgt über ein internes Speicherinterface. Wo bisher aufwändige Netzwerkverbindungen viele Schichten über JDBC, TCP/IP und Ethernet durchlaufen haben, arbeiten Sie mit den Daten lokal im gleichen Speicher. Geben Sie Ihrem bestehenden Java-Code einen ordentlichen Kick in Sachen Ausführungsgeschwindigkeit, in dem Sie die Klassen einfach in die Oracle-Datenbank importieren, statt als getrennte Client-Programme laufen zu lassen.

Und wenn Sie an die zukünftigen Nutzer, Verwalter und Weiterentwickler Ihrer Applikation denken: Eine zunehmende Zahl von Entwicklern kann Java programmieren, fast jede Hochschule hat Java in den Kanon der gelehrten Programmiersprachen aufgenommen. Eine Fülle von Programmierleitfäden, Büchern und Hilfeforen sind für Fragen rund um Java verfügbar.

Wer allerdings einmal PL/SQL ausprobiert und herausgefunden hat, dass zehn Zeilen Java im Durchschnitt etwa zwei Zeilen PL/SQL entsprechen, den wird man erst wieder mühsam von Javas Vorteilen überzeugen müssen. Wenn die Entwicklungsdauer zum kritischen Argument wird, PL/SQL-Know-how vorhanden ist und Portierbarkeit keine Rolle spielt, dann ist PL/SQL natürlich weiterhin die erste Wahl. An einem bestimmten Punkt vermischen sich die beiden Sprachen – spätestens, wenn eine Java Stored Procedure einen PL/SQL-Block oder eine gespeicherte Prozedur in PL/SQL aufruft, und erst recht, wenn ein PL/SQL-Block eine Java-Stored Procedure aufruft. Da sollte man als Architekt einer Datenbankanwendung vorsichtig planen, wenn man den Überblick über die Abhängigkeiten und Verknüpfungen behalten will. Auch der transparente Übergang des Java-Codes von draußen vor dem Server nach drinnen in den Server birgt seine kleinen Tücken und Denkaufgaben. Denn die gleichen Programme funktionieren sowohl als Client-Programme als auch als serverseitige Komponenten – die Umgebungen jedoch unterscheiden sich natürlich.

Die Oracle-JVM

Die JVM des Oracle-Servers ist eine vollständige, Java 2-kompatible Laufzeitumgebung für Java-Bytecode. Die JVM läuft im gleichen Prozess und Speicher wie der Kern der Oracle-Datenbank. Der Zugriff auf die Daten in der Datenbank erfolgt also lokal, im gleichen Speicher. Dieses integrierte Design garantiert den serverseitigen Java-Komponenten eine optimale Zugriffs- und Ausführungsgeschwindigkeit.

Die verwendete Version von JDK und JRE liegt im Pfad *jdk* des Oracle-Installationspfads. Da bei der Oracle-Datenbank die Stabilität im Vordergrund steht und jedes Update des JDK zunächst ausgiebig vorbereitet und getestet werden muss, verwendet der Oracle-Server in der Regel ein etwas älteres JDK.

Im Gegensatz zu einer lokalen JVM, bei der der Einstiegspunkt eines Java-Programms die statische Methode `main` ist, werden die Methoden der auf dem Server eingespielten Klassen direkt im Rahmen einer SQL-Anweisung aufgerufen. Dabei wird durch einen Aufruf die Klasse aktiviert und bei Bedarf, z.B. nach dem Ende einer Transaktion oder nach dem Ablauf einer bestimmten Zeitspanne, wieder deaktiviert.

Eine grafische Oberfläche ist auf dem Server nicht verfügbar. Auch wenn Sie Instanzen der Klassen aus den Paketen `java.awt` und `javax.swing` erzeugen und nutzen können, so wird der Versuch, eine grafische Oberfläche sichtbar zu machen, ausnahmslos mit einer Ausnahme quittiert.

Die Oracle JVM sollte hauptsächlich für die Ausführung fertiger Komponenten genutzt werden. Sie ist nicht besonders gut als Entwicklungsumgebung geeignet, da bei jeder Änderung die Klassen wieder auf den Server geladen werden müssen und die Ausgaben in der Trace-Datei des Oracle-Servers verschwinden. Da die in Java geschriebenen Programme in gleicher Weise lokal wie auch auf dem Server funktionieren, entwickeln und testen Sie Ihre Anwendung zunächst lokal, um sie dann, nach Abschluss der Entwicklung, auf den Server zu spielen.

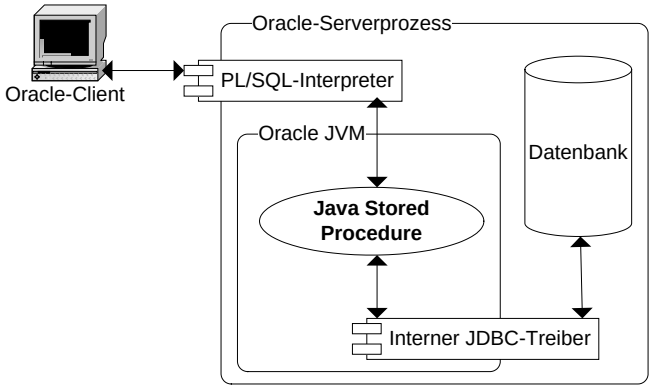


Abbildung 6.2 Die Oracle Virtual Machine und Java Stored Procedures

Entwicklungsumgebung

Da wir in den Beispielen zum Zugriff auf die Datenbank das in diesem Buch vorgestellte SQLJ verwenden, gelten hier die gleichen Anweisungen zur Entwicklungsumgebung wie im Kapitel zu SQLJ. Da die aktuelle Version der SQLJ-Implementierung von Oracle auch eine aktuelle Version des JDK voraussetzt, der Oracle-Server aber in der Regel eine etwas ältere Version von JDK und JRE nutzt, müssen Sie darauf achten, dass Ihr Quelltext mit der vom Server verwendeten Version kompatibel ist. Am besten stellen Sie die Umgebungsvariable

PATH so ein, dass Sie für die Entwicklung die gleiche Version vom JDK und JRE verwenden wie der Oracle-Server.

Die optimale Entwicklungsumgebung für Java Stored Procedures ist Oracles JDeveloper. Dieses Werkzeug bietet umfassende Unterstützung für die Entwicklung von gespeicherten Prozeduren in Java.

6.2.1 Theorie: Java und der Oracle-Server

Bevor wir die erste lauffähige Java Stored Procedure über das *SQL*Plus Worksheet* ausführen, sind einige Schritte nachzuvollziehen. So müssen wir einige Details zur Umgebung erwähnen, in der die Java-Klassen ausgeführt werden. Dann gibt es Kommandozeilenwerkzeuge zum Laden und Entladen von Klassen und Ressourcen, nämlich `loadjava` und `dropjava`. Und wenn Sie nun die Klassen geladen haben, wie können Sie feststellen, ob alles wie gewollt funktioniert und die Klassen zur Verfügung stehen? Bevor Sie eine Java Stored Procedure nun mit dem *SQL*Plus Worksheet* ausführen können, muss diese nämlich zuerst für den Aufruf aus einer PL/SQL-Anweisung registriert werden. Das sind die Themen des kommenden Abschnitts.

Unterschiede zu Clientanwendungen

Um eine Java-Klasse auf der Oracle-Datenbank zu verwenden, muss diese erst einmal auf diesem gespeichert werden. Das Tool `loadjava` dient dazu, eine oder mehrere Java-Klassen auf dem Oracle-Server zu speichern. Einmal auf dem Server werden die Java-Klassen in einer speziellen Umgebung ausgeführt, auf deren Randbedingungen wir im Folgenden eingehen.

- **Datenbankverbindung**
  Wenn Sie Java Stored Procedures mit SQLJ programmieren, stellt die SQLJ-Implementierung automatisch einen Verbindungskontext her. Die Verbindung zur lokalen Datenbank wird also implizit erzeugt, ohne dass eine weitere Anweisung notwendig ist. Wenn Sie die Methode `connect` von `oracle.sqlj.runtime.Oracle` für die Verbindung aufrufen, so wird der Aufruf von der serverseitigen SQLJ-Implementierung schlichtweg ignoriert. Sie können über den Thin-JDBC-Treiber auf dem Datenbankserver über `DefaultContext` auch Verbindungen zu anderen Oracle-Datenbanken herstellen. Dazu brauchen Sie aber spezielle Berechtigungen zum Herstellen von Verbindungen zu anderen Hosts, die Ihnen vom Systemadministrator erteilt werden müssen. Die Verbindung zum lokalen Datenbankserver kann von einer Java Stored Procedure nicht geschlossen werden. Wenn Sie also auf einer Instanz von `java.sql.Connection` die Methode `close` aufrufen, so bleibt dieser Aufruf ohne Folgen.

- **Iteratoren und reservierte Cursor**
  Wenn Sie in Ihrem Programm Iteratoren einsetzen, sollten Sie mehr als bei einer clientseitigen Applikation darauf achten, diese auch mit `close` wieder zu schließen. Ein von einem Iterator reservierter Datenbank-Cursor wird erst nach dem Aufruf der Methode `close` freigegeben, oder viel später während der automatischen Entfernung nicht mehr verwendeter Iteratorobjekte durch die automatische Speicherverwaltung des *Garbage Collector* der JVM (»Finalisierung«). Da eine Java Stored Procedure eine erheblich längere Laufzeit als eine einfache Clientanwendung haben kann, ist sie in der Lage, erhebliche Mengen von Datenbank-Cursorn für sich zu vereinnahmen – bis zu dem Punkt, dass alle verfügbaren Cursor belegt sind und eine Ausnahme auftritt.

- **Transaktionsmanagement**
  Bei einem serverseitigen JDBC-Treiber sind Sie als Programmierer für das Transaktionsmanagement verantwortlich. Die Einstellung `auto-commit` wird nicht unterstützt. Sie müssen also auf jeden Fall selbst Transaktionen mit `COMMIT` abschließen oder mit `ROLLBACK` abbrechen.

- **Aufruf von Java Stored Procedures**
  Der Einstiegspunkt für Java Stored Procedures sind immer Methoden vom Typ `public static`. Ihre Klasse oder Klassenbibliothek sollte also eine solche Methode, analog zur Methode `main` in eigenständigen Java-Anwendungen, implementieren. Eine Java Stored Procedure kennt, wie ihre statische Einstiegsmethode, keine Sitzungen im Sinne einer *Session* aus der Internetwelt. Jeder Aufruf erfolgt anonym und gleichberechtigt neben allen anderen Aufrufen.

- **Threadsicherheit**
  Ihre Einstiegsmethode wird unter Umständen von mehreren Clients parallel aufgerufen. Sie muss also threadsicher sein. Wenn Ihnen der Begriff der Threadsicherheit nicht ganz geläufig sein sollte: Seien Sie vorsichtig mit allen Variablen und Klassen, die nicht innerhalb der Einstiegsmethode deklariert wurden. Diese können dann nämlich von mehreren Aufrufen parallel verändert werden.

- **Ausgabe**
  Java Stored Procedures sind zunächst sehr stumm, was die Aufrufe von `System.out.println` angeht. Die Ausgaben landen alle in der Trace-Datei des Oracle-Servers. Wenn Sie für interessierte Nutzer Ausgaben erzeugen möchten, innerhalb einer Anwendung wie *SQL*Plus Worksheet* beispielsweise, können Sie dieses mit den aus PL/SQL bekannten Funktionen `dbms_output.enable` und `dbms_output.put_line` über einen SQLJ-Block oder einen JDBC-Aufruf machen. Alternativ können Sie über die Funktion `dbms_`

`java.set_output` die Ausgaben des Java-Programms in den Puffer `dbms_output` umleiten.

- **Entsorgung**
  Um Java-Klassen wieder vom Server zu löschen, benutzen Sie das Tool `dropjava`.

Laden von Java-Klassen mit loadjava

Mit dem Werkzeug `loadjava` können Sie Java-Ressourcendateien in die Oracle-Datenbank importieren. Dateien mit den Endungen *java* und *sqlj* werden dabei automatisch kompiliert. Dateien mit den Endungen *zip* und *jar* werden extrahiert, und die darin enthaltenen Objekte werden einzeln importiert. Kompilierte Java-Klassen sind dann in der Tabelle `USER_OBJECTS` als Objekte vom Typ `JAVA CLASS` sichtbar. Alle anderen Dateien (denn Sie können auch Konfigurationsdateien oder binäre Informationen in der Oracle-Datenbank speichern) sind dann Objekte vom Typ `JAVA RESOURCE`.

Die Syntax des Befehls `loadjava` sieht wie folgt aus:

```
loadjava -user <Nutzername>/<Passwort>[@<Datenbank>]
    [<Optionen>]
    <Dateiliste>
```

Dabei bedeuten:

- `<Nutzername>/<Passwort>`
  Die Benutzerauthentifizierung, beispielsweise Scott/Tiger
- `<Datenbank>`
  Die Oracle-SID Ihrer Datenbank
- `<Dateiliste>`
  Eine durch einfache Leerzeilen getrennte Liste mit den Dateien, die gespeichert werden soll.
- `<Optionen>`
  Eine oder mehrere von folgenden Optionen (Auszug).

Option	Beschreibung
`-debug`	Erzeugt beim Kompilieren der Dateien in `<Dateiliste>` Bytecode mit Debug-Informationen.
`-encoding`	Ermöglicht die Angabe der Kodierung der Quelltextdateien für den Java-Compiler (analog zu `javac -encoding`).

Tabelle 6.3 Optionen von loadjava

Option	Beschreibung
-force	Erzwingt das Laden der Dateien, auch bei Fehlern.
-grant <Nutzerliste>	Den in <Nutzerliste> angegebenen Datenbanknutzern wird die Berechtigung EXECUTE für die geladenen Java-Klassen eingeräumt. Die Liste wird dabei ohne Leerzeichen mit Kommata getrennt: -grant Scott,Alice,Bruce
-help	Gibt die Syntax und die komplette Liste der Optionen zu loadjava aus.
-noverify	Java-Klassen werden ohne Prüfung des Bytecodes geladen.
-oci8	Weist loadjava an, den OCI-JDBC-Treiber zu benutzen. Die Optionen –oci8 und –thin schließen sich gegenseitig aus.
-thin	Verwendet den Thin-JDBC-Treiber.
-order	Die Klassen werden in der Reihenfolge ihrer Abhängigkeiten in die Datenbank geladen.
-resolve	Es wird überprüft, ob alle zur Ausführung notwendigen Klassen geladen und verfügbar sind.
-schema <Schema>	Die Dateien werden in das angegebene Tabellenschema <Schema> gespeichert.
-verbose	Es werden ausführliche Statusmeldungen für die von loadjava ausgeführten Aktionen ausgegeben.

Tabelle 6.3 Optionen von loadjava (Forts.)

Prüfen, ob alles gut geladen wurde

Nachdem Sie mit loadjava Klassen und Dateien auf den Server gespeichert haben, möchten Sie natürlich gerne sehen, ob das Aufspielen erfolgreich verlaufen ist. Die neuen Objekte sind dabei in der Tabelle USER_OBJECTS des im Parameter –user definierten Datenbankbenutzers sichtbar. Wenn Sie also überprüfen möchten, welche Objekte verfügbar sind, können Sie das mit einer Abfrage auf dieser Tabelle machen.

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
    FROM USER_OBJECTS
    WHERE OBJECT_TYPE LIKE 'JAVA%';
```

Besonders wichtig ist die Spalte STATUS, die angibt, ob die Objekte mit Erfolg geladen wurden, also VALID sind, oder ob ein Fehler beim Laden aufgetreten ist, die Objekte also INVALID sind.

Wenn eine der Java-Klassen, die Sie über `loadjava` geladen haben, unter `STATUS` die Ausgabe `INVALID` zeigt, so müssen Sie unbedingt zunächst die Fehlerursache finden. Eine Java Stored Procedure kann nur auf der Grundlage einer erfolgreich importierten Klasse erstellt werden!

Wenn eine Klasse erfolgreich geladen wurde, die Ausgabe unter `STATUS also VALID` lautet, dann können wir nun aus dieser eine Java Stored Procedure erstellen. Ein Schritt bleibt noch: Die Einstiegsmethode muss als `PROCEDURE` oder `FUNCTION` beim Server registriert werden.

Aufrufspezifikationen erstellen

Um die Einstiegsmethode der Java Stored Procedure aus PL/SQL aufrufen zu können, muss diese über eine `CREATE PROCEDURE`- oder `CREATE FUNCTION`-Anweisung zugänglich gemacht werden. Dabei werden die Parameter für die Prozedur nach außen hin festgelegt. Nach innen werden dem Oracle-Server die bei Aufruf der Java Stored Procedure notwendigen Typkonversionen mitgeteilt, und natürlich die Methode, die aufgerufen werden soll.

Die Syntax einer Anweisung zum Erstellen einer `FUNCTION` auf der Basis einer geladenen Java-Klasse:

```
CREATE [ OR REPLACE ] FUNCTION
    <Funktionsname>( <PL/SQL-Parameterliste> )
    RETURN <PL/SQL-Typbezeichner> AS
LANGUAGE java
NAME '<Paket>.<Klasse>.<Methode>( <Java-Parameterliste> )
      return <Java-Typbezeichner>';
```

Eine `PROCEDURE` wird analog zum obigen Aufruf ohne die `RETURN`-Anweisung erzeugt.

```
CREATE [ OR REPLACE ] PROCEDURE
    <Prozedurname>( <PL/SQL-Parameterliste> ) AS
LANGUAGE java
NAME '<Paket>.<Klasse>.<Methode>( <Java-Parameterliste> )';
```

Erläuterungen zur Syntax:

- `<Funktionsname>` bzw. `<Prozedurname>`
  Ein gültiger PL/SQL-Bezeichner für ein gespeicherte Funktion bzw. Prozedur. Der Bezeichner kann, aber muss nicht, mit dem Bezeichner der Java-Methode übereinstimmen.

- `<PL/SQL-Parameterliste>`
  Eine gültige Liste von `IN`-, `OUT`- oder `INOUT`-Parametern und Typen. Die Liste muss mit den Typen der Java-Methode übereinstimmen. `OUT`- und `INOUT`-Parameter müssen speziell modelliert werden. Dazu werden wir später mehr erklären.
- `<Java-Parameterliste>`
  Die Parameterliste der angegebenen Methode.
- `<Paket>.<Klasse>.<Methode>`
  Der vollständige Bezeichner der Methode mit Klassen- und Paketnamen. Die Methode muss `public` und `static` sein.

Am besten, Sie erzeugen die Java Stored Procedure mit der Enterprise Manager-Konsole. Dort erhalten Sie ausführliche Fehlermeldungen und können den Quelltext der `CREATE`-Anweisung komfortabel bearbeiten.

Entsorgen von Java-Klassen mit dropjava

Wenn die in der Datenbank gespeicherten Java-Klassen eines Tages obsolet geworden sind, so können Sie diese mit dem Werkzeug `dropjava` wieder entfernen. Sie rufen `dropjava` dann mit den gleichen Parametern auf wie `loadjava`:

```
dropjava -user <Nutzername>/<Passwort>[@<Datenbank>]
    [<Optionen>] <Dateiliste>
```

- `<Nutzername>/<Passwort>`
  Die Authentifizierungsdaten des Nutzers.
- `<Datenbank>`
  Die Oracle-SID Ihrer Datenbank.
- `<Dateiliste>`
  Eine mit Leerzeichen getrennte Liste der Java-Ressourcen, die wieder aus der Datenbank entfernt werden sollen.
- `<Optionen>`
  Eine oder mehrere von folgenden Optionen.

Option	Beschreibung
`-oci8` oder `-thin`	Der zu verwendende JDBC-Treiber: Entweder der OCI-Treiber über `-oci8` oder der Thin-JDBC-Treiber `-thin`
`-schema`	Das Schema, in dem die Java-Objekte liegen

Tabelle 6.4 Optionen des Werkzeugs dropjava

Option	Beschreibung
`-stdout`	Die Ausgaben werden an die Standardausgabe `stdout` ausgegeben, nicht an `stderr`, die für Fehlermeldungen reserviert ist
`-verbose`	Ausführliche Ausgaben zu den durchgeführten Aktionen

Tabelle 6.4 Optionen des Werkzeugs dropjava (Forts.)

6.2.2 Erzeugen der ersten Java Stored Procedure

Im Folgenden erstellen wir das »Hallo Welt«-Beispiel für Java Stored Procedures. Wir gehen dabei Schritt für Schritt den vollständigen Weg vom SQLJ-Quelltext bis zum Aufruf der Prozedur im *SQL*Plus Worksheet*. Zunächst erstellen wir als einen Quelltext, der »Hallo Welt« in den Ausgabepuffer `dbms_output` schreibt und dazu geeignet ist, auf den Oracle-Server aufgespielt zu werden.

```
import oracle.jdbc.driver.OracleSQLException;
import java.sql.SQLException;

public class C622_1 {
  public static void halloWelt()
    throws OracleSQLException, SQLException {
      #sql {
        CALL dbms_output.enable(500)
      };
      #sql {
        CALL dbms_output.put_line('Hallo Welt')
      };
  }
}
```

Listing 6.18 C622_1.sqlj: Eine »Hallo-Welt-Procedure«

Wie Sie sehen, nehmen wir die beiden Ausnahmen `OracleSQLException` und `SQLException` in die `throws`-Klausel der Methode auf. Damit ermöglichen wir dem Oracle-Server zwischen geplanten Ausnahmen, also einer `OracleSQLException`, und ungeplanten Ausnahmen zu unterscheiden. Eine Verbindung wird in dem Quelltext nicht explizit erzeugt. Wie bereits angedeutet, wird die Verbindung von der serverseitigen SQLJ-Implementierung automatisch hergestellt. Wenn der Quelltext auch als Client funktionieren soll, fügen Sie einfach den aus dem vorigen Kapitel bekannten Aufruf der Methode `connect` der

Klasse `oracle.sqlj.runtime.Oracle` in den Quelltext ein. Dieser wird dann von der serverseitigen SQLJ-Implementierung geflissentlich ignoriert.

Laden der Klasse mit loadjava

Die erzeugte Bytecodedatei *C622_1.class* laden wir nun mit Hilfe von `loadjava` in die Datenbank ein. Wechseln Sie also in das Verzeichnis, in dem die Klasse liegt, und führen Sie `loadjava` wie unten angegeben aus.

```
loadjava -user Scott/Tiger@test -verbose -resolve C622_1.class
```

Die auf den Befehl folgende Ausgabe sollte so aussehen:

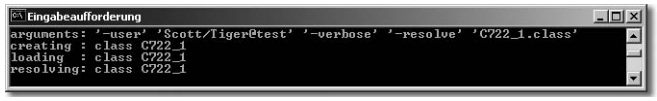


Abbildung 6.3 Ausgabe von loadjava

Dabei bedeutet das `creating` in der Ausgabe, dass die Klasse als Datenbankobjekt registriert wird. Anschließend wird die Klasse in den dafür vorgesehenen Speicher geladen: `loading`. Anschließend wird der Bytecode der Klasse auf Referenzen zu anderen Klassen überprüft, also während des Vorgangs des `resolving`, und daraufhin, ob diese Referenzen auch alle von der Oracle JVM aufgelöst werden können.

Überprüfen, ob die Klasse geladen wurde

Jetzt ist die Klasse in der Datenbank verfügbar. Oder ist sie es vielleicht nicht? Wir überprüfen also den Stand der Dinge mit der folgenden Anweisung, die die Tabelle der Datenbankobjekte des angemeldeten Benutzers abfragt. Melden Sie sich also sinnvollerweise mit dem gleichen Benutzerkonto an, das Sie auch bei `loadjava` zuvor angegeben hatten.

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
FROM USER_OBJECTS
WHERE OBJECT_NAME='C622_1'
```

Wenn die Klasse mit Erfolg geladen wurde, dann sollte die Antwort der Datenbank sinngemäß wie unten dargestellt aussehen:

```
OBJECT_NAME                    OBJECT_TYPE      STATUS
------------------------------ ---------------- -------
C622_1                         JAVA CLASS       VALID
```

Wenn unter Status `VALID` steht, wurde die Klasse erfolgreich in den Oracle-Server importiert.

Aber noch sind wir nicht ganz am Ziel, denn bis jetzt kann die Methode `C622_1.halloWelt` noch nicht über eine PL/SQL-Anweisung aufgerufen werden. Denn, wie im vorangegangenen Abschnitt angedeutet, muss man zur Java-Klasse zunächst eine Aufrufspezifikation erzeugen.

Alternativ, entsprechende Zugriffsrechte auf die Datenbank vorausgesetzt, können Sie auch mit der *Oracle Enterprise ManagerKonsole* (und zwar unter **Schema · Scott · Quelltypen · Java-Klassen)** die Liste der geladenen Java-Klassen einsehen.

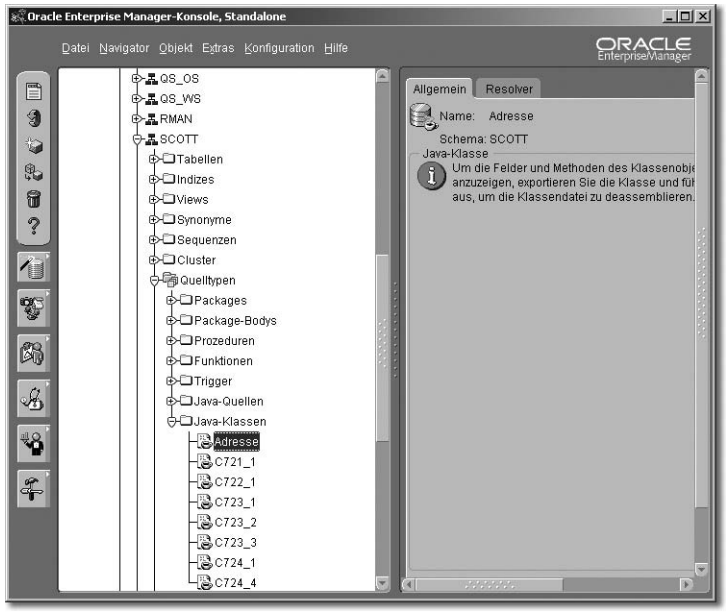


Abbildung 6.4 Verfügbare Java-Klassen in der Manager-Konsole einsehen

Gerade bei misslungenen Klassenimporten gibt die Manager-Konsole ausführlichere Informationen zu der Fehlerursache. Es steht außerdem zu erwarten, das der Import der Java-Klassen im Rahmen zukünftiger Versionen der Manager-Konsole grafisch unterstützt wird.

Erstellen der Aufrufspezifikation

Die Aufrufspezifikation ist im Fall der Klasse `C622_1` recht einfach gehalten, besitzt sie doch keine Übergabeparameter oder Rückgabewerte:

```
CREATE OR REPLACE PROCEDURE hallowelt AS
LANGUAGE java
NAME 'C622_1.halloWelt()';
```

Die Oracle Enterprise Manager Konsole, entsprechende Berechtigungen vorausgesetzt, ist mit ihrem Editor ein sehr komfortables Werkzeug zum Erstellen von Prozeduren und Funktionen, also auch geeignet für das Erstellen unserer Aufrufspezifikation.

Dazu wechseln wir in der Oracle Enterprise Manager Konsole einfach in **Schema · Scott · Quelltypen · Prozeduren** und drücken die Taste N, um eine neue Prozedur zu erstellen.

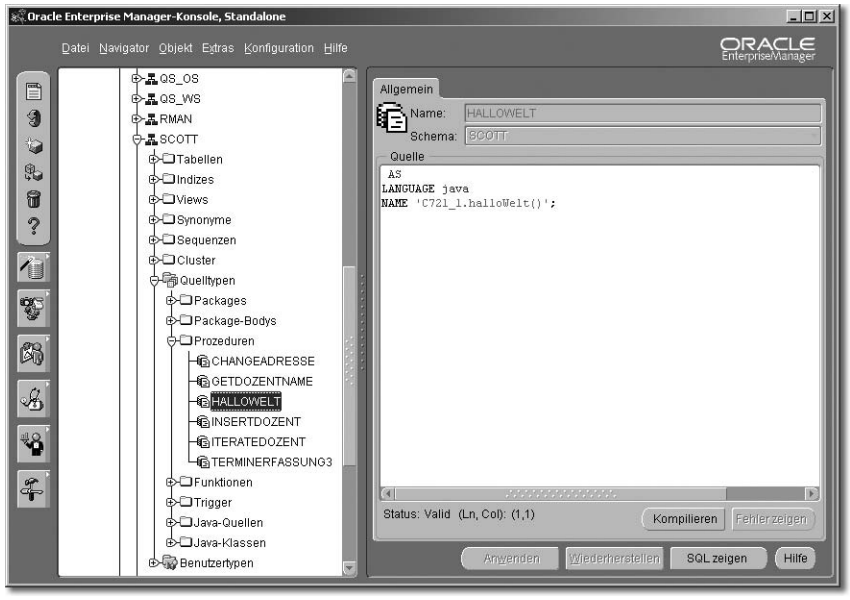


Abbildung 6.5 Erzeugen der Aufrufspezifikation für halloWelt mit OEM

Auch hier gilt analog: Die Fehlerbeschreibungen sind ungleich detaillierter als die rudimentären Ausgaben auf der Konsole.

Aufrufen der Java Stored Procedure

Wenn diese Prozedur mit Erfolg angelegt wurde, kann sie nun durch eine PL/SQL-Anweisung aufgerufen werden. Damit die Ausgabe »Hallo Welt« nicht sang- und klanglos im Puffer verschwindet, stellen Sie die Ausgabe zuvor mit

der Anweisung `SET SERVEROUTPUT ON` an. Anschließend rufen wir die erzeugte Prozedur auf:

```
CALL HALLOWELT();
```

Aufruf und Antwort sollten nun etwa wie in Abbildung 6.6 dargestellt aussehen.

Abbildung 6.6 Aufruf der Prozedur halloWelt mit SQL*Plus Worksheet

6.2.3 Übergeben von Parametern an Java Stored Procedures

Eine Java-Klasse kann erst aufgerufen werden, wenn in PL/SQL die Aufrufspezifikation definiert wurde. Eine Java Stored Procedure wird daher immer nur aus einer PL/SQL-Anweisung heraus aufgerufen werden. Das bedeutet für die Umgebung auf dem Server, dass stets die Parameter aus den internen Oracle-Datentypen in entsprechende Java- bzw. JDBC-Datentypen umgewandelt werden müssen. Wie umgewandelt werden soll, definieren Sie implizit in der `CREATE`-Anweisung, mit der Sie die Aufrufspezifikation erzeugen:

```
CREATE PROCEDURE
    beispiel(
        <param1> <Oracletyp1>,
        <param2> <Oracletyp2>
        /* evtl. weitere Parameter */ )
AS LANGUAGE java
NAME
    'Beispielklasse.eineMethode(
         <Javatyp1>,
         <Javatyp2> /* evtl. weitere Parameter */ )'
```

Wie Sie in der oben gezeigten Anweisung sehen, wird implizit eine Umwandlung von `<Oracletyp1>` in `<Javatyp1>` definiert. Dabei gilt es folgende Aspekte zu beachten:

Index

C

D

E

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z