SQL, erweiterte Möglichkeiten

5.1 Allgemeines

Im Kapitel 4 haben Sie einige Dinge über die Abfragesprache SQL erfahren. Abgesehen von wenigen Ausnahmen genügen alle dort behandelten SQL-Anweisungen der ANSI-Norm. Das bedeutet, dass diese Anweisungen in dieser Form auch von anderen Datenbanksystemen wie beispielsweise INFORMIX verstanden werden. Dies gilt selbst für lokale „Datenbanken" wie dBase, Paradox oder Access. Mit der dort dargestellten Syntax sind allerdings die Möglichkeiten, die Oracle bietet, bei weitem noch nicht ausgereizt.

Die Themen, die Sie im Verlauf dieses Kapitels kennenlernen werden, sind schon spezifischer auf die Oracle-Datenbank zugeschnitten, d.h. diese Funktionalität können Sie nicht 1:1 auf beispielsweise INFORMIX oder Microsofts SQL Server portieren. Die lokalen Datenbanken versagen gar völlig den Dienst. Trigger und Stored Procedures werden Sie beispielsweise bei dBase nicht finden.

5.2 Sequenzen

Autozähler bei Access
für jede Tabelle eine eigene Sequenz

Nehmen wir als Beispiel wieder unsere Kundentabelle. Jeder Datensatz innerhalb dieser Tabelle ist durch eine eindeutige Nummer gekennzeichnet. Sollen neue Sätze der Tabelle zugefügt werden, muss das Anwendungsprogramm nach der bisher größten Nummer suchen, 1 addieren und diesen Wert zurückschreiben, sofern nicht andere Randbedingungen berücksichtigt werden müssen. Eine Randbedingung wäre z.B. hier ein Führen getrennter Nummernkreise für verschiedene Arten von Kunden. In einem solchen Fall müsste eine neue Kundennummer nach einem anderem Algorithmus bestimmt werden. Doch zurück zur Problematik der automatischen Nummernvergabe. Bei der Entwicklung von Datenbankanwendungen wird Ihnen dieses Problem häufiger begegnen. Oracle hat das auch erkannt und mit den *Sequenzen*, eine Möglichkeit geschaffen, diesen Prozess zu automatisie-

ren. Damit haben Sie als Anwendungsentwickler die Möglichkeit, ein Stück Intelligenz aus der Anwendung in die Datenbank bzw. auf den Datenbankserver auszulagern.
Über die Anweisung

```
CREATE SEQUENCE <Sequenz-Bezeichnung>
```

wird eine solche Sequenz erzeugt. Die Abbildung 5-1 zeigt den praktischen Einsatz.

```
Oracle SQL*Plus
File Edit Search Options Help

SQL> create sequence nr;

Sequence created.

SQL> insert into kunden
  2  (kundennr, name)
  3  values
  4  (nr.nextval, 'Heitsiek');

1 row created.

SQL>
```

Abbildung 5-1: *Sequenzen erzeugen*

Gelöscht wird eine solche Sequenz wieder über die DROP-Anweisung:

```
DROP SEQUENCE NR;
```

Über die Anweisung

select nr. currval from kunde

```
<Sequenz-Bezeichnung>.nextval
```

erhält man den um „1" erhöhten Zähler,

```
<Sequenz-Bezeichnung>.currval
```

liefert den aktuellen Zähler. Das folgende Listing zeigt die Verwendung einer Sequenz beim INSERT-Befehl auf die Kundentabelle:

```
SQL> INSERT INTO kunden (kundennr, anlage)
  1  VALUES (nr.nextval, sysdate);

1 row created.

SQL>
```

Eine Selektion nach der Kundennummer wird Ihnen zeigen, dass eine „1" für die Kundennummer des letzten Datensatzes eingetragen wurde. Das Problem hierbei ist, dass sich schon Datensätze in der Kundentabelle befinden und die Kundennummer bei 100 beginnt. Bei vier vorhandenen Datensätzen wäre 105 der logisch korrekte Wert gewesen. Sie ahnen es: Man muss einer Sequenz noch bestimmte Eigenschaften mitgeben, so unter anderem

einen Startwert. Per Voreinstellung erhält eine neu eingerichtete Sequenz folgende Eigenschaften, die man natürlich noch modifizieren kann:

- der Startwert einer Sequenz ist „1";
- die Schrittgröße der Sequenz beträgt „1";
- die Sequenz wird erhöht, d.h. die nächste Nummer ist größer als die vorhergehende;
- das Maximum einer Sequenz ist erreicht bei 10^{27};
- es erfolgt bei Erreichen der Grenze eine Neuinitialisierung mit dem alten Startwert.

Im folgenden werden wir eine Sequenz für die Spalte *kundennr* der Tabelle Kunden erstellen. Der Zähler soll dabei bei 106 beginnen, mit der Schrittgröße 1 (Abbildung 5-2).

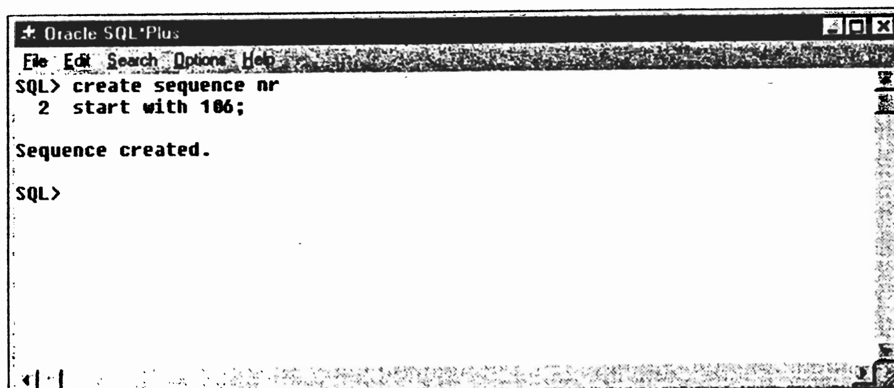

Abbildung 5-2: *Sequenz mit Startwert*

Soll zusätzlich die Schrittweite verändert werden, wird der Zusatz

```
INCREMENT BY <Nummer>
```

benötigt. Das folgende Listing zeigt eine entsprechende Anweisung:

```
SQL> CREATE SEQUENCE nr
  2  INCREMENT BY 5
  3  START WITH 105;

Sequenz wurde angelegt.

SQL>
```

Die obigen Zusätze (Zeile 2 und 3) sind ohne Zweifel die wichtigsten im Zusammenhang mit Sequenzen. Sequenzen besitzen aber noch eine Reihe weiterer Eigenschaften, die die Tabelle 5-1 auflistet. Im Prinzip können alle dort aufgeführten Anweisungen genauso eingesetzt werden, wie mit *increment by* und *start with* geschehen.

Schlüsselwort	Beschreibung
INCREMENT BY	Optionaler Parameter, der angibt, in welcher Schrittweite die Sequenz erfolgen soll. Bei positiven Werten steigt die Sequenz an, bei negativen Werten erfolgt eine absteigende Zählung. 0 ist nicht erlaubt.
MINVALUE	Definiert die Untergrenze einer Zählung. Diese Einstellung spielt nur eine Rolle, wenn es sich um eine absteigende Sequenz handelt. Logischerweise muss dieser Wert kleiner sein als die Variable MAXVALUE. MINVALUE darf außerdem nicht größer sein als der WERT, der über START WITH definiert wurde.
NOMINVALUE	Definiert, dass keine untere Grenze verwendet wird. In diesem Fall gelten die maximalen Intervallgrenzen (Voreinstellung).
MAXVALUE	Definiert eine Obergrenze für eine Sequenz. MAXVALUE muss größer als MINVALUE und größer als der mit START WITH definierte Parameter sein.
NOMAXVALUE	Gibt an, dass keine Obergrenze verwendet wird. In diesem Fall gelten die maximalen Intervallgrenzen.
START WITH	Gibt den Startwert der Sequenz an.
CYCLE	Gibt an, dass die Sequenz in einem Ringpuffer definiert ist. Bei Erreichen der Obergrenze wird damit als darauffolgender Wert die Untergrenze zurückgeliefert und umgekehrt.
NOCYCLE	NOCYCLE gibt an, dass die Sequenz nicht bei Erreichen einer Grenze mit der anderen Intervallgrenze wieder von vorn beginnt (Voreinstellung).
CACHE	Definiert die Anzahl der Sequenznummern, die im Speicher gehalten werden sollen. Dieser darf nicht größer sein als das Ergebnis der folgenden Berechnung: (CEIL (MAXVALUE-MINVALUE)) / ABS(INCREMENT)
NOCACHE	Bewirkt, dass keine Sequenznummern im Speicher gehalten werden. Jede neue Anforderung einer Sequenz erzeugt somit einen Plattenzugriff, da der aktuelle Wert aus einer Tabelle gelesen werden muss. Wenn weder die CACHE- noch die NOCACHE-Anweisung angegeben wird, hält Oracle per Voreinstellung 20 Nummern im Speicher.
ORDER	Sorgt bei einem parallelen Zugriff von mehreren Benutzern auf die Sequenz dafür, dass die Nummern in der Reihenfolge der Anforderungen vergeben werden. Dadurch lässt die Vergabe der Nummern einen Rückschluss auf den Zeitpunkt des Schreibens des Datensatzes zu.
NOORDER	Schaltet die Option ORDER aus.

Tabelle 5-1: *Sequenz-Schlüsselwörter*

Im Prinzip steckt hinter dieser Funktionsweise nicht viel Programmieraufwand. Jeder Anwendungsprogrammierer sollte in der Lage sein, einen ähnlichen Algorithmus zu implementieren. Große Bedeutung kommt diesen Sequenzen erst in einer Mehrbenutzerumgebung zu. Stellen Sie sich ein Programm vor, das auf mehreren Clients läuft. Jeder Anwender legt jetzt quasi gleichzeitig einen neuen Satz in der Kundentabelle an. Für einen solchen Fall müssen Sie als Anwendungsprogrammierer dafür sorgen, dass auch bei gleichzeitigem Schreibzugriff auf die Tabelle immer eine eindeutige Kundennummer ver-

geben wird. Solche Sequenzen nehmen Ihnen dabei ein großes Stück Arbeit ab. Ein weiterer Vorteil ist die erhebliche Performance solcher Sequenzen. Da ein Pool von 20 Sequenznummern im Speicher gehalten wird, ist kein weiterer Plattenzugriff notwendig, um eine neue Nummer zu bestimmen.

Hinweis:
Obwohl Sie eine Sequenz über die CREATE-Anweisung erzeugen, handelt es sich im eigentlichen Sinn nicht um ein Objekt in einer Oracle-Datenbank. Dies erkennen Sie daran, dass Sie eine solche Sequenz nicht über den Personal Navigator finden, wie es beispielsweise bei einer View der Fall ist. Über CREATE SEQUENCE erzeugen Sie einen Datensatz in einer der System-Tabellen von Oracle. In dieser Tabelle sind all die Informationen abgelegt, die Sie mit den Parametern der obigen Tabelle einstellen können.

5.3 Synonyme

5.3.1 Allgemeines

Ein Synonym ist ein Alias-Name für eine Tabelle oder View. Hierüber wird also ein zweiter Name für eine Tabelle (oder View) vergeben. Zu beachten ist, dass ein Synonym lediglich eine Referenz auf eine Tabelle darstellt. Die vorhandene Tabelle bleibt unter ihrem ursprünglichen Namen bestehen, und es wird keine Kopie der Tabelle angelegt. Die Arbeit mit Synonymen bietet gegenüber der Verwendung der ursprünglichen Tabellennamen folgende Vorteile:

- der wirkliche Tabellenname und der Besitzer der Tabelle bleiben verborgen;
- der Ort, an dem die physikalische Tabelle gespeichert ist, bleibt verborgen;
- die Vergabe von Synonymen stellt ausserdem eine Vereinfachung für den Anwender dar, der mittels interaktiver SQL-Anwendungen auf die Datenbank zugreift.

Abhängig von der Definitionsart des Synonyms unterscheidet man zwischen *privaten* und *öffentlichen* Synonymen. Die privaten besitzen nur für den Benutzer ihre Gültigkeit, der sie auch angelegt hat. Öffentliche Synonyme haben dagegen Gültigkeit für die gesamte Datenbank, alle Benutzer haben also Zugang zu diesen Synonymen.

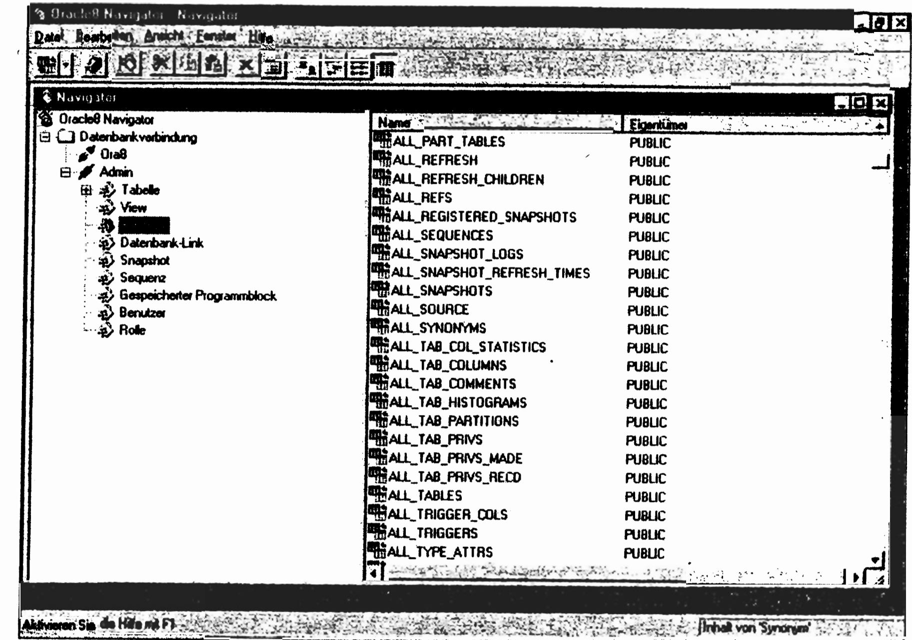

Abbildung 5-3: *Synonyme von Oracle*

Der Oracle Navigator zeigt hier zwei verschiedene Arten von Synonymen an. Synonyme, die mit V$ beginnen, basieren auf *virtuellen* Tabellen. Diese Tabellen existieren lediglich zur Laufzeit der Datenbank. Sie enthalten Informationen über den aktuellen Zustand der Datenbank und deren Tabellen. Die Tabellen stellen sich für den Anwender bzw. Datenbankadministrator als Tabellen dar; in der Realität handelt es sich aber lediglich um Zeiger auf einen Adressraum (SGA) im Hauptspeicher des Servers.

Bei den Synonymen, denen kein V$ vorangestellt ist, handelt es sich um „echte" Tabellen. Führen Sie auf einem solchen Objekt einen Doppelklick im Navigator aus. Es erscheint ein Dialog, wie er Abbildung 5-4 zu sehen ist.

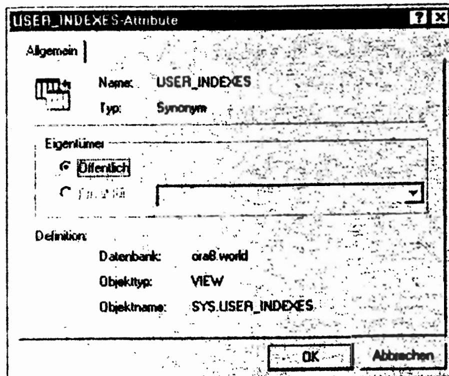

Abbildung 5-4: Synonym für Benutzer-Indizes

Ein Doppelklick auf ein Synonym öffnet also nicht die Tabelle, sondern zeigt lediglich einen Dialog mit den Eigenschaften des Synonyms an. Aus der Abbildung 5-4 erfahren Sie so, dass die Tabelle ebenfalls USER_INDEXES heisst und der Besitzer der Tabelle SYS ist. Den Inhalt der Tabelle bzw. die Tabellenstruktur können Sie sich über SQL*Plus ansehen:

```
SQL> desc user_indexes;
 Name                             Null?    Type
 -------------------------------- -------- ----
 INDEX_NAME                       NOT NULL VARCHAR2(30)
 TABLE_OWNER                      NOT NULL VARCHAR2(30)
 TABLE_NAME                       NOT NULL VARCHAR2(30)
 TABLE_TYPE                                VARCHAR2(11)
 UNIQUENESS                                VARCHAR2(9)
 TABLESPACE_NAME                  NOT NULL VARCHAR2(30)
 INI_TRANS                        NOT NULL NUMBER
 MAX_TRANS                        NOT NULL NUMBER
 INITIAL_EXTENT                            NUMBER
 NEXT_EXTENT                               NUMBER
 MIN_EXTENTS                      NOT NULL NUMBER
 MAX_EXTENTS                      NOT NULL NUMBER
 PCT_INCREASE                     NOT NULL NUMBER
 FREELISTS                                 NUMBER
 FREELIST_GROUPS                           NUMBER
 PCT_FREE                         NOT NULL NUMBER
 BLEVEL                                    NUMBER
 LEAF_BLOCKS                               NUMBER
 DISTINCT_KEYS                             NUMBER
 AVG_LEAF_BLOCKS_PER_KEY                   NUMBER
 AVG_DATA_BLOCKS_PER_KEY                   NUMBER
 CLUSTERING_FACTOR                         NUMBER
 STATUS                                    VARCHAR2(11)

SQL>
```


5.3.2 Synonyme erzeugen

Zum Erzeugen eines Synonyms verwenden Sie folgende Syntax:

```
CREATE SYNONYM <Synonym-Name> FOR <Besitzer.Objektname>
```

5.3.3 Synonyme löschen

Gelöscht wird ein Synonym entsprechend über eine DROP-Anweisung:

```
DROP SYNONYM <Synonym-Name>
```

5.3.4 Synonyme am Beispiel

Zur Verdeutlichung der Funktionalität von Synonymen lassen Sie mich an dieser Stelle ein kurzes Beispiel einfügen. Starten Sie dazu SQL*Plus und melden Sie sich mit Ihrem Namen bzw. mit Ihrer Benutzerkennung an (also so, wie Sie sich in Abschnitt 3.4 für die Datenbank eingerichtet haben). Versuchen Sie jetzt einen Zugriff auf die Tabelle *dept* zu realisieren, dessen Besitzer ja bekanntlich der Benutzer Scott ist. Oracle wird dieses mit folgender Fehlermeldung quittieren:

```
SQL> SELECT * from dept;
SELECT * FROM dept
              *
FEHLER in Zeile 1:
ORA-00942: Tabelle oder View nicht vorhanden

SQL>
```

Selbst das Voranstellen des Besitzernamens vor den Tabellennamen bringt keinen Erfolg:

```
SQL> SELECT * FROM scott.dept;
SELECT * FROM scott.dept
                    *
FEHLER in Zeile 1:
ORA-00942: Tabelle oder View nicht vorhanden

SQL>
```

Die Ursache liegt darin, dass Sie als Benutzer (noch) keine Berechtigung haben, eine beliebige Operation auf der Tabelle *dept* durchzuführen. Starten Sie jetzt eine zweite Instanz von SQL*Plus und melden Sie sich als Scott an. In dieser Sitzung vergeben Sie zunächst

die Leserechte auf der Tabelle *dept* für Ihren eigenen Benutzer. Der folgende Befehl zeigt diese Rechtevergabe noch einmal:

```
SQL> GRANT SELECT ON dept TO heitsiek;

Grant succeeded.

SQL>
```

Wechseln Sie wieder zur ersten SQL*Plus-Sitzung, in der Sie mit Ihrem eigenen Namen angemeldet sind. Jetzt haben Sie die Möglichkeit, SELECT-Anweisungen auf der Tabelle *dept* durchzuführen:

```
SQL> SELECT * FROM scott.dept;

   DEPTNO DNAME          LOC
--------- -------------- ------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL>
```

Wie Sie sehen, funktioniert die Selektion allerdings nur, wenn dem Tabellennamen der originäre Besitzer der Tabelle vorangestellt wird. Genau wegen dieser Umständlichkeit werden Synonyme so oft verwendet. Erstellen Sie jetzt über die folgende SQL-Anweisung ein Synonym für die Tabelle *dept* des Besitzers *Scott*:

```
CREATE SYNONYM dept FOR scott.dept;
```

Dadurch haben Sie die Möglichkeit, die Tabelle anzusprechen, ohne den Besitzer *Scott* immer mit angeben zu müssen. Der Zugriff erfolgt jetzt immer über die Referenz des Synonyms:

```
SQL> SELECT * FROM dept;

   DEPTNO DNAME          LOC
--------- -------------- ------------
       10 ACCOUNTING     NEW YORK
       20 RESEARCH       DALLAS
       30 SALES          CHICAGO
       40 OPERATIONS     BOSTON

SQL>
```


5.4 Trigger

5.4.1 Allgemeines

Zu Beginn dieses Kapitels haben wir uns mit Sequenzen beschäftigt und dabei gelernt, dass es damit möglich ist, ein Stück Intelligenz von der Anwendung in das DBMS auszulagern. Mit Triggern verhält es sich ähnlich. Mit ihnen wird ebenfalls eine bestimmte Funktionalität in die Datenbank ausgelagert; das Anwendungsprogramm und damit der Client wird also entlastet. Zur Laufzeit der Anwendung wird außerdem das Netzwerk wesentlich weniger belastet, je mehr Routinen direkt auf dem Server ablaufen. Dies ist gerade bei einer großen Anzahl von Benutzern ein nicht zu unterschätzender Faktor; und schließlich ist die Performance einer Anwendung ein wesentliches Kriterium für die Akzeptanz durch die Anwender! Stellen Sie sich folgendes Beispiel vor:

In einem Unternehmen ist ein Verwaltungsprogramm für das Materiallager installiert. Von diesem Anwender wird gewünscht, dass das Verwaltungsprogramm für bestimmte Artikel automatisch Bestellungen auslöst, sobald eine bestimmte Menge des am Lager befindlichen Produkts unterschritten wird. Die Anwendung muss also bei jeder Entnahme überprüfen, ob eine zuvor definierte Menge unterschritten wird. Eine solche Materialentnahme kann aber (so sollte es zumindest sein) unterschiedlich realisiert werden. Ein Anwender kann eine Reservierung aussprechen; ein anderer entnimmt wirklich eine bestimmte Anzahl und bucht diese ab; ein Dritter führt eine Inventurkorrektur durch usw. All diese Vorgänge reduzieren im Prinzip die Menge des Artikels am Lager. Die Routine, die die Unterschreitung der Menge prüft, müsste also in diesem Beispiel an drei verschiedenen Stellen im Quelltext eingebaut werden. Eine solche Vorgehensweise erhöht nicht nur den Wartungsaufwand für die Software, das Programm wird auch wesentlich fehleranfälliger.

Hier nimmt Oracle dem Anwendungsprogrammierer ein ganzes Stück lästiger Arbeit ab. Mit Hilfe von Triggern können Aktionen ausgelöst werden, wenn bestimmte, zuvor definierte Ereignisse eintreten. Man kann für das eben erwähnte Beispiel einen Trigger definieren, der eine Prozedur aufruft, sobald sich die Menge eines Artikels (in der Datenbanksprache würde man jetzt von dem Inhalt eines Feldes sprechen) in der Lager-Datei ändert. Durch diese Vorgehensweise ergeben sich folgende Vorteile:

- die Prozedur, die die Bestellung auslöst, muss lediglich an einer Stelle definiert werden
- ein Stück Intelligenz wird auf die Datenbank ausgelagert
- durch diese Art des Client-Server-Computings sinkt die Netzlast, da die Operationen direkt auf der Datenbank ablaufen
- die Software wird insgesamt wartungsfreundlicher

Man kann einen Trigger für INSERT-, UPDATE- oder DELETE-Anweisungen definieren, die auf eine Tabelle abgesetzt werden. Unterschieden werden sie außerdem noch in *BEFORE*- und *AFTER-Trigger*. BEFORE-Trigger werden **vor** der eigentlichen Abarbeitung des Trigger-auslösenden Ereignisses ausgeführt, AFTER-Trigger **nach** Ausführung der entsprechenden INSERT-, UPDATE- oder DELETE-Anweisung. Um die Verwirrung zu kom-

plettieren, werden solche Trigger weiterhin noch in *Zeilentrigger* und *Anweisungstrigger* unterschieden. Zeilentrigger werden jedesmal ausgeführt, wenn die auslösende Anweisung eine Zeile beschreibt. Werden durch eine Anweisung beispielsweise zehn Zeilen einer Tabelle upgedatet, so würde ein entsprechend definierter Zeilentrigger zehn Mal ausgeführt. Dem steht der Anweisungstrigger gegenüber, der nur einmal pro auslösender Anweisung ausgeführt wird. Alle drei Eigenschaften des Triggers sind natürlich kombinierbar. Der mathematisch begabte Leser möge sich an dieser Stelle die Anzahl der möglichen Triggerarten selbst ausrechnen. Die folgende Liste gibt Aufschluss über die Reihenfolge, in der die verschiedenen Trigger ausgeführt werden:

1. Vor der eigentlichen Ausführung der auslösenden SQL-Anweisung werden BEFORE-Anweisungstrigger gestartet.
2. Dann werden die BEFORE-Zeilentrigger ausgeführt.
3. Danach erfolgt die Ausführung des eigentlichen SQL-Statements.
4. Nach der Ausführung der SQL-Anweisung erfolgt die Abarbeitung der AFTER-Zeilentrigger (Die Punkte 2. bis 4. wiederholen sich jetzt so oft, wie Zeilen betroffen sind).
5. Im letzten Schritt werden die AFTER-Anweisungstrigger ausgeführt.

5.4.2 Zur Syntax

Trigger werden wie alle anderen Objekte auch durch SQL-Anweisungen erzeugt. Aufgrund ihrer Flexibilität und deren zahlreicher Optionen ist die Syntax zum Erzeugen eines Triggers entsprechend kompliziert:

```
CREATE [OR REPLACE] TRIGGER Triggername (BEFORE | AFTER)
Trigger-Ereignis ON Tabellenname
[FOR EACH ROW]
[WHEN Bedingung]
Anweisungsblock
```

Die Anweisungen in den eckigen([]) Klammern sind optional. Der Zusatz OR REPLACE ist dann sinnvoll, wenn schon ein Trigger mit gleichem Namen existiert. Dieser Zusatz sorgt dafür, dass das System die alte Definition überschreibt. Wird er nicht mit angegeben und die Erzeugung eines Triggers versucht, der dem Namen nach schon existiert, kommt es zu einer Fehlermeldung. *Triggername* gibt dabei den Namen des Triggers an.

Nach Angabe des Namens erfolgt die Definition, ob es sich bei dem zu erzeugenden Trigger um einen BEFORE- oder AFTER-Trigger handeln soll. Diese Angabe ist nicht optional; einer der beiden Werte muss angegeben werden (Ich muss nicht extra erwähnen, dass sich diese Parameter gegenseitig ausschließen).

Das *Trigger-Ereignis* definiert, wann der Trigger überhaupt ausgelöst werden soll. Mögliche Werte hierfür sind INSERT, UPDATE, DELETE. Diese Parameter schließen sich gegenseitig nicht aus. Eine Kombination aus zwei oder drei Werten ist möglich. In diesem Fall sind die einzelnen Eigenschaften über OR zu verknüpfen.

Die optionale Angabe FOR EACH ROW definiert den Trigger als *Zeilentrigger*. Erfolgt sie nicht, so handelt es sich bei dem zu erzeugenden Trigger per Voreinstellung um einen

Anweisungstrigger.

Der WHEN-Zusatz kann noch eine zusätzliche Bedingung abfragen. Die Abarbeitung des Triggers kann man hierüber von dem Booleschen Wert in *Bedingung* abhängig machen.

Der Anweisungsblock enthält die SQL-Anweisung, die bei Auslösung des Triggers ausgeführt werden sollen.

Bezüglich der Zeilentrigger möchte ich noch einige zusätzliche Informationen geben. Zeilentrigger werden dann eingesetzt, wenn mit den Feldinhalten von bestimmten Zeilen Operationen durchgeführt werden sollen. In diesen Fällen werden oftmals die Feldinhalte benötigt, weil sie im Anweisungsblock des Triggers verarbeitet werden. Denkbar wäre beispielsweise ein Zeilentrigger, der den alten Wert eines Feldes (oder mehrerer Felder) in einer Historiendatei sichert, bevor dieser durch eine UPDATE-Anweisung überschrieben wird. Abhängig von der Art des Triggers (INSERT, UPDATE und DELETE) stehen im Anweisungsblock bestimmte globale Variablen zur Verfügung. *Spaltenname* definiert dabei den Namen der Tabellenspalte, die modifiziert werden soll. Tabelle 5-2 fasst die Definitionsmöglichkeiten noch einmal zusammen.

Art des Triggers	Variable	Beschreibung
INSERT	:new.Spaltenname	Nur gültig bei einem BEFORE-Trigger. In dieser Variablen befindet sich dann der einzufügende Wert.
UPDATE	:old.Spaltenname	Der alte Wert des Feldes befindet sich vor Ausführung des Trigger in dieser Variablen.
	:new.Spaltenname	Enthält den Wert der UPDATE-Anweisung, d.h. der neue Feldinhalt.
DELETE	:old.Spaltenname	Enthält den Wert des zu löschendes Feldes.

Tabelle 5-2: *Trigger-Definitionen*

Achten Sie bei der Verwendung dieser Variablen darauf, dass den Schlüsselwörtern immer ein „:" vorangestellt ist. Ansonsten wird Oracle die Arbeit verweigern. Es handelt sich hierbei um einen beliebten Fehler. Die Variable *:old* ist für die INSERT-Anweisung nicht definiert, bzw. sie besitzt den Wert NULL. Das gleiche gilt für die Variable *:new* im Zusammenhang mit der DELETE-Anweisung.

Hinweis:
Bei *:new* und *:old* handelt es sich um Schlüsselwörter des DBMS. Unter Umständen kann es jetzt passieren, dass innerhalb einer Tabelle genau diese Bezeichnung als Tabellenspaltennamen verwendet wurden. Zu diesem Zweck kennt Oracle die Anweisung REFERENCING. Hierüber ist es möglich, innerhalb eines Triggers diesen beiden Schlüsselwörtern neue Bezeichnungen zuzuweisen:

```
REFERENCING new AS newest
REFERENCING old AS oldest.
```

Ab Oracle 7.3 besteht die Möglichkeit, Trigger auf eine View zu definieren, denn Views können ebenso wie Tabellen beschrieben und ausgelesen werden. Allerdings gibt es eine Ausnahme. Sollte eine View aus einem Join mehrerer Tabellen aufgebaut sein, funktioniert ein INSERT oder UPDATE nicht in jedem Fall, sondern nur dann, wenn lediglich eine Tabelle upgedatet oder beschrieben werden soll. Vor der Ausführung einer INSERT-Anweisung wird natürlich überprüft, ob diese überhaupt zulässig ist. Wenn jetzt für eine solche INSERT-Anweisung zusätzlich ein Trigger definiert ist, wird natürlich auch dessen Ausführung verworfen, sofern die INSERT-Anweisung nicht ausgeführt wird. Genau für diesen Fall gibt es in Oracle 8 einen neuen Typ des Triggers, den *INSTEAD OF*-TRIGGER.

Mit diesem Typ kann ein Trigger definiert werden, der anstelle eines anderen Triggers (im Beispiel also der Trigger auf die View) ausgeführt wird, falls die Ausführung des ursprünglichen Triggers scheitert. Er kann nur als Zeilentrigger definiert werden.

Ich kann aber vor der Verwendung dieser Art der Programmierung nur warnen. Views sollten für den Zweck verwendet werden, wofür sie ursprünglich dienen sollten, und was die deutsche Übersetzung – Sicht – bestens beschreibt. Views sollen das, manchmal recht komplizierte, Datenbankmodell einer Anwendung vereinfachen, so dass auch einfache Anwender beispielsweise intuitive Reports anfertigen können. Sie sollen die Daten eben aus einer anderen Sicht darstellen. Davon abgesehen zeigt sich, dass Treiber wie die SQL Links von Borland oder diverse ODBC-Treiber so ihre Schwierigkeiten mit *Updatable Views* haben, so dass dadurch schon die Verwendung erheblich eingeschränkt sein dürfte.

5.4.3 Trigger erzeugen

Für das Erzeugen eines Triggers muss der angemeldete Benutzer gewisse Rechte besitzen. Wenn Sie sich also in SQL*Plus mit Ihrem in Kapitel 3 eingerichteten Benutzernamen anmelden, besitzen Sie diese Rechte noch nicht. Am schnellsten erhalten Sie das Recht, indem sie die entsprechende Anweisung in SQL*Plus absetzen. Abbildung 5-5 zeigt diesen Dialog.

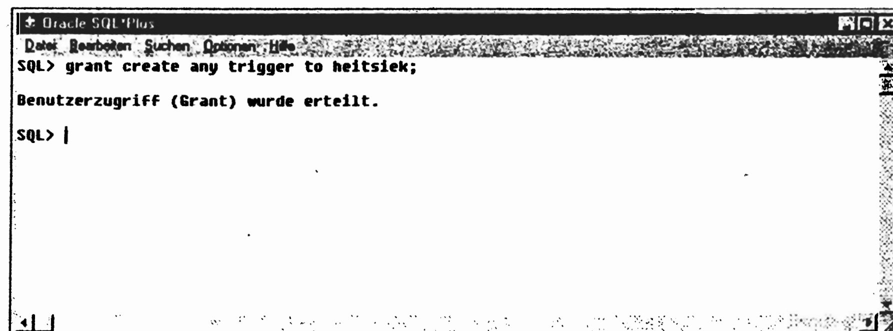

Abbildung 5-5: *Rechte vergeben*

Damit haben Sie die „Lizenz zum Triggern". Das folgende Listing zeigt ein Script, welches eine Historien-Tabelle und einen entsprechenden Trigger erzeugt:

```
DROP TABLE HISTORIE;

CREATE TABLE historie (
   DATUM date,
   KUNDENNR NUMBER(3),
   KUNDENNEU NUMBER(3));

CREATE OR REPLACE TRIGGER hist BEFORE
UPDATE ON kunden
FOR EACH ROW

BEGIN
   INSERT INTO historie
   (datum, kundennr, kundenneu)
   VALUES
   (sysdate, :old.kundennr, :new.kundennr);
END;
/
```

Hinweis:
Dieses Script finden Sie im Netz in dem Verzeichnis *Scripts* unter dem Dateinamen TRIGGER.SQL. Enorm wichtig bei einem PL/SQL-Script ist die letzte Zeile. Befindet sich hier kein Slash (/), werden Sie das Script nicht zum Laufen bringen.

Die erste Zeile dieser Datei veranlasst das DBMS, die Tabelle *Historie* zu löschen. Im Prinzip benötigt man diese Zeile nicht; zur Entwicklungszeit einer Anwendung kommt es aber häufiger vor, dass man an Tabellenstrukturen Veränderungen vornehmen muss. Dabei sind diese Zeilen dann schon sinnvoll. Danach wird die Historien-Tabelle erzeugt. In dieser Tabelle sollen Änderungen protokolliert werden, die an dem Feld *Kundennr* der Kunden-Tabelle vorgenommen werden. Da das Feld *Kundennr* der Tabelle *Kunden* das eindeutige Schlüsselfeld ist, sollte man damit äußerst vorsichtig umgehen.

In der Historien-Tabelle wird die Veränderung (UPDATE-Anweisung) der Kundennummer protokolliert. Dabei sollen sowohl die alte als auch die neue Kundennummer in die Historien-Tabelle eingetragen werden. Gleichzeitig soll noch das aktuelle Tagesdatum gespeichert werden. Hierüber ist dann eine Rückverfolgbarkeit gewährleistet, ab welchem Datum ein Kunde eine neue Kundennummer erhalten hat.

Die folgenden Zeilen definieren dabei den Trigger-Kopf:

```
CREATE OR REPLACE TRIGGER hist BEFORE
UPDATE ON kunden
FOR EACH ROW
```

Name des Triggers ist *hist*. Es handelt sich um einen BEFORE-Trigger, der bei jedem UPDATE-Zugriff pro Tabellenzeile aktiviert wird. Die nächsten Zeilen definieren den Trigger-Rumpf, die eigentliche Funktionalität des Triggers. Dieser Rumpf ist syntaktisch ähnlich

wie eine Prozedur in Pascal aufgebaut. Der eigentliche Funktionsblock wird in die Anweisungen BEGIN und END gekapselt. Vor dem Block erfolgt die (mögliche) Deklaration von Variablen. In Pascal wird dies durch das Schlüsselwort VAR eingeleitet, in dem Oracle-Dialekt durch das Schlüsselwort DECLARE. Innerhalb des Funktionsblockes ist eine einfache INSERT-Anweisung definiert.

```
DECLARE

BEGIN
   INSERT INTO historie
   (datum, kundennr, kundenneu)
   VALUES
   (sysdate, :old.kundennr, :new.kundennr);
END;
/
```

Führen Sie dieses Script jetzt in SQL*Plus aus. Danach steht Ihnen die neue Tabelle *historie* sowie der Trigger *hist* zur Verfügung. Die Historien-Tabelle ist nach Ausführung der Anweisung noch leer, wie die folgende SQL-Anweisung belegt:

```
SQL> SELECT COUNT(*) FROM historie;

 COUNT(*)
---------
        0

SQL>
```

In der Kundentabelle befinden sich z.Zt. sechs Datensätze:

```
SQL> SELECT kundennr FROM kunden;

 KUNDENNR
---------
      100
      101
      102
      103
      104
      105

6 rows selected.

SQL>
```


Führen Sie die folgende Anweisung in SQL*Plus aus. Diese modifiziert Kundennummer 100:

```
UPDATE  kunden
SET     kundennr=500
WHERE   kundennr=100;
```

Wenn unser Trigger korrekt gearbeitet hat, müsste sich jetzt ein Datensatz in der Historie-Tabelle befinden. Überprüfen Sie dies mit der folgenden Anweisung:

```
SELECT * FROM historie;
```

Abbildung 5-6 zeigt die Ergebnisse in den Tabellen *Kunden* und *Historie*.

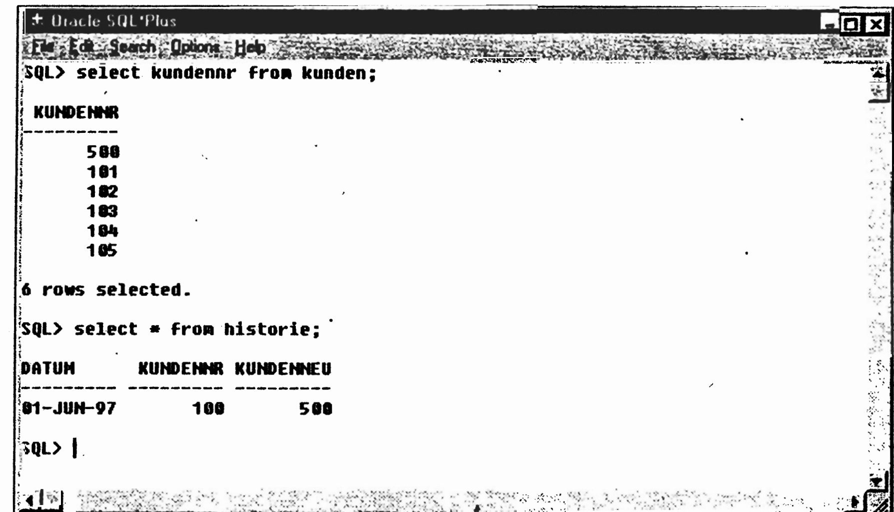

Abbildung 5-6: *Ergebnis des Triggers*

Das folgende Script zeigt weitere Einsatzmöglichkeiten von Triggern. Aufgabe dieses Triggers ist es, nach jedem Löschen oder Hinzufügen von Datensätzen in der Kundentabelle die aktuelle Anzahl an Datensätzen mit dem aktuellen Tagesdatum in eine weitere Tabelle, der Tabelle *Statist*, abzulegen. Hierüber lässt sich eine Statistik führen, wie und in

welchem Zeitraum der Kundenstamm wächst. Das entsprechende Script finden Sie im Internet unter dem Dateinamen TRIGGER2.SQL:

```
DROP TABLE statist;

CREATE TABLE statist(
   datum       DATE,
   anz_kund    NUMBER);

CREATE OR REPLACE TRIGGER test AFTER
INSERT OR DELETE ON kunden

DECLARE
   Anzahl_Kunden NUMBER;

BEGIN
   SELECT COUNT(Kundennr)
   INTO   Anzahl_Kunden
   FROM   KUNDEN;

   INSERT INTO statist  (datum, anz_kund)
   VALUES               (sysdate, Anzahl_Kunden);
END;
/
```

Hinweis:
Mit der SELECT INTO-Anweisung weisen Sie die Ergebnismenge der Abfrage einer Variablen zu; im obigen Fall wird die Anzahl der Kundensätze in der Tabelle *Kunden* der Variablen *Anzahl_Kunden* zugewiesen. Ein solche Zuweisung erfolgt allerdings immer nur dann fehlerfrei, wenn sichergestellt ist, dass genau ein Ergebnissatz und nicht mehr zurückgeliefert werden. Dies erreicht man am einfachsten dadurch, dass man in einer WHERE-Bedingung auf den eindeutigen Schlüssel der Tabelle abfragt.

Vor Beginn der eigentlichen Routine wird die Variable *Anzahl_Kunden* deklariert, in der später die Anzahl Sätze der Kundentabelle geschrieben wird. Anhand dieses Scripts lernen Sie auch die Verwendung von Variablen innerhalb eines Triggers. Eine solche Variable wird im Prinzip deklariert wie in jeder anderen Programmiersprache auch. Bei den zur Verfügung stehenden Datentypen handelt es sich um die Typen, die auch bei Anlegen von Tabellen Verwendung finden. Die Aktualisierung der Anzahl Kunden muss sowohl beim Neuanlegen wie auch beim Löschen von Kunden erfolgen. Aus diesem Grund sind die Trigger-Ereignisse INSERT und DELETE über OR miteinander verknüpft.

Führen Sie dieses Script jetzt aus. Wie nicht anders zu erwarten, ist die Tabelle *Statist* noch leer:

```
SQL> SELECT * FROM statist;

no rows selected

SQL>
```

Für einen einfachen Test löschen wir einfach alle Sätze aus der Kundentabelle. Danach fragen wir die Tabelle *Statist* ab, um am Ende wieder ein Rollback zu fahren und die Datensätze wiederherzustellen. Abbildung 5-7 zeigt diesen Vorgang.

```
Oracle SQL*Plus
File Edit Search Options Help
SQL> delete from kunden;

6 rows deleted.

SQL> select * from statist;

DATUM      ANZ_KUND
---------  ---------
01-JUN-97          0

SQL> rollback;

Rollback complete.

SQL>
```

Abbildung 5-7: *Kundenstatistik führen*

Ein anderer interessanter Anwendungsfall für den Einsatz von Triggern sind automatische Berechnungen von Feldwerten innerhalb einer Tabelle. Denkbar wäre solch ein Einsatz beispielsweise für eine Datenbankanwendung, die neben ihrer eigentlichen Funktionalität noch verschiedene, kundenspezifische Kalkulationsschemata abbilden soll. Solch eine Aufgabenstellung lässt sich hervorragend über Trigger und Stored Procedures abbilden. Sie haben damit die Möglichkeit, zusätzliche Funktionen zu implementieren, ohne die ausführbaren Dateien anfassen zu müssen. Lassen Sie mich dazu ein kurzes Beispiel geben.

Zunächst erzeugen wir auf der Datenbank eine einfache Tabelle mit drei Spalten, alle vom Typ NUMBER:

```
SQL> CREATE TABLE kalk
  2      ( zeile NUMBER(3),
  3        a NUMBER(13,5),
  4        b NUMBER(13,5),
  5        c NUMBER(13,5));
```

Aufgabe dieser Tabelle ist es, in den Spalten a und b verschiedene Arten von Kosten abzulegen. In der Spalte c soll außerdem die Summe aus den beiden ersten Werten erscheinen. Dazu definieren wir den folgenden Trigger:

```
CREATE TRIGGER kalk_1
BEFORE INSERT ON kalk
FOR EACH ROW

BEGIN

:new.c := :new.a + :new.b;

END;
/
```

Sie sehen hieran, dass es durchaus möglich ist, vor dem eigentlichen Schreiben in eine Tabelle Berechnungen durchzuführen. Es stellt sich nun die Frage, was passiert, wenn ein INSERT auf diese Tabelle nicht alle Spalten, genauer die Spalte „c" nicht umfasst. Lassen Sie uns dies durch folgende Anweisung testen:

```
SQL> INSERT INTO kalk (zeile,a,b)
  1  VALUES (1, 10.0, 15.3);
```

Eine Selektion auf die Tabelle liefert folgendes Ergebnis:

```
SQL> SELECT * FROM kalk;

    ZEILE          A          B          C
--------- ---------- ---------- ----------
        1         10         15         25

SQL>
```

Interessant hierbei ist, dass die Spalte c beschrieben wird, obwohl sie nicht in der INSERT-Anweisung erscheint. Dies ist nur ein einfaches Beispiel für die mannigfaltigen Einsatzmöglichkeiten von Triggern. Denkbar wäre in diesem Zusammenhang die automatische Berechnung von gleitenden Durchschnittspreisen einer Lager-Verwaltung oder eine automatische Warnmeldung bei Unterschreitung von Mindestbeständen.

Lassen Sie mich noch einen wichtigen Hinweis in Bezug auf Zeilentrigger geben. Wenn Sie mit solchen Triggern arbeiten, werden Sie früher oder später auf das folgende Problem stoßen: Eine INSERT-Anweisung löst einen bestimmten Zeilentrigger aus. Innerhalb dieses Triggers erfolgt ein weiterer Schreibzugriff auf die den Trigger auslösende Tabelle, beispielsweise in Form einer UPDATE-Anweisung. Dazu die Definition des folgenden Triggers. Das entsprechende Script hierzu finden Sie unter dem Dateinamen KALK2.SQL im Netz:

```
CREATE TRIGGER kalk_2
AFTER INSERT ON kalk
FOR EACH ROW

BEGIN
   UPDATE kalk
   SET    c = :new.a + :new.b
   WHERE  zeile = :new.zeile;
END;
/
```

Im Prinzip soll dieser Trigger genau die gleiche Funktionalität erfüllen wie die oben beschriebene Lösung. Ein INSERT auf die entsprechende Tabelle führt jetzt allerdings zu einer Fehlermeldung:

```
SQL> INSERT INTO kalk
  2  (zeile, a, b)
  3  VALUES
  4  (2, 5.0, 5.0);
insert into kalk
            *
ERROR at line 1:
ORA-04091: table HEITSIEK.KALK is mutating, trigger/function may not see it
ORA-06512: at line 2
ORA-04088: error during execution of trigger 'HEITSIEK.KALK_2'
```

Dieser Fehler (ORA-04091) tritt immer dann auf, wenn innerhalb eines Triggers ein Schreibzugriff auf die gleiche Tabelle ausgeführt werden soll, die auch den Trigger ausgelöst hat. Das Problem ist, dass die Trigger-auslösende INSERT-Anweisung im Prinzip noch nicht abgeschlossen ist. Es ist also noch kein COMMIT abgesetzt worden. Dieses COMMIT ist aber zwingend notwendig, denn innerhalb des Triggers wird (soll) auf Tabellendaten zurückgegriffen werden, die erst durch die auslösende Anweisung geschrieben werden.

Hinweis:
Ab der Version 7.1 von Oracle können mehr als ein Trigger je Typ auf einer Tabelle definiert werden. Denkbar sind somit beispielsweise drei Zeilentrigger, die vor einer INSERT-Anweisung ausgeführt werden. Problematisch ist hierbei nur, dass die Reihenfolge, in der die drei Trigger ausgeführt werden, nicht vom Anwender beeinflusst werden kann. Eine unterschiedliche Reihenfolge der Abarbeitung kann dadurch nicht ausgeschlossen werden!

5.4.4 Trigger ausschalten und löschen

Nachdem wir im letzten Abschnitt Trigger erzeugt haben, ist es natürlich interessant zu erfahren, wie solche Trigger wieder gelöscht werden können. Es gibt zwei Möglichkeiten, sich eines Triggers zu entledigen:

- Löschen
- Inaktivierung

Das Löschen eines Triggers erfolgt analog dem Löschen einer Tabelle oder allgemein eines Objektes innerhalb der Datenbank:

```
DROP TRIGGER Triggername
```

Das Löschen ist aber oftmals nicht erwünscht, möchte man einen Trigger doch nur für einen bestimmten Zeitraum „Abschalten". Dafür kennt Oracle eine weitere Anweisung:

```
ALTER TRIGGER Triggername DISABLE
```

Über diese Anweisung wird ein Trigger abgeschaltet. Die folgende Anweisung schaltet ihn wieder ein:

```
ALTER TRIGGER Triggername ENABLE
```

Im Zusammenhang mit Triggern sind außerdem noch folgende Punkte zu beachten:

- Es ist durchaus möglich, dass eine SQL-Anweisung die Aktivierung mehrerer Trigger auslöst.
- Trigger können selbst Trigger auslösen. Findet ein Schreibzugriff durch einen Trigger auf eine Tabelle statt, auf der ebenfalls ein Trigger liegt, so spricht man von *kaskadierender Ausführung*.
- Innerhalb des Funktionsblockes können nicht die Anweisungen COMMIT und ROLLBACK verwendet werden.
- Innerhalb des Funktionsblockes eines Triggers muss nicht zwingend der Quelltext in Form von SQL-Anweisungen stehen. Innerhalb des Funktionsblockes können auch *Stored Procedures* aufgerufen werden (siehe Abschnitt 5.5.1).
- Trigger werden außerdem gelöscht, wenn die Tabelle, die die Trigger referenzieren, gelöscht wird.

5.4.5 Trigger im Überblick

Bei Triggern handelt es sich nicht um Objekte innerhalb der Datenbank, wie es beispielsweise Tabellen oder Indizes sind. Sie werden deshalb auch vergeblich nach Triggern im Personal Navigator suchen. Ähnlich den Sequenzen wird auch die Definition eines Triggers in den System-Tabellen von Oracle abgelegt. Über eine entsprechende Selektion aus diesen System-Tabellen kann man sich dann einen Überblick über die im System definier-

ten Trigger verschaffen. Abgelegt werden die Informationen in der Tabelle *user_triggers*. Im folgenden sehen Sie die Struktur dieser Tabelle:

```
SQL> desc dba_triggers
 Name                             Null?    Type
 -------------------------------- -------- ----
 OWNER                            NOT NULL VARCHAR2(30)
 TRIGGER_NAME                     NOT NULL VARCHAR2(30)
 TRIGGER_TYPE                              VARCHAR2(16)
 TRIGGERING_EVENT                          VARCHAR2(26)
 TABLE_OWNER                      NOT NULL VARCHAR2(30)
 TABLE_NAME                       NOT NULL VARCHAR2(30)
 REFERENCING_NAMES                         VARCHAR2(87)
 WHEN_CLAUSE                               VARCHAR2(2000)
 STATUS                                    VARCHAR2(8)
 DESCRIPTION                               VARCHAR2(2000)
 TRIGGER_BODY                              LONG

SQL>
```

Sie sehen, dass neben den „normalen" Datentypen auch der Typ *long* innerhalb dieser Tabelle verwendet wird. In diesem Feld wird sozusagen der Quelltext des jeweiligen Triggers abgelegt. Über entsprechende Abfragen dieser Tabelle erhalten Sie alle im Zusammenhang mit den auf dem System eingerichteten Triggern aufgelistet:

```
SQL> SELECT trigger_name, trigger_type, triggering_event,
  2  status FROM user_triggers;

TRIGGER_NAME TRIGGER_TYPE     TRIGGERING_EVENT           STATUS
------------ ------------     ------------------------   -------
HIST         FORE EACH ROW    UPDATE                     ENABLED
TEST         AFTER STATEMENT  INSERT OR DELETE           ENABLED

SQL>
```

Den Quelltext eines Triggers erhalten Sie folgendermaßen:

```
SQL> SELECT trigger_body FROM user_triggers
  2  WHERE  trigger_name='TEST';

TRIGGER_BODY
----------------------------------------------
DECLARE
  Anzahl_Kunden Number;
BEGIN
  SELECT COUNT(Kundennr)
  INTO   Anzahl_Kunden
  FROM   KUNDEN;
```

```
    INSERT INTO statist (DATUM, ANZ_KUND)
    VALUES              (sysdate, Anzahl_Kunden);
END;

SQL>
```

Hinweis:
Sollte SQL*Plus die Anzeige auf Ihrem Rechner mitten im Quelltext einfach abschneiden, überprüfen Sie innerhalb von SQL*Plus die Einstellung *long*. Standardmäßig steht diese Variable auf 80, d.h. aus long-Datenfeldern werden die ersten 80 Zeichen angezeigt. Modifikationen an diesem Parameter können Sie im Menü *Options / Set Options* setzen. Abbildung 5-8 zeigt den entsprechenden Dialog.

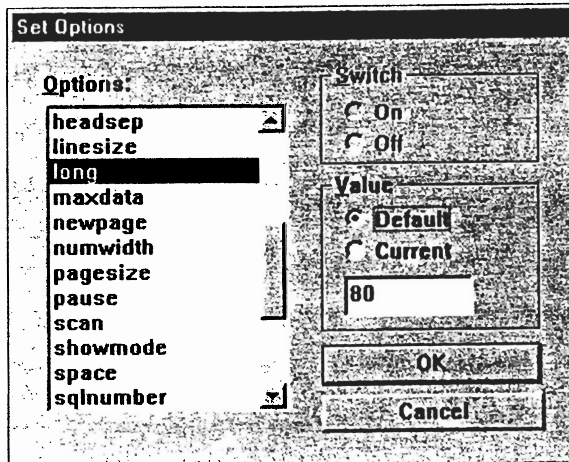

Abbildung 5-8:
*Einstellung von SQL*Plus*

5.5 Stored Procedures

5.5.1 Allgemeines

Stored Procedures sind nichts weiter als Unterroutinen, die in compilierter Form auf dem Datenbankserver liegen. Wie Oracle Stored Procedures zur Verfügung stellt, ähnelt sehr stark der Programmiersprache Turbo Pascal. Zunächst unterscheidet man zwischen Routinen, die einen Wert zurückliefern, und solchen, die keinen Wert zurückliefern. Erstere werden als Funktionen, zweitere als Prozeduren bezeichnet. Beide werden in der Fachterminologie unter dem Begriff *Stored Procedure* zusammengefasst.

Wie auch die Verwendung von Sequenzen und Triggern bietet die Verwendung von Stored Procedures wesentliche Vorteile gegenüber der herkömmlichen Anwendungsprogrammierung:

- Stored Procedures liegen direkt auf der Datenbank und können vom Anwendungsprogramm aufgerufen werden. Dadurch sinkt die Netzbelastung erheblich.
- Die mögliche Wiederverwendbarkeit ist sehr hoch. Neue Versionen der Anwendungssoftware können immer noch auf die gleichen Prozeduren zurückgreifen. Diese Stored Procedures können außerdem direkt aus SQL*Plus heraus aufgerufen werden, was ebenfalls zu einer schlankeren Softwareentwicklung beiträgt.
- Stored Procedures sind plattform-unabhängig, d.h. es müssen keine Anpassungen vorgenommen werden, möchte man die gleichen Routinen auf dem DBMS eines Unix- oder eines Windows NT-Servers nutzen.

Hinweis:
In den Entwicklungssystemen Delphi und C++ Builder der Fa. Imprise (ehemals Borland) ist der Aufruf einer Stored Procedure in einer eigenen Komponente gekapselt. Damit unterstützen diese Systeme die Client/Server-Programmierung optimal. Ab der Version 3 beider Systeme können innerhalb dieser Komponente sogar komplette Ergebnismengen (Cursor) als Rückgabewert weiterverarbeitet werden.

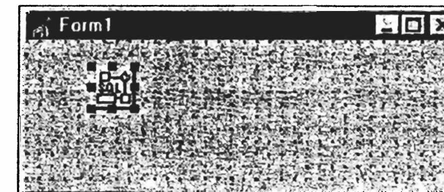

Abbildung 5-9:
TStoredProc von Borlands C++-Builder

5.5.2 Stored Procedures erzeugen

Bevor Sie versuchen, über SQL*Plus Stored Procedures zu erzeugen, müssen Sie die entsprechende Berechtigung besitzen. Der schnellste Weg führt wieder über den Personal Navigator. Abbildung 5-10 zeigt den entsprechenden Dialog zur Rechtevergabe.

Hinweis:
Das Recht `CREATE ANY PROCEDURE` beinhaltet das Recht sowohl zum Erzeugen von Prozeduren als auch zum Erzeugen von Funktionen.

Nachdem Sie sich die entsprechende Rechte zugesprochen haben, kann man mit der eigentlichen Arbeit beginnen. Die SQL-Anweisung zur Erzeugung einer Prozedur oder einer Funktion ist schon recht umfangreich (und glauben Sie mir bitte, es klappt nie auf Anhieb). Es empfiehlt sich deshalb, diese Anweisung als Scripts zu erzeugen und diese in SQL*Plus ablaufen zu lassen.

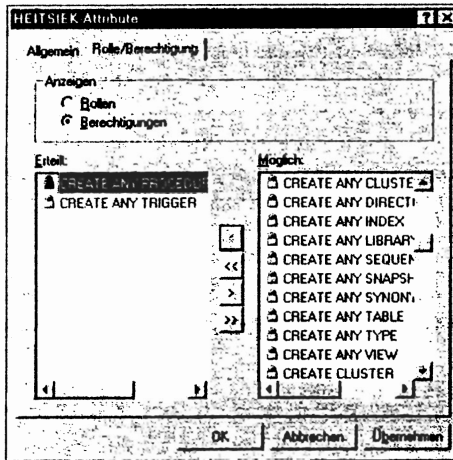

Abbildung 5-10:
Rechte vergeben

Hinweis:
Innerhalb einer Prozedur oder Funktion sind lediglich die SQL-Befehle zur Datenmanipulation zulässig. Befehle zur Definition und Kontrolle der Datenbank und deren Benutzer werden bei der Kompilierung einer solchen Routine mit Fehlermeldungen zurückgewiesen.

Die Syntax für das Erzeugen einer Prozedur lautet wie folgt:

```
CREATE [OR REPLACE] PROCEDURE Prozedurname
[Argumentenliste] IS
Anweisungsblock;
```

Wie auch schon von den Triggern bekannt, ermöglicht der Zusatz `OR REPLACE` das Überschreiben einer schon vorhandenen Prozedur gleichen Namens. In *Prozedurname* wird der Name der Prozedur definiert. Die Angabe eines Namens ist natürlich erforderlich. Optional hingegen ist die Verwendung einer Argumentenliste. Wie Ihnen von anderen Programmiersprachen bekannt sein dürfte, kann man einer Unterroutine eine (fast) beliebige Anzahl an Variablen übergeben. Bei der Deklaration der Prozedur werden in der Argumentenliste sowohl Name als auch Typ der Variablen angegeben. Mehrere Argumente werden jeweils durch Komma getrennt. Dieser Liste folgt das Schlüsselwort `IS`. Es leitet auch den Anweisungsblock ein. Ein solcher Anweisungsblock ist im Prinzip genauso aufgebaut, wie Sie ihn bei den Triggern schon kennengelernt haben. Die eigentlichen Anweisungen sind zwischen den Schlüsselwörtern `BEGIN` und `END` eingebettet. Vor dem ersten `BEGIN` kann man noch zusätzlich prozedurglobale Variablen definieren.

Die Syntax zur Funktionsdeklaration und -definition ist sehr ähnlich aufgebaut:

```
CREATE [OR REPLACE] FUNCTION Funktionsname
[Argumentenliste]
RETURN Datentyp IS
Anweisungsblock;
```


Nach der (optionalen) Argumentenliste muss bei der Funktionsdefinition noch formuliert werden, welchen *Datentyp* die Funktion zurückliefert. Dieser Datentyp kann einer der Typen sein, die Oracle auch zur Definition von Tabellen zulässt. Im Gegensatz zu anderen Datenbankengines wie beispielsweise Informix ist es nicht möglich, mehrere Werte oder Ergebnisse aus einer Funktion zurückzuliefern.

Hinweis:
Innerhalb der Argumentenliste wird lediglich der Datentyp selbst, nicht aber seine Länge mit angegeben!

Der Anweisungsblock weist im Vergleich zu Prozeduren ebenfalls einen Unterschied auf. Eine Funktion wird mit einer `RETURN`-Anweisung abgeschlossen. Hier wird genau die Variable auf den Stack gelegt, deren Typ zuvor im Funktionskopf deklariert wurde:

```
BEGIN
   ...
   RETURN Zurückgelieferte_Variable
END;
```

Im folgenden werden wir die im vorherigen Abschnitt erzeugten Trigger etwas modifizieren. Die Anweisungsblöcke innerhalb der beiden Trigger werden jeweils in eine Prozedur und eine Funktion ausgelagert. Im Anweisungsblock des Triggers erfolgt dann lediglich der Aufruf der entsprechenden Prozedur oder Funktion. Zunächst soll eine Prozedur geschrieben werden, die eine Änderung der Kundennr aus der Kundentabelle in der Historientabelle mitprotokolliert. Es folgt das entsprechende Script:

```
CREATE OR REPLACE PROCEDURE Write_History

(old VARCHAR2,
 new VARCHAR2)

BEGIN
  INSERT INTO historie
         (datum, kundennr, kundenneu)
  VALUES  (sysdate, old, new);
END;
/
```

Wenn Sie diese Anweisungen genau so in SQL*Plus eingeben, wird Oracle mit einer Warnmeldung an Sie herantreten, weil sich in die SQL-Anweisung ein kleiner Fehler eingeschlichen hat. Hinter der Argumentenliste fehlt das Wort „`IS`". Diesen Bug habe ich mit

Absicht eingebaut, um zwei Dinge zu zeigen. Zunächst einmal ist zu beachten, dass Oracle nicht direkt einen Syntaxfehler meldet, sondern lediglich die Meldung

```
Warning: Procedure created with compilation errors.
```

ausgibt. Über die Anweisung

```
show errors
```

können Sie SQL*Plus dazu veranlassen, die Compiler-Fehler auszugeben. Bei einer größeren Anzahl an Warnungen oder Fehlern möchte ich noch einmal auf das Spool-Kommando hinweisen, mit dem man die Ausgaben von SQL*Plus in eine Textdatei schreiben kann.

Stored Procedures werden in Oracle in compilierter Form abgelegt. Der Fehler bzw. die Warnmeldung, die hier ausgegeben wird, stammt von dem SQL-Compiler. Hieran wird noch einmal deutlich, dass die Verwendung von Stored Procedures einen enormen Geschwindigkeitsvorteil gegenüber der herkömmlichen Anwendungsprogrammierung mittels Embedded SQL bringt. Bei Embedded SQL können Sie lediglich SQL-Befehle an die Datenbank schicken, die dann zur Laufzeit erst compiliert werden müssen, bevor sie ausgeführt werden können! Bei der Verwendung von Stored Procedures senden Sie lediglich einen Aufruf der entsprechenden Prozedur an die Datenbank. Gerade bei kompliziert verschachtelten SQL-Abfragen, die vielleicht zehn oder mehr Tabellen miteinander verknüpfen, macht sich der Performance-Vorteil wirklich bemerkbar! Abgesehen davon beansprucht ein einzelner Aufruf einer Stored Procedure das Netzwerk wesentlich weniger als die komplette SQL-Abfrage. Bei der Anwendungsentwicklung sollten Sie dieses auch immer im Hinterkopf behalten. Stellen Sie sich bei der Datenbankentwicklung immer vor, dass mit Ihrer Anwendung 100 Benutzer gleichzeitig auf die Daten zugreifen müssen. Als zweites möchte ich noch auf den interessanten Compiler von Oracle hinweisen. Bei Entwicklungssystemen wie Borland Delphi oder Borland C++-Builder verweigert der Compiler seinen Dienst, sollte der Quelltext, mit dem er gefüttert wird, syntaktisch nicht korrekt sein. Oracle prangert zwar das fehlende „IS" an (Abbildung 5-11), ersetzt dieses jedoch anschließend. Die Warnung soll lediglich den Anwender darauf hinweisen, dass die Datenbank den Fehler bemerkt hat.

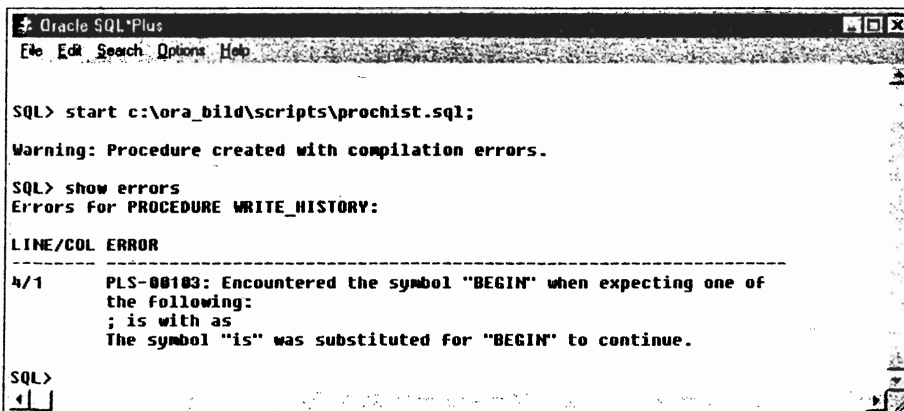

Abbildung 5-11: Compiler-Fehler von Oracle

Das folgende Script ist syntaktisch korrekt und enthält gleichzeitig eine neue Definition des Triggers *hist*. Im Internet finden Sie dieses Script unter dem Dateinamen PROCHIST.SQL:

```
CREATE OR REPLACE PROCEDURE Write_History

(old VARCHAR2,
 new VARCHAR2) IS

BEGIN
  INSERT INTO historie
         (datum, kundennr, kundenneu)
  VALUES (sysdate, old, new);
END;
/

CREATE OR REPLACE TRIGGER hist BEFORE
UPDATE ON kunden
FOR EACH ROW

BEGIN
   Write_History(:old.kundennr, :new.kundennr);
END;
/
```

Im Anweisungsblock dieses Triggers sehen Sie den Aufruf der Prozedur `Write_History`. Als Parameter werden die internen Trigger-Variablen übergeben, die zum Zeitpunkt des Aufrufes den neuen und den alten Wert der Kundennummer enthalten.

Das folgende Listing zeigt entsprechend das Erzeugen einer Funktion und die Modifikation des Triggers *test*. Bei dieser Aufgabe bot sich die Verwendung einer Funktion an, weil diese eben genau eine Variable zurückliefern kann. Im Internet finden Sie dieses Script unter dem Dateinamen FUNCSTAT.SQL:

```
CREATE OR REPLACE FUNCTION Write_Statistic
RETURN NUMBER
IS
   Anzahl_Kunden NUMBER;

BEGIN
   SELECT COUNT(Kundennr)
   INTO   Anzahl_Kunden
   FROM   kunden;
   RETURN Anzahl_Kunden;
END;
/
```

```
CREATE OR REPLACE TRIGGER test AFTER
INSERT OR DELETE ON kunden

BEGIN
   INSERT INTO statist
   (DATUM, ANZ_KUND)
   VALUES
   (sysdate, Write_Statistic);
END;
```

5.5.3 Parameterübergabe von Stored Procedures

Anhand des vorhergehenden Abschnitts haben Sie die grundlegenden Elemente von Stored Procedures kennengelernt. Es wird zwischen Prozeduren und Funktionen unterschieden, wobei Funktionen einen Wert bzw. eine Variable zurückliefern und Prozeduren dies eben nicht können. Nichtsdestotrotz gibt es auch bei der Verwendung von Prozeduren Möglichkeiten, auf deren Ergebnisse zurückzugreifen. Die einer Prozedur übergebenen Parameter können noch über einige Schlüsselwörter näher klassifiziert werden. Tabelle 5-3 gibt Auskunft über diese Schlüsselwörter:

Modus	Beschreibung
IN	Die in der Parameterliste über IN spezifizierten Parameter werden der Prozedur übergeben. Die Prozedur legt intern eine Kopie dieses Parameters an, so dass der ursprünglich übergebene Wert nicht verändert wird.
OUT	Hierüber wird ein Parameter als Zeiger übergeben. Dadurch kann auf den Inhalt dieses Parameter auch außerhalb einer Prozedur zugegriffen werden. Mit Hilfe des OUT-Modus können also auch Prozeduren anderen Programmteilen Variablen und Inhalte zur Verfügung stellen. Hierbei ist zu beachten, dass mit OUT spezifizierte Parameter keine Werte an eine Prozedur übergeben können. Sie dienen lediglich als Platzhalter bzw. Container für Berechnungen innerhalb der Prozedur. Der Wert der Variablen steht auch nach Verlassen der Prozedur noch in dem übergeordneten Modul zur Verfügung.
IN OUT	Hierbei handelt es sich um eine Kombination aus den Modi IN und OUT. Solche Parameter übergeben einer Prozedur also Werte, gleichzeitig können mit diesen Werten Berechnungen durchgeführt werden, die dann auch außerhalb der eigentlichen Prozedur zur Verfügung stehen.

Tabelle 5-3: Übergabemodi der Parameterliste

Die Prozedur WRITE_HISTORY würde demnach folgende Modi für die einzelnen Parameter erwarten:

```
CREATE OR REPLACE PROCEDURE Write_History

(old IN VARCHAR2,
 new IN VARCHAR2) IS

BEGIN
   INSERT INTO historie (datum, kundennr, kundenneu)
   VALUES               (sysdate, old, new);
END;
/
```

Mit Hilfe dieser differenzierteren Definition einzelner Parameter wäre es ebenfalls möglich, die Funktion WRITE_STATISTIC in eine Prozedur umzuschreiben:

```
CREATE OR REPLACE PROCEDURE Write_Statistic
       (Anzahl_Kunden OUT NUMBER)
IS
BEGIN
   SELECT COUNT(Kundennr)
   INTO   Anzahl_Kunden
   FROM   kunden;
END;
/
```

Bei dieser Art der Implementierung ist darauf zu achten, dass die Variable ANZAHL_KUNDEN außerhalb der Prozedur WRITE_STATISTIC deklariert wird, da der Prozedur lediglich ein Zeiger auf diese Variable übergeben wird. Sie ist außerhalb der Prozedur verfügbar, weil sie auch außerhalb deklariert wurde. Die Deklaration im übergeordneten Modul ist für OUT-Parameter zwingend erforderlich.

Neben der Definition von Parametern gibt es allerdings noch einige weitere interessante Dinge über die Parameterübergabe zu erfahren. PL/SQL kennt beispielsweise zwei grundsätzlich verschiedene Arten der Parameterübergabe:

- Übergabe per Position
- Übergabe per Name

Beide Arten sollen anhand eines einfachen Beispiels erläutert werden. Die folgende Prozedur kapselt die INSERT-Anweisung in die Kundentabelle. Anwender, die diese Tabelle

beschreiben möchten, sind also nicht zwangsläufig auf die Syntax der INSERT-Anweisung angewiesen:

```
CREATE OR REPLACE PROCEDURE insert_kunden(
                                 nr          NUMBER,
                                 name        VARCHAR2 := NULL,
                                 vorname     VARCHAR2 := NULL,
                                 straße      VARCHAR2 := NULL,
                                 plz         CHAR := NULL,
                                 anlage      DATE := NULL)

IS

BEGIN
  INSERT INTO Kunden
         (kundennr, name, vorname, straße, plz, anlage)
         VALUES
         (nr, name, vorname, straße, plz, anlage);
END;
/
```

Anhand der Variablendaklaration kann man erkennen, dass mit Ausnahme der Kundennummer jeder Parameter mit einem Standardwert vorbelegt wird. Die Prozedur könnte mit folgenden Parametern aus SQL*Plus heraus gestartet werden:

```
EXECUTE insert_kunden(106,   ← kein Zeilenumbruch
  'Heitsiek',
  'Stefan',
  'Postweg 17',
  '55555',
  sysdate);
```

Welcher der übergebenen Parameter zu welchem Parameter in der Deklarationsliste gehört, identifiziert der SQL-Parser anhand der Position. Er „weiß" durch die Deklaration, dass der erste übergebene Parameter vom Typ NUMBER ist, der zweite muss vom Typ VARCHAR2 sein usw.

Was passiert jetzt aber, wenn ein Anwender einen neuen Satz in die Kundentabelle einfügen möchte, aber nicht alle sechs Parameter ausfüllen kann, weil ihm bestimmte Informationen fehlen? Intuitiv habe ich zunächst folgende Syntax versucht, als ich das erste Mal auf dieses Problem stieß:


```
SQL> EXECUTE insert_kunden(111,'Gates',,,,sysdate);
```

Leider erwiderte die Datenbank diese Anweisung mit der Fehlermeldung:

```
BEGIN insert_kunden(111,'Gates',,,,sysdate); end;
                                 *

FEHLER in Zeile 1:
ORA-06550: Zeile 1, Spalte 33:
PLS-00103: Fand das Symbol "," als eines der folgenden erwartet wurde:
( - + mod not null others <ein Bezeichner>
<ein begrenzter Bezeichner in doppelten Anführungszeichen>
<eine Bindevariable> avg Zählung Aktuell Existiert max min
Vorher sql stddev Summe Abweichung Cast
<ein Zeichenkettenliteral mit Zeichensatzspezifikation>
<eine Zahl> <eine SQL-Zeichenkette in Hochkommata>

SQL>
```

Die Datenbank erwartet also immer eine vollständige Parameterliste, auch dann, wenn die entsprechenden Parameter in der Deklaration vorinitialisiert werden. Der SQL-Parser benötigt diese vollständige Liste, um die einzelnen Parameter durch ihre Position genau definieren zu können.

Dies ist oftmals jedoch sehr umständlich, gerade bei Funktionen und Prozeduren, denen eine große Anzahl an Parametern übergeben wird. Zu diesem Zweck kennt Oracle außerdem noch die *Identifikation über Name*. Hierdurch wird es möglich, einzelne Parameter anzugeben, ohne zusätzlich den gesamten Parameteroverhead mit Default-Werten füllen zu müssen. Das folgende Beispiel zeigt eine solche Parameterübergabe:

```
SQL> EXECUTE insert_kunden(nr => 110, anlage => sysdate);

PL/SQL-Prozedur wurde erfolgreich abgeschlossen.

SQL>
```

Über den „=>"-Operator wird der Name eines Parameters (im obigen Fall *nr* und *anlage* mit einem Wert verknüpft. Diese Art der Übergabe vereint gleich zwei Vorteile in sich. Zum einen ist es dadurch nicht mehr notwendig, die gesamte Liste zu füllen; zum anderen gestaltet sich der Quelltext wesentlich leserlicher. Implizit findet hierdurch eine Art der Dokumentation statt, die sich gerade bei umfangreichen PL/SQL-Projekten als sehr nützlich erwiesen hat.

5.5.4 Stored Procedures im Überblick

Analog zu den Triggern speichert Oracle den Quelltext jeder Prozedur oder Funktion in verschiedenen System-Tabellen. Für einen einfachen Überblick legt das System bei der Installation die View USER_SOURCE an. Es folgt die Beschreibung der einzelnen Felder:

```
SQL> DESC user_source;
 Name                            Null?    Type
 ------------------------------- -------- ----
 NAME                            NOT NULL VARCHAR2(30)
 TYPE                                     VARCHAR2(12)
 LINE                            NOT NULL NUMBER
 TEXT                                     VARCHAR2(2000)

SQL>
```

Die Spalte *name* enthält den Namen der Prozedur oder Funktion. *type* gibt an, ob es sich um eine Prozedur oder Funktion handelt. In *line* wird eine Zeilennummer des Quelltextes mitgeführt, der in der Spalte *text* abgelegt ist. Eine Übersicht über alle zur Zeit verfügbaren Prozeduren und Funktionen erhalten Sie über die folgende Anweisung:

```
SQL> SELECT DISTINCT name, type FROM user_source;

NAME                           TYPE
------------------------------ -----------
WRITE_HISTORY                  PROCEDURE
WRITE_STATISTIC                FUNCTION

SQL>
```

5.5.5 Stored Procedures aus SQL*Plus starten

Sobald Sie Stored Procedures einmal auf der Datenbank eingerichtet haben, können diese auch von anderen Anwendungen heraus aufgerufen werden. In SQL*Plus geschieht dies durch die Anweisung

```
EXECUTE
```


Dazu folgt ein Beispiel:

```
SQL> EXECUTE Write_History(70,88);

PL/SQL-Prozedur wurde erfolgreich abgeschlossen.

SQL>
```

5.5.6 Rekursive Prozeduren

PL/SQL erlaubt es auch, rekursive Prozeduren zu schreiben. Diese zeichnen sich dadurch aus, dass die Prozedur sich unter bestimmten Bedingungen selbst aufruft. Beispiel hierfür ist die Faktorisierung, wie Sie im folgenden Script implementiert ist. Sie finden dieses Script im Internet unter dem Dateinamen REKURSIV.SQL:

select fact(4) from dual

```
CREATE OR REPLACE FUNCTION fact(n NUMBER)
RETURN NUMBER IS

x NUMBER;
BEGIN
  IF n<0 THEN
     RETURN 0;
  END IF;
  IF n<=1 THEN
     RETURN 1;
  ELSE
     x := n*fact(n-1);
     RETURN x;
  END IF;
END;
/

SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE factor_test
IS

BEGIN
   DBMS_OUTPUT.ENABLE;
   DBMS_OUTPUT.PUT_LINE(fact(5));
END;
/
```

Schon im Vorgriff auf den Abschnitt 5.6.6 werden hier einige Standardprozeduren verwendet. Wenn Sie die Prozedur *factor_test* starten, erhalten Sie in SQL*Plus folgenden Ausdruck:

```
SQL> EXECUTE factor_test;
120

PL/SQL-Prozedur wurde erfolgreich abgeschlossen.

SQL>
```

5.5.7 Externe Prozeduren

Ab der Version 8 ist es möglich, innerhalb von PL/SQL *externe Prozeduren* zu implementieren. Es handelt sich hierbei um Routinen, die innerhalb von PL/SQL (und den entsprechenden Entwicklungsumgebungen) aufgerufen und verwendet werden können, deren Programmcode aber außerhalb der eigentlichen Datenbank definiert ist. In der Regel liegt dieser Code innerhalb einer dynamischen Linkbibliothek, auf die die Datenbankengine zur Laufzeit Zugriff hat. Damit Sie den PL/SQL-Dialekt um eigene Routinen erweitern können, ist eine Reihe von Schritten durchzuführen, die im folgenden beschrieben werden.

Zunächst muss die Funktion, die später in PL/SQL zur Verfügung stehen soll, programmiert werden. Man kann natürlich auch schon existierende DLLs einbinden; hier muss man allerdings genaue Informationen über deren Aufbau besitzen. Dazu empfiehlt sich in der Regel das Header-File, in dem die Funktionsköpfe deklariert sind.

Als Entwicklungsumgebung für die DLLs empfiehlt Oracle selbst einen C-Compiler, da alle Funktionsaufrufe nach der C-Konvention erfolgen. Das folgende Script wurde allerdings mit Borland Delphi erstellt und genügt den Anforderungen voll und ganz (zu finden im Internet im Verzeichnis DLL):

```
library dll;

uses
  Windows,
  IniFiles;

procedure CREATEINIT(i : Integer);
var
  IniFile : TIniFile;
begin
  IniFile := TIniFile.Create('c:\test.ini');
  IniFile.WriteInteger('Section', 'Test', i);
  IniFile.Free;
end;
```



```
procedure PEEP;
begin
  MessageBeep(MB_ICONASTERISK);
end;

exports
  CREATEINIT,
  PEEP;

end.
```

Innerhalb dieses Listings werden zwei Prozeduren definiert. Die Routine CREATEINIT realisiert einen einfachen Datei-Zugriff und erzeugt die Datei TEST.INI, sofern diese noch nicht vorhanden ist. Hierbei ist zu beachten, dass dieses Verzeichnis nicht zwingend für die Datenbankengine mittels des Parameters `UTL_FILE_DIR` freigegeben werden muss. Dieser Parameter ist nur zu definieren, falls ein direkter Datei-Zugriff aus einer PL/SQL-Routine heraus realisiert werden soll (vgl. Abschnitt 5.6.7). Die in externen DLLs zur Verfügung gestellten Funktionen unterliegen also nicht bzw. nur teilweise den Sicherheitsmechanismen der Datenbank. Es ist natürlich möglich, auf Datenbankebene zu definieren, wer eine bestimmte externe Prozedur ausführen darf.

Die zweite Prozedur PEEP erzeugt einen Piepton, der über den PC-Speaker ausgegeben wird.

Hinweis:
Während C bzw. C++ case sensitive arbeitet, findet in Delphi keine Unterscheidung zwischen Groß- und Kleinschreibung statt. Da in PL/SQL aber alle Objekt-Bezeichnungen in Großbuchstaben gespeichert werden, empfiehlt es sich, diese Schreibweise zu übernehmen, um etwaige Compiler-Probleme bezüglich der Schreibweise schon im Vorfeld auszuräumen.

Im nächsten Schritt muss die DLL der Datenbank bekannt gemacht werden. Innerhalb der Systemtabellen wird dabei eine Zuordnung zwischen der externen DLL und dem Pfad der Datei definiert. Eine solche Bibliothek erzeugt man mit

```
CREATE LIBRARY Bibliotheksname IS
 'Pfad_der_DLL';
```

Hier wird zunächst nur ein Datensatz innerhalb der Systemtabellen erzeugt, es erfolgt keine Prüfung, ob sich die DLL tatsächlich in dem durch Pfad_der_DLL spezifizierten Verzeichnis befindet. Es erfolgt auch keine Prüfung, ob es sich überhaupt um ein gültiges Verzeichnis handelt. Pro DLL ist eine Bibliothek zu erzeugen.
Entsprechend wird über eine DROP-Anweisung die Bibliothek wieder gelöscht:

```
DROP LIBRARY Bibliotheksname;
```

Hinweis:
Die Einträge von Bibliotheken findet man in den Systemtabellen ALL_LIBRARIES, DBA_LIBRARIES und USER_LIBRARIES.

Im nächsten Schritt muss jede einzelne Prozedur der Bibliothek der Datenbankengine bekannt gemacht werden. Dies geschieht über die bekannte Anweisung CREATE PROCEDURE. Hier die vollständige Syntax:

```
CREATE [OR REPLACE] PROCEDURE Prozedurname
[Argumentenliste]
AS EXTERNAL
LIBRARY Bibliotheksname;
[NAME Externer_Name]
[LANGUAGE Sprache]
[CALLING STANDARD {C | PASCAL}]
[WITH CONTEXT]
[PARAMETERS Parameterliste]
```

Über das Schlüsselwort AS EXTERNAL wird die Prozedur als extern deklariert. In welcher DLL sie zu finden ist, legt der *Bibliotheksname* fest, der zuvor erzeugt wurde. In der *Argumentenliste* werden die formalen Parameter definiert, die der Funktion übergeben werden. Diese Liste sollte kongruent zu der Argumentenliste des Quelltexts der DLL sein. Die Angabe eines Namens (*Externer_Name*) ist nur dann erforderlich, wenn der *Prozedurname* nicht mit dem Namen der Routine innerhalb der DLL übereinstimmt. *Sprache* definiert die Sprache, in der die DLL geschrieben ist. Der Parameter CALLING STANDARD definiert die Art der Parameterübergabe zwischen der Prozedur. Erfolgt keine Angabe, erfolgt die Parameterübergabe nach C-Konvention, ansonsten nach Pascal-Konvention, d.h. die Parameter werden in umgekehrter Reihenfolge auf den Stack geschoben.

Über die Angabe WITH CONTEXT kann die externe Prozedur Zugriff auf eine Reihe von Umgebungsvariablen erhalten. Innerhalb des Quelltextes der entsprechenden Prozedur muss zusätzlich ein Zeiger vom Typ

```
OCIExtProcContext *with_context
```

definiert werden. Über diesen Zeiger erhält die Funktion Zugriff auf eine Struktur, in der verschiedene Umgebungsvariablen der jeweiligen Sitzung abgespeichert sind. Über PARAMETERS kann optional eine Datentyp-Umsetzung initialisiert werden. Da die verschiedenen Datentypen auf unterschiedlichen Systemen und Entwicklungsumgebungen unterschiedlich definiert sein können, kommt man in besonderen Fällen um solch eine Umsetzung nicht umhin.

Das folgende Script erzeugt eine Bibliothek zu der oben beschriebenen DLL und definiert ebenfalls entsprechende Prozedurköpfe. Im Internet ist das Script unter dem Dateinamen EXTERNAL.SQL zu finden.

```
DROP LIBRARY ext;
CREATE LIBRARY EXT IS 'c:\ora_8\dll\dll.dll';

CREATE OR REPLACE PROCEDURE CREATEINIT( i BINARY_INTEGER)
AS EXTERNAL
LIBRARY ext
name "CREATEINIT";
/
```



```
CREATE OR REPLACE PROCEDURE PEEP
AS EXTERNAL
LIBRARY ext
name "PEEP";
/
```

Danach können Sie die Prozeduren in gleicher Weise aufrufen und nutzen, wie Sie auch die in Abschnitten zuvor erstellen Routinen verwenden können. Zur Laufzeit muss nur sichergestellt werden, dass sich die DLL in dem in der Bibliothek definierten Verzeichnis befindet.

5.6 Pakete

5.6.1 Allgemeines

Oracle bietet die Möglichkeit, *Pakete* („packages“) zu erzeugen. In einem solchen Paket sind mehrere Stored Procedures zu einer Einheit zusammengefasst. Hierüber haben Sie die Möglichkeit, eigene Routinen in logische Einheiten zusammenzufassen. So wäre es beispielsweise denkbar, ein Paket mit bestimmten Stored Procedures für eine spezifische Datenbankanwendung zu erzeugen, ein anderes Paket mit nützlichen Routinen, die anwendungsglobal zur Verfügung stehen. Das Erzeugen solcher Packages erinnert an die Programmierung in C bzw. in C++. Der Quelltext eines C-Programmes teilt sich in der Regel in zwei Dateien auf:

- die Header-Datei und
- den eigentlichen Quelltext.

In der Header-Datei sind lediglich die Funktions-Deklarationen abgelegt, in dem eigentlichen Quelltext sind diese dann definiert. Beide Dateien zusammen bilden eine Einheit. Ähnlich verhält es sich mit den Paketen unter Oracle. Hier teilt man ein Paket auf in

- eine Spezifikation und
- in den Paketrumpf.

Die Spezifikation entspricht der Header-Datei, der Paketrumpf dem eigentlichen Quelltext der C-Funktionen.

5.6.2 Paketspezifikationen erzeugen

Zunächst muss die Spezifikation eines Paketes definiert werden. Hierin werden der Name des Paketes definiert und alle Prozeduren und Funktionen, die der Paketrumpf enthält, deklariert. Die allgemeine Syntax für diesen Vorgang lautet:

```
CREATE PACKAGE Paketname>P> IS
Deklarationsliste
END Paketname;
```

Paketname definiert hierbei einen eindeutigen Namen für ein Paket, die *Deklarationsliste* enthält eine Aufzählung über alle Funktionen und Prozeduren, die dieses Paket beinhalten soll. Es ist außerdem möglich, Variablen zu deklarieren, die nur innerhalb eines Paketes Gültigkeit besitzen. Die Deklaration dieser Variablen erfolgt ebenfalls an dieser Stelle.

5.6.3 Paketrümpfe erzeugen

Analog zu C bzw. C++ erfolgt nach dem Erzeugen der Header-Datei bzw. der Paketspezifikation die Definition der Routinen. Die allgemeine Syntax lautet hier:

```
CREATE PACKAGE BODY Paketname IS
Variablendefinitionen
Prozedurdefinitionen
Funktionsdefinitionen
END Paketname
```

Paketname enthält wieder den Namen des zu definierenden Paketes. Danach erfolgt die Definition der einzelnen Routinen. Das folgende Listing erzeugt ein Paket, das die beiden Routinen enthält, die im Abschnitt zuvor erarbeitet wurden. Im Internet finden Sie diese Datei unter dem Namen PAKETE.SQL:

```
CREATE OR REPLACE PACKAGE Useful_Package IS
FUNCTION Write_Statistic
         RETURN NUMBER;

PROCEDURE Write_History
          (old varchar2,
           new varchar2);
END Useful_Package;
/

CREATE OR REPLACE PACKAGE BODY Useful_Package IS
```



```
FUNCTION Write_Statistic
RETURN NUMBER
IS
   Anzahl_Kunden NUMBER;
BEGIN
   SELECT COUNT(Kundennr)
   INTO   Anzahl_Kunden
   FROM   KUNDEN;
   RETURN Anzahl_Kunden;
END;

PROCEDURE Write_History
        (old VARCHAR2,
         new VARCHAR2) IS
BEGIN
   INSERT INTO historie
   (datum, kundennr, kundenneu)
   VALUES
   (sysdate, old, new);
END;

END;
/
```

Wenn Sie eine Prozedur oder eine Funktion aus einem Paket heraus aufrufen möchten, so ist dem Namen der Routine zusätzlich der Paketname voranzustellen, z.B.:

```
Useful_Package.Write_Statistic
```

Die Funktionalität von Paketen ähnelt dem der *Dynamic Link Libraries* unter Windows. Wenn Sie die obige Anweisung an die Datenbank senden, überprüft Oracle, ob der ausführbare Code der Funktion sich vielleicht noch von einem früheren Aufruf her im Arbeitsspeicher befindet. Ist er dort nicht vorhanden, wird dieser in den Hauptspeicher des Datenbankservers geladen und dort ausgeführt. Zu beachten ist, dass nicht nur der ausführbare Code der einen Routine sondern der ausführbare Code des gesamten Paketes geladen wird. Der erste Aufruf einer Routine aus einem noch nicht im Hauptspeicher befindlichen Paketes dauert dafür zwar etwas länger. Danach stehen aber sämtliche Routinen des Paketes wesentlich schneller zur Verfügung. Natürlich ist der Hauptspeicher eines Datenbankservers nicht unbegrenzt, so dass ein Paket nicht über die gesamte Laufzeit der Datenbank im Speicher gehalten werden kann. Hier verfügt Oracle über einen Caching-Algorithmus, der nach den Paketen (und einigen anderen Dingen) sucht, die längere Zeit nicht mehr aufgerufen wurden. Diese Elemente werden dann gelöscht, um Platz für neue, benötigte Routinen zu schaffen.

Es besteht allerdings auch die Möglichkeit, ein Paket statisch in den Hauptspeicher zu laden. In einem solchen Fall wird das Paket über die gesamte Laufzeit der Datenbank im (schnellen) Hauptspeicher gehalten. Der Caching-Algorithmus von Oracle beachtet dieses Paket dann nicht weiter.

5.6.4 Pakete löschen

Über die Anweisung

`DROP PACKAGE Paketname`

löschen Sie wieder ein Paket. Dabei wird sowohl die Paketspezifikation als auch der Paketrumpf aus der Datenbank entfernt.

5.6.5 Pakete im Überblick

Bei der Verwendung von Paketen ergibt sich die gleiche Problematik, wie wir Sie auch schon bei Triggern und Stored Procedures kennengelernt haben. Man sieht einer Datenbank nicht unbedingt auf den ersten Blick an, welche Pakete zum aktuellen Zeitpunkt aktiviert sind. Hier hilft uns auch wieder eine Abfrage über die Systemtabellen hinweg. Das folgende Listing zeigt die Abfrage nebst zugehöriger Ergebnismenge:

```
SQL> DESC user_objects
 Name                           Null?    Type
 ------------------------------ -------- ----
 OWNER                                   VARCHAR2(30)
 OBJECT_NAME                             VARCHAR2(128)
 SUBOBJECT_NAME                          VARCHAR2(30)
 OBJECT_ID                               NUMBER
 DATA_OBJECT_ID                          NUMBER
 OBJECT_TYPE                             VARCHAR2(15)
 CREATED                                 DATE
 LAST_DDL_TIME                           DATE
 TIMESTAMP                               VARCHAR2(19)
 STATUS                                  VARCHAR2(7)
 TEMPORARY                               VARCHAR2(1)
 GENERATED                               VARCHAR2(1)

SQL>

SQL> SELECT object_name FROM user_objects
  2  WHERE object_type='PACKAGE';

OBJECT_NAME
---------------------------------------------
DBMS_ALERT
DBMS_APPLICATION_INFO
DBMS_AQ
DBMS_AQADM
```



```
DBMS_AQ_IMPORT_INTERNAL
DBMS_BACKUP_RESTORE
DBMS_DDL
DBMS_DEBUG
DBMS_DEFER_IMPORT_INTERNAL
DBMS_DEFER_SYS
DBMS_DESCRIBE
DBMS_DISTRIBUTED_TRUST_ADMIN
DBMS_EXPORT_EXTENSION
DBMS_IJOB
DBMS_INTERNAL_TRIGGER
DBMS_IREFRESH
DBMS_ISNAPSHOT
DBMS_JOB
DBMS_LOB
DBMS_LOCK
DBMS_OUTPUT

OBJECT_NAME
---------------------------------------------
DBMS_PIPE
DBMS_PITR
DBMS_PSWMG_IMPORT
DBMS_RCVMAN
DBMS_REFRESH
DBMS_ROWID
DBMS_SESSION
DBMS_SNAPSHOT
DBMS_SNAPSHOT_UTL
DBMS_SNAP_INTERNAL
DBMS_SNAP_REPAPI
DBMS_SPACE
DBMS_SQL
DBMS_STANDARD
DBMS_SYSTEM
DBMS_SYS_ERROR
DBMS_SYS_SQL
DBMS_TRANSACTION
DBMS_UTILITY
DIANA
DIUTIL

OBJECT_NAME
---------------------------------------------
PBREAK
PBRPH
PBSDE
```

```
PBUTL
PIDL
PLITBLM
STANDARD
UTL_FILE
UTL_HTTP
CTX_ACCESS
CTX_ADM
CTX_BIN
CTX_DDL
CTX_DML
CTX_INFO
CTX_LING
CTX_QUERY
CTX_SVC
CTX_THES
CTX_THS
CTX_UDA

OBJECT_NAME
-----------------------------------------------
CTX_VP
DISPATCHER
DMLQ
DMLTRIG
DRASERM
DRDBG
DRDINTE
DRIG
DRQ_INT_TRIG
DRUE
DR_DEF
DR_IDX
DR_REC
DR_REWRITE
DR_RTM
DR_UTL
LISTENER
PIPE
SRCQ
SVCQ
EMPLOYEE

OBJECT_NAME
-----------------------------------------------
DR_REWRITE
CSR_POINTER
```



```
EMPLOYEE
USEFUL_PACKAGE

88 Zeilen ausgewählt.

SQL>
```

Sie sehen: Neben dem von uns implementierten Paket USEFUL_PACKAGE ist innerhalb der Datenbank eine Reihe weiterer Pakete definiert und aktiviert. Es handelt sich hierbei um Pakete, die zum Lieferumfang der Datenbankinstallation gehören. Mittlerweile ist die Sammlung an mitgelieferten Paketen schon erheblich gestiegen. Während die Version 7 von Oracle mit ca. 30 Paketen installiert wurde, erhält man mit Oracle 8 schon fast 90. Einige der wichtigsten möchte ich im folgenden aufgreifen und näher erläutern.

5.6.5.1 Das Paket DBMS_OUTPUT

Wie in Abschnitt 5.6.5 gesehen, können Sie als Datenbankadministrator schon auf eine Reihe nützlicher Pakete mit diversen Routinen zurückgreifen. Die entsprechenden Script-Dateien finden Sie im Ordner *RdbmsXX\Admin* der jeweiligen Installation des DBMS („XX" steht dabei für die Versionsnummer der Datenbank). Als sehr nützlich bei meiner Arbeit mit Oracle-Datenbanken hat sich das Paket DBMS_OUTPUT erwiesen. In diesem Paket sind Routinen zusammengefasst, die eine Ausgabe von Informationen innerhalb von SQL*Plus erlauben. Im Hinblick auf die fehlenden Debugging-Funktionalitäten innerhalb von PL/SQL handelt es sich hierbei um äußerst nützliche Routinen. Tabelle 5-4 gibt einen Überblick über die wichtigsten Routinen innerhalb dieses Paketes.

Prozedur	Beschreibung
procedure enable (buffer_size in integer default 20000);	Initialisiert einen Textausgabepuffer. Muss einmal aufgerufen werden, bevor die anderen Funktionen genutzt werden können.
procedure disable;	Gibt die allozierten Ressourcen wieder frei.
procedure put(a varchar2);	Gibt ein Wort aus.
procedure put(a number);	Gibt eine Zahl aus.
procedure put(a date);	Gibt ein Datum aus.
procedure put_line(a varchar2);	Gibt eine Textzeile (mit CR/LF) aus.
procedure put_line(a number);	Druckt eine Zahl und führt CR/LF aus.
procedure put_line(a date);	Druckt ein Datum und führt CR/LF aus.
procedure new_line;	Führt ein CR/LF aus.
procedure get_line(line out varchar2, status out integer);	Liest eine Zeile aus dem Textausgabepuffer.

Tabelle 5-4: Routinen innerhalb von DBMS_OUTPUT

Die beiden wichtigsten Prozeduren sind ohne Zweifel

- `dbms_output.enable`
- `dbms_output.putline`.

Das folgende Listing zeigt ein einfaches Beispiel mit diesen beiden Routinen, wie innerhalb von SQL*Plus Textausgaben generiert werden können. Dieses Listing finden Sie im Internet unter der Bezeichnung OUTPUT.SQL:

```
set serveroutput on;

CREATE OR REPLACE procedure Hello
IS

BEGIN

dbms_output.enable;
dbms_output.put_line('Hello world!');
/* dbms_output.disable; */
END;
/
```

Abbildung 5-12 zeigt das Ergebnis dieser Prozedur.

```
Oracle SQL*Plus
File Edit Search Options Help

SQL*Plus: Release 3.2.2.0.1 - Production on Sat Oct 04 09:53:34 1997

Copyright (c) Oracle Corporation 1979, 1994.  All rights reserved.

Connected to:
Personal Oracle7 Release 7.2.2.3.1 - 90 day trial license
To purchase a production license, call 1-800-633-0586 (U.S. only)

With the distributed and replication options
PL/SQL Release 2.2.2.3.1 - Production

SQL> @output

Procedure created.

SQL> execute hello
Hello world!

PL/SQL procedure successfully completed.

SQL> |
```

Abbildung 5-12: *Textausgaben mittels DBMS_OUTPUT*

Sie werden sich vielleicht fragen, weshalb die Zeile `dbms_output.disable;` in Kommentarzeilen gefasst ist. In der Regel sollte man als Programmierer darauf achten, dass alle manuell im Programm reservierten Speicherbereiche auch wieder freigegeben werden. So ist beispielsweise jedes Objekt, welches unter Delphi mittels der Methode *Create* erzeugt wurde, über *Free* wieder freizugeben. Aus diesem Grund sollte man vermuten, dass genau das gleiche auch mit den Objekten unter Oracle geschehen sollte, d.h. vor dem Verlassen der Prozedur sollte der Textausgabepuffer wieder freigegeben werden. Allerdings stellt sich dann das Problem, dass die Textausgabe überhaupt nicht erfolgt bzw. dass die Textausgabe zwar erfolgt, den Bruchteil einer Sekunde später allerdings schon wieder gelöscht wird. Eine Freigabe dieses Puffers mittels `dbms_output.disable` sollte also erst wirklich am Ende des Programmes erfolgen.

5.6.5.2 Das Paket UTL_FILE

Mit den in diesem Paket definierten Funktionen und Prozeduren kann man Textdateien erzeugen und löschen sowie diese beschreiben und auslesen. Dazu wird eine Reihe von Funktionen und Prozeduren zur Verfügung gestellt, die in dieser Form einen Dateizugriff auf unterschiedlichen Server-Plattformen realisieren, d.h. es spielt keine Rolle, ob das DBMS auf einem Unix- oder einem NT-Server installiert ist. Dieses Package steht allerdings erst ab der Version 7.3 zur Verfügung.

Im Prinzip stellen die in diesem Paket implementierten Routinen alles zur Verfügung, was man als PL/SQL-Programmierer benötigt, um eine gesamte Datenbank zu exportieren. Diese Funktionen sind also mit einer gehörigen Portion Vorsicht zu genießen. Um dennoch die größtmögliche Sicherheit zu gewährleisten, können die über dieses Paket erzeugten Dateien nur in ganz bestimmte Verzeichnisse auf dem Server geschrieben werden. Welche Verzeichnisse das sind, wird in der INIT.ORA bzw. in der INITxxxx.ORA definiert, wobei die „xxxx" für die Bezeichnung der Instance stehen. In dieser Datei muss der Parameter `UTL_FILE_DIR` für alle Verzeichnisse definiert werden, für die ein Schreibvorgang erlaubt sein soll. Die beiden folgenden Einträge in der Datei sorgen dafür, dass die Verzeichnisse C:\TMP und C\USER\EXPORT beschrieben werden können:

```
UTL_FILE_DIR=c:\tmp
UTL_FILE_DIR=c:\user\export
```

Hinweis:
Es ist an dieser Stelle nicht möglich, Umgebungsvariablen wie beispielsweise %ORAHOME% einzusetzen. Es bleibt nur die Möglichkeit, die Pfade „hart" zu codieren. Im Zusammenhang mit den folgenden Funktionen ist es auch nicht möglich, den Dateizeiger manuell zu positionieren. Die Aktualisierung des Zeigers erfolgt implizit mit dem Aufruf der entsprechenden Funktionen.

Neben der dezidierten Bestimmung der freigegebenen Laufwerke besteht außerdem die Möglichkeit, alle Serverlaufwerke freizugeben:

```
UTL_FILE_DIR=*
```

Im folgenden werden die wichtigsten Routinen dieses Pakets aufgelistet:

```
FUNCTION fopen(location  IN VARCHAR2,
               filename  IN VARCHAR2,
               open_mode IN VARCHAR2) RETURN file_type
```

Mittels `fopen()` wird eine Datei geöffnet bzw. erzeugt. *location* gibt dabei den Pfad an, in dem sich die Datei befindet bzw. wo sie neu erzeugt werden soll. Dieser Parameter wird mit den zulässigen Werte verglichen, die in der INITxxxx.ORA definiert sind. *filename* definiert den Dateinamen mit Endung und *open_mode* den Modus. Mit dem Modus 'r' wird die Datei zum Lesen, mit 'w' zum Schreiben und mit 'a' zum Anhängen geöffnet. Zu beachten ist, dass der Modus 'w' eventuell vorhandene Daten in der zum Schreiben geöffneten Datei überschreibt, denn der Datensatzzeiger wird durch diese Einstellung an den Anfang der Datei positioniert. Der Modus 'r' kann nur eingesetzt werden, wenn die Datei schon vorhanden ist. Diese Funktion liefert ein Dateihandle zurück, das alle anderen Funktionen und Prozeduren verwenden.

```
FUNCTION is_open (file IN file_type) RETURN BOOLEAN
```

`is_open()` liefert einen Wahrheitswert zurück, der angibt, ob die durch `file` identifizierte Datei geöffnet ist (TRUE) oder nicht (FALSE).

```
PROCEDURE fclose(file IN OUT file_type)
```

Diese Prozedur schliesst die durch `file` identifizierte Datei.

```
PROCEDURE fclose_all
```

Hiermit werden alle zur Zeit des Aufrufs geöffneten Dateien wieder geschlossen. Diese Prozedur sollte allerdings nur in Notfällen verwendet werden, denn hierüber werden nicht die Dateihandles geschlossen. `is_open()` würde also weiterhin TRUE zurückliefern!

```
PROCEDURE get_line(file   IN file_type,
                   buffer OUT VARCHAR2)
```

Hierüber wird eine Zeile aus der Datei gelesen und in einen Puffer geschrieben. Die auszulesende Datei wird dabei wieder über das Dateihandle identifiziert. In dem Parameter `buffer` befindet sich nach Ausführung die Textzeile, sofern keine Exception ausgelöst wurde.

```
PROCEDURE put (file   IN file_type,
               buffer IN VARCHAR2)
```

Die Prozedur `put()` schreibt die in `buffer` übergebenen Zeichen in die Datei, die durch `file` identifiziert wird. Die Zeichen werden an der Stelle in die Datei geschrieben, an der sich momentan der Dateizeiger befindet. Es wird nicht automatisch ein Zeilenumbruch eingefügt.


```
PROCEDURE new_line(file  IN file_type,
                   lines IN NATURAL := 1)
```

Hierüber wird an die aktuelle Position innerhalb der Datei ein Zeilenumbruch eingefügt. `file` identifiziert dabei wieder die Datei. Mit Hilfe des Parameters `lines` kann definiert werden, wie viele Zeilenumbrüche erzeugt werden sollen. Standardmäßig ist dieser Parameter mit '1' vorbelegt, so dass er nicht bei jedem Aufruf mit angegeben werden muss.

```
PROCEDURE put_line(file   IN file_type,
                   buffer IN VARCHAR2)
```

Diese Prozedur ist zusammengesetzt aus den Prozeduren `put()` und `new_line()`, d.h. es werden die in buffer übergebenen Zeichen in die Datei geschrieben. Danach erfolgt ein Zeilenumbruch.

```
PROCEDURE putf(file   IN file_type,
               format IN VARCHAR2,
               arg1   IN VARCHAR2 DEFAULT NULL,
               arg2   IN VARCHAR2 DEFAULT NULL,
               arg3   IN VARCHAR2 DEFAULT NULL,
               arg4   IN VARCHAR2 DEFAULT NULL,
               arg5   IN VARCHAR2 DEFAULT NULL)
```

Die Prozedur `putf()` ähnelt sehr stark der C-Funktion `fprintf()`. Mit ihr ist es möglich, einen formatierten String in eine Datei zu schreiben. Die Datei wird wiederum durch `file` identifiziert. In `Format` wird der zu schreibende String übergeben. Innerhalb dieses Strings sind zwei besondere Zeichen definiert, '\n' erzeugt dabei einen Zeilenumbruch, und '%s' dient als Platzhalter für die eventuelle übergebenen Parameter in `arg1..arg5`. Soll beispielsweise der String:

„Sie nutzen Oracle 8.0
seit dem 1.1.1998"

ausgegeben werden, wobei 'Oracle 8.0' und '1.1.1999' als Parameter übergeben werden sollen, ist die Prozedur folgendermaßen aufzurufen:

```
putf(file,'Sie nutzen Oracle %s\n seit dem %s',
     'Oracle 8.0', '1.1.1998');
```

```
PROCEDURE fflush(file IN file_type)
```

Hierüber wird der Textpuffer geleert. Sofern sich also noch Daten in diesem Cache befinden, werden sie physikalisch auf die Platte geschrieben.

Keine Programmiersprache ohne das obligatorische „Hello world". Sofern Sie die INITxxxx.ORA dahingehend erweitert haben, dass das Serverlaufwerk freigegeben wurde, können Sie die folgende Prozedur auf der Datenbank erzeugen:

```
CREATE OR REPLACE PROCEDURE Hello_world
IS
  fHandle  UTL_FILE.FILE_TYPE;

BEGIN
  fHandle := UTL_FILE.FOPEN('c:\tmp', 'hello.txt','w');

  UTL_FILE.PUT_LINE(fHandle, 'Hello world!');
  UTL_FILE.FCLOSE(fHandle);
END;
/
```

Das folgende Beispiel zeigt, wie man eine einfache Export-Funktion mit diesen Routinen realisieren kann:

```
CREATE OR REPLACE PROCEDURE export_kunden
IS
  fHandle  UTL_FILE.FILE_TYPE;
  cNr      kunden.kundennr%type;
  cName    kunden.name%type;
  cVorname kunden.vorname%type;

  CURSOR  csr_kunden IS
          select kundennr, name, vorname FROM kunden;
BEGIN
  fHandle := UTL_FILE.FOPEN('c:\tmp', 'kunden.txt','w');

  OPEN csr_kunden;
  LOOP
      FETCH csr_kunden
      INTO  cNr, cName, cVorname;
      EXIT  WHEN csr_kunden%NOTFOUND;

      UTL_FILE.PUT(fHandle, cNr);
      UTL_FILE.PUT(fHandle, ',');
      UTL_FILE.PUT(fHandle, cName);
      UTL_FILE.PUT(fHandle, ',');
      UTL_FILE.PUT_LINE(fHandle, cVorname);
  END LOOP;

  UTL_FILE.FCLOSE(fHandle);
END;
/
```


Demgegenüber jetzt die entsprechende Import-Routine. Der Einfachheit halber werden bestehende Datensätze lediglich aktualisiert. Ein Hinzufügen zu den bestehenden Sätzen der Tabelle wäre möglich, allerdings benötigte man dann eine Sequenz zur Vergabe der eindeutigen Kundennummern. Das entsprechende Script finden Sie im Internet unter dem Namen IMPORT.SQL:

```
CREATE OR REPLACE PROCEDURE import_kunden
IS
  fHandle  UTL_FILE.FILE_TYPE;

  nNr      kunden.kundennr%type;
  cName    kunden.name%type;
  cVorname kunden.vorname%type;

  cZeile   VARCHAR2(255);              /*max. 2000 Zeichen*/
  nKomma1  NUMBER;
  nKomma2  NUMBER;

BEGIN
  fHandle := UTL_FILE.FOPEN('c:\tmp', 'kunden.txt','r');

  LOOP
      BEGIN
         UTL_FILE.GET_LINE(fHandle, cZeile);
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
              EXIT;
      END;

      /* in nKommaX merke ich mir die Position  */

      nKomma1 := INSTR(cZeile,',', 1, 1);
      nKomma2 := INSTR(cZeile,',', 1, 2);

      nNr      := TO_NUMBER(SUBSTR(cZeile, 1, nKomma1 - 1));
      cName    := SUBSTR(cZeile, nKomma1 + 1, nKomma2 - nKomma1 - 1);
      cVorname := SUBSTR(cZeile, nKomma2 +1);

      UPDATE kunden
      SET name = cName,
          vorname = cVorname
      WHERE kundennr = nNr;
  END LOOP;

  UTL_FILE.FCLOSE(fHandle);
  COMMIT;
END;
/
```

5.6.5.3 Das Paket DBMS_RANDOM

Mit Hilfe der in diesem Paket definierten Prozeduren kann man Zufallszahlen erzeugen. Der Generator ist dazu zunächst über die Prozedur

```
PROCEDURE initialize (seed IN BINARY_INTEGER)
```

zu initialisieren. `seed` definiert dabei einen zufälligen Zahlenwert, den der Programmierer frei vergeben kann. Mit Hilfe dieses Parameters wird der Rechenalgorithmus initialisiert, der die Zufallszahl dann berechnet. Nachdem dies geschehen ist, kann über die Funktion

```
FUNCTION random RETURN BINARY_INTEGER;
```

eine Zufallszahl generiert werden. Ein Aufruf dieser Funktion liefert jeweils eine Zufallszahl zurück. `random` kann mehrmals hintereinander aufgerufen, die Initialisierungsprozedur muss jedoch nur einmal aufgerufen werden. Nachdem die benötigten Zufallszahlen ermittelt wurden, ist am Ende die Prozedur

```
PROCEDURE terminate
```

aufzurufen. Hierüber werden die allozierten Ressourcen wieder freigegeben.

5.6.5.4 Das Paket DBMS_JOB

Das Paket DBMS_JOB ist besonders wichtig. Mit den hier definierten Prozeduren ist es möglich, eine Job-Steuerung auf dem Datenbanksystem zu implementieren. Über solche Jobs wird der Aufruf bestimmter Stored Procedures zeitlich gesteuert. Das Einsatzgebiet dieser Jobsteuerung reicht von einfachen Wartungsfunktionen, die bestimmte Parameter der Datenbank überwachen, bis hin zu kompletten Schnittstellenprogrammierungen. Denkbar ist so der Einsatz der Dateiroutinen aus dem Paket UTL_FILE, so dass jeden Tag zu bestimmten Zeitpunkten eine Textdatei vom Server gelesen und die entsprechenden Daten in das DBMS geschrieben werden.

Die Verwaltung der einzelnen Jobs auf der Datenbank erfolgt über eine eindeutig identifizierende Job-Nummer. Diese Nummer wird entweder automatisch vergeben, oder man kann sie selbst definieren:

```
PROCEDURE isubmit (job       IN BINARY_INTEGER,
                   what      IN VARCHAR2,
                   next_date IN DATE,
                   interval  IN VARCHAR2 DEFAULT 'null')
```

Über diese Prozedur wird ein neuer Job auf der Datenbank angelegt. *`Job`* definiert dabei die eindeutige Job-Nummer, *`what`* ist der Name der Prozedur, die aufgerufen werden soll, und *`interval`* definiert die Zeitspanne bis zur nächsten Aktivierung.

Beispiel:
Sie haben eine Stored Procedure 'Hello_world' geschrieben, die jeden Tag zur gleichen Zeit aufgerufen werden soll. Mit Hilfe des folgenden Aufrufs implementieren Sie diese auf der Datenbank:

```
isubmit(1,'hello_world;', sysdate, 'sysdate+1');
```


Dabei ist zu beachten, dass hinter dem Prozedurnamen ein Semikolon eingefügt wird. Ansonsten kommt es zu einer kryptischen Fehlermeldung bei der Compilierung. Mit Hilfe der Prozedur

```
PROCEDURE submit (job       OUT BINARY_INTEGER,
                  what      IN  VARCHAR2,
                  next_date IN  DATE,
                  interval  IN  VARCHAR2 DEFAULT 'null')
```

kann man ebenfalls Jobs auf der Datenbank einrichten, allerdings wird im Gegensatz zu `isubmit()` hier nicht die Job-Nummer selbst vergeben, sondern vom System automatisch definiert (die Nummer wird von der Sequenz sys.jobseq geliefert). Anstelle der Job-Nummer wird hier also eine Variable gesetzt, die den Wert aufnehmen kann:

```
submit(:X, 'hello_world;', sysdate, 'sysdate+1');
```

Mit Hilfe der beiden obigen Prozeduren können Jobs eingerichtet werden. Über

```
PROCEDURE remove ( job IN BINARY_INTEGER)
```

wird ein Job, identifiziert durch die Job-Nummer, gelöscht.

```
PROCEDURE change (job       IN BINARY_INTEGER,
                  what      IN VARCHAR2,
                  next_date IN DATE,
                  interval  IN VARCHAR2 DEFAULT 'null')
```

ändert einen vorhandenen Job, d.h. es kann der Startzeitpunkt sowie das Intervall modifiziert werden. Es kann aber auch eine andere Prozedur für diesen Job definiert werden. Das gleiche kann auch die Prozedur

```
PROCEDURE what (job       IN BINARY_INTEGER,
                what      IN VARCHAR2)
```

Auch hierüber kann also eine neue, andere Stored Procedure für einen vorhandenen Job definiert werden. Über die Prozedur `next_date()` kann der Starttermin des nächsten, anstehenden Starts dieser Prozedur bzw. dieses Jobs modifiziert werden:

```
PROCEDURE next_date(job       IN BINARY_INTEGER,
                    next_date IN DATE)
```

Die Prozedur

```
PROCEDURE interval (job      IN BINARY_INTEGER,
                    interval IN VARCHAR2 DEFAULT 'null')
```

modifiziert das Intervall des durch `job` definierten Jobs.

Mit Hilfe der Prozedur

```
PROCEDURE broken (job        IN BINARY_INTEGER,
                  broken     IN BOOLEAN,
                  next_date  IN DATE DEFAULT SYSDATE)
```

kann ein Job deaktiviert werden, d.h. er ist noch auf der Datenbank definiert. Allerdings greift der Aktivierungsmechnismus nicht mehr. Zur Deaktivierung ist der Parameter broken auf TRUE zu setzen, mittels broken=FALSE wird der Job wieder aktiviert. Er wird dann gemäß dem in next_date übergebenen Zeitpunkt wieder gestartet.

Möchte man einen Job sofort zur Ausführung bringen, kann dies mit Hilfe der folgenden Prozedur geschehen:

```
PROCEDURE run (job      IN BINARY_INTEGER);
```

Das folgende Script (JOBS.SQL) richtet einen Job auf der Datenbank ein. Dieser Job ruft jede Stunde die Prozedur *Hello_world* auf, die wiederum den Text „Hello World!" in eine Textdatei auf dem Laufwerk C:\TMP schreibt:

```
CREATE OR REPLACE PROCEDURE Hello_world
IS
  fHandle  UTL_FILE.FILE_TYPE;
BEGIN
  fHandle := UTL_FILE.FOPEN('c:\tmp', 'hello.txt','a');
  UTL_FILE.PUT(fHandle, 'Hello world!');
  UTL_FILE.PUT_LINE(fHandle, to_char(sysdate));
  UTL_FILE.FCLOSE(fHandle);
END;
/

DECLARE
   job_number NUMBER;

BEGIN
   job_number := 11;

   dbms_output.enable;
   dbms_job.isubmit( job_number, 'hello_world;', sysdate, 'sysdate + (1/24)');

   dbms_output.put_line(job_number);
   dbms_job.run(job_number);
END;
/
```


Der Datenbankexplorer von Borland Delphi zeigt auf sehr bequeme Weise die auf der Datenbank vorhandenen Jobs an, wie die Abbildung 5-13 zeigt.

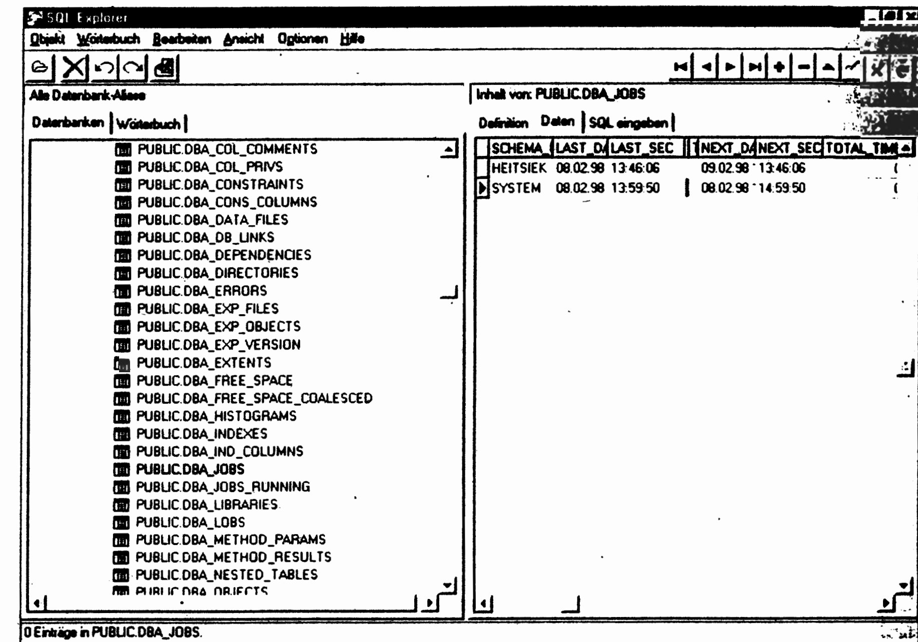

Abbildung 5-13: *Jobs auf der Datenbank*

Falls dem Leser dieses Programm nicht zur Verfügung steht, kann er auch direkt das Data Dictionary von Oracle abfragen. Die relevanten Informationen zu Jobs findet man in den Views USER_JOBS und USER_JOBS_RUNNING.

Um einen Job wiederum zu löschen, sind zwei Prozeduraufrufe notwendig. Zunächst muss mit Hilfe der Prozedur

```
broken(jobnummer, FALSE);
```

der aktive, zu löschende Job aus der Jobliste herausgenommen werden. Dadurch bleibt er jedoch in seiner Definition erhalten. Mittels

```
broken(jobnummer, TRUE);
```

könnte er jederzeit wieder aktiviert werden. Nachdem die Ausführung eines Jobs gestoppt wurde, wird er durch

```
remove(jobnummer);
```

von der Datenbank gelöscht.