```
FUNCTION substr(lob_loc IN BLOB,
                amount  IN INTEGER := 32767,
                offset  IN INTEGER := 1)
RETURN RAW;

FUNCTION substr(lob_loc IN CLOB CHARACTER SET ANY_CS,
                amount  IN INTEGER := 32767,
                offset  IN INTEGER := 1)
RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET;

FUNCTION substr(file_loc IN BFILE,
                amount   IN INTEGER := 32767,
                offset   IN INTEGER := 1)
RETURN RAW;
```

substr() liefert einen Teil eines LOB-Objektes zurück. amount definiert dabei die Anzahl Bytes, die zurückgegeben werden sollen. offset bestimmt, ab welchem Byte gelesen werden soll.

```
PROCEDURE trim(lob_loc IN OUT BLOB,
               newlen  IN     INTEGER);

PROCEDURE trim(lob_loc IN OUT CLOB CHARACTER SET ANY_CS,
               newlen  IN     INTEGER);
```

Über trim() wird die Größe der LOB-Typen verringert. Der Parameter *newlen* definiert dabei die neue Länge der Variablen. Dabei werden die Zeichen bzw. Bytes vom Ende abgeschnitten. Der Wert von *newlen* wird danach auch über getlength() zurückgegeben.

```
PROCEDURE write(lob_loc IN OUT BLOB,
                amount  IN     BINARY_INTEGER,
                offset  IN     INTEGER,
                buffer  IN     RAW);

PROCEDURE write(lob_loc IN OUT CLOB     CHARACTER SET ANY_CS,
                amount  IN     BINARY_INTEGER,
                offset  IN     INTEGER,
                buffer  IN     VARCHAR2 CHARACTER SET lob_loc%CHARSET);
```

Die write()-Funktionen schreiben Daten in einen LOB-Typ. *lob_loc* definiert dabei wieder die Variable, die übrigen Parameter besitzen die schon bekannten Funktionen.

Fortgeschrittenere Datenbank-Techniken

8.1 Allgemeines

Nachdem Sie im Kapitel 3 etwas über die bei der Oracle Installation mitgelieferten Werkzeuge und in den Kapiteln 4 bis 6 die Grundlagen von SQL bzw. PL/SQL erfahren haben, ist es jetzt an der Zeit, weitere fortgeschrittenere Techniken der Datenbankprogrammierung zu erlernen. Abgesehen von wenigen Ausnahmen wurden die bisherigen Beispiele immer nur für eine Ein-Benutzerumgebung ausgelegt. `COMMIT` und `ROLLBACK` wurden dabei so gut wie gar nicht genutzt. Aber gerade bei Datenbankanwendungen mit konkurrierenden Zugriffen auf die gleichen Datenbestände spielen Benutzerverwaltung und Zugriffsmethoden auf die Datenbank eine wichtige Rolle. Stellen Sie sich einmal vor, zwei Anwender eines Datenbanksystems greifen gleichzeitig auf den gleichen Datensatz zu und ändern diesen. Welche Änderung hat nun Gültigkeit? Ein anderes wichtiges Feature einer „echten" Datenbank ist die Möglichkeit, logische Tabellenverbindungen zu verwalten und automatisch zu überwachen, um damit die logische bzw. die referentielle Integrität der Daten sicherzustellen. Neben diesen wichtigen Mechanismen wird in diesem Kapitel noch eine Reihe weiterer nützlicher Funktionen und Operatoren beschrieben.

Oracle verwendet für die Verwaltung konkurrierender Zugriffe, also gleichzeitiger Zugriffe auf gleiche Datenbestände, den Mechanismus des *row locking*, d.h. es besteht die Möglichkeit, *einen* Datensatz gegen Änderung durch Dritte zu sperren. Leider ist dieses Verfahren nicht standardisiert. Auch hier „kocht jeder Datenbankhersteller sein eigenes Süppchen". So unterstützt DB2 und der Microsoft SQL Server beispielsweise das *page locking*, d.h. es wird immer eine ganze Seite bzw. Page gesperrt. Wieviel Datensätze solch eine Page beinhaltet, hängt wiederum davon ab, wieviel Tabellenspalten die Tabelle besitzt und von welchem Typ diese Tabellenspalten sind. Bei Informix-Datenbanken kann für jede Tabelle einzeln festgelegt werden, ob für sie das row oder das page locking gelten soll. Allein an dem Vergleich dieser Systeme sehen Sie also schon, wie kompliziert es ist, einen gemeinsamen Nenner bei der Satzsperre zu finden.

Hinweis:
Soll eine Datenbankanwendung unabhängig von dem eigentlichen Datenbanksystem, also lauffähig auf Oracle, Informix und dem SQL Server oder anderer SQL-Datenbanken sein, dürfen Sie auf keinen Fall einen spezifischen Mechanismus der Datenbank nutzen. Hier kommen Sie nicht umhin, einen eigenen Algorithmus zu implementieren. Denkbar wäre es beispielsweise, einen Datensatz durch das Setzen einer bestimmten Eigenschaft als *in Bearbeitung* zu markieren. Die Anwendung selbst müsste dann vor der Änderung eines Satzes zunächst überprüfen, ob diese Eigenschaft gesetzt ist. Ist das der Fall, darf keine Änderung vorgenommen werden, und es muss solange gewartet werden, bis die Markierung wieder aufgehoben wird, d.h. bis ein anderer Anwender die Bearbeitung abschließt. Als erweiterte Version wäre denkbar, dass das Merkmal *in Bearbeitung* außerdem einen Verweis auf den User erhält, der den Datensatz in Bearbeitung hält.

mit Sql Plus arbeiten

8.2 SELECT FOR UPDATE

Doch zurück zum SELECT FOR UPDATE. Angenommen, Sie möchten einen Satz der Kundentabelle auf Gültigkeit aller Angaben überprüfen. Falls Sie feststellen, dass bestimmte Informationen geändert werden müssen, erfolgt dies über eine zweite SQL-Anweisung. Für den Zeitraum zwischen Selektion der Daten und der Entscheidung, dass beispielsweise der Name geändert werden muss, müssen Sie sicherstellen, dass kein anderer Anwender ebenfalls diesen Datensatz ändert. Sie müssen ihn also für sich *reservieren*. Unter Oracle geschieht dies mit dem Zusatz FOR UPDATE innerhalb der SELECT-Anweisung. Starten Sie jetzt SQL*Plus, und geben Sie folgende Anweisung ein:

```
SELECT * FROM kunden
WHERE  kundennr=100
       FOR UPDATE;
```

Abbildung 8-1 zeigt das Ergebnis dieser Abfrage. Bitte beachten Sie, dass ich vor der SELECT-Anweisung die Spaltenbreite der Tabelle auf 20 Zeichen begrenzt habe.

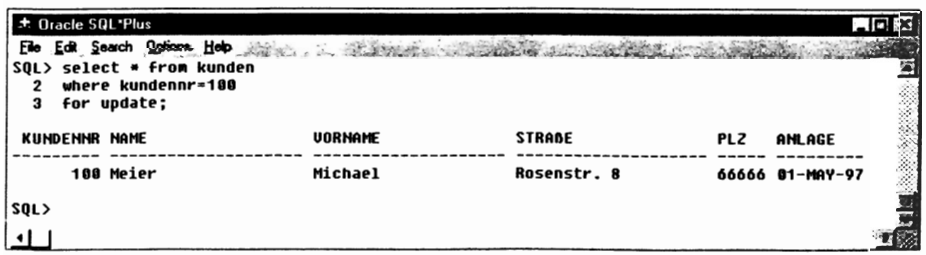

Abbildung 8-1: SELECT FOR UPDATE-*Anweisung*

Zunächst einmal lässt sich bezüglich der Ergebnismenge kein Unterschied feststellen. Durch den Zusatz FOR UPDATE haben Sie aber diesen Datensatz intern für den Änderungszugriff durch andere Benutzer gesperrt. Wir simulieren jetzt den Zugriff eines anderen Benutzers dadurch, dass wir eine zweite SQL*Plus-Sitzung starten und eine Modifizierung dieses Datensatzes versuchen. Die Abbildung 8-2 zeigt die zweite Sitzung mit einer entsprechenden UPDATE-Anweisung.

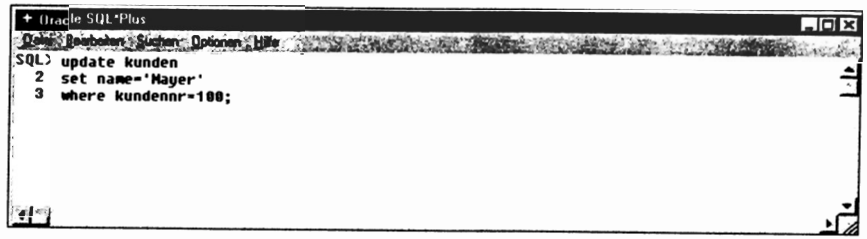

Abbildung 8-2: UPDATE-*Anweisung in SQL*Plus*

Versuchen Sie an dieser Stelle, den Namen des ersten Kunden zu verändern, und bestätigen Sie die obige Anweisung mit RETURN. Sie werden sehen, dass der Cursor von SQL*Plus nicht wieder auf dem Bildschirm erscheint, weil die Anweisung nicht beendet bzw. durchgeführt werden kann. Erst wenn Sie in der ersten SQL*Plus-Sitzung ein COMMIT absetzen, kehrt auch in der zweiten Sitzung der Cursor zurück.

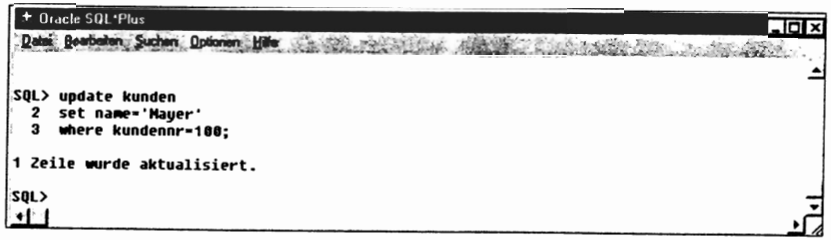

Abbildung 8-3: *Geänderter Name*

Wenn versucht wird, von zwei SQL*Plus-Sitzungen den gleichen Satz zu sperren, erscheint bei dem zweiten Versuch eine Fehlermeldung, wie sie in Abbildung 8-4 zu sehen ist.

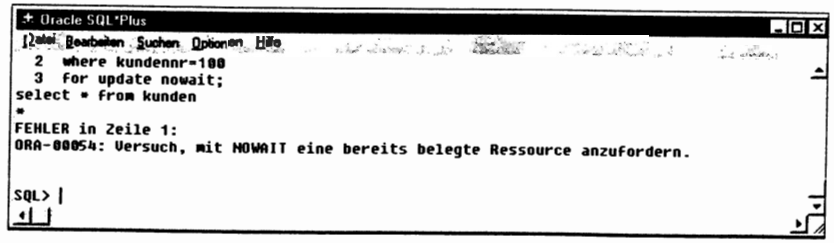

Abbildung 8-4: *Fehlermeldung von SQL*Plus*

Sie können zwar diese Fehlermeldung quittieren, die Änderung wird jedoch nicht abgespeichert. Von keinem anderen Frontend haben Sie jetzt die Möglichkeit, Änderungen auf diesem Datensatz durchzuführen; er ist für den Anwender reserviert, der über SQL*Plus auf ihn zugreift.

Die allgemeine Syntax für die SELECT FOR UPDATE lautet:

```
Select-Anweisung
FOR UPDATE [NOWAIT]
```

Select-Anweisung definiert dabei die schon bekannte SELECT-Anweisung mit all ihren Varianten. Angehangen wird der Zusatz FOR UPDATE. Optional kann außerdem noch das Schlüsselwort NOWAIT angegeben werden. Damit definieren Sie, wie Oracle sich verhalten soll, wenn es auf einen schon gesperrten Satz trifft. NOWAIT definiert dabei, dass Oracle eine Fehlermeldung zurückgibt (Abbildung 8-4). Erfolgt die Angabe von NOWAIT nicht, gibt Oracle keine Fehlermeldung zurück, sondern wartet solange, bis die Sperre aufgehoben wurde. Erst danach liefert die Datenbank direkt das Ergebnis der Selektion zurück.

8.3 Datenbanktuning über Indizes

8.3.1 Allgemeines

In Kapitel 1 wurden schon einmal Indizes erwähnt. Sie erlauben einen wesentlich schnelleren Zugriff auf Tabellendaten. In unseren bisherigen Beispielen mit der Kunden- und Telefontabelle spielten Indizes keine Rolle, weil einfach eine zu geringe Anzahl Daten vorhanden war. Sie dürfen aber nicht vergessen, dass alle bisherigen Such- und Selektions-Anweisungen in den Tabellen sequentiell erfolgten. Am Beispiel der Kundentabelle bedeutet das, dass die Anweisung

```
SELECT * FROM kunden
WHERE  kundennr = 100,
```

bei der Kundennummer 100 beginnt und die gesamte Tabelle bis zum Ende hin durchläuft. Bei sechs Datensätzen merken Sie diesen Zeitverlust nicht, bei 20000 Datensätzen können dabei aber schon einige Rechnertakte vergehen. Die Definition von Indizes erfolgt in der Regel zeitgleich mit der Anwendungsentwicklung bzw. mit dem Datenbankdesign. Man sollte sich dabei sehr genau überlegen, auf welchen Tabellenspalten eine Selektion häufig durchgeführt wird und diese dann in die Indizierung mit einbeziehen. So macht es beispielsweise wenig Sinn, auf das Feld *Vorname* einen Index zu legen; nach dieser Spalte dürfte wohl vergleichsweise wenig gesucht und sortiert werden. Die Kundennummer hingegen ist das Schlüsselkriterium, hierauf sollte auf jeden Fall ein Index gelegt werden. Es empfiehlt sich außerdem, nicht mehr Indizes als unbedingt notwendig anzulegen. Da diese

Indizes bei jeder Änderung mitgepflegt werden müssen, beansprucht dies entsprechend Zeit. Die Pflege von Indizes, die ohnehin niemand benötigt, ist Zeitverschwendung.

Als Faustregel gilt: Indizes sollten für die Tabellenspalten angelegt werden, die häufig im WHERE-Abschnitt einer Selektion erscheinen. Es ist zudem nicht sinnvoll, Indizes auf Tabellenspalten zu legen, deren Wertemenge nur wenige Unterscheidungskriterien zulässt. Beispiel hierfür wäre eine Tabellenspalte, in der das Geschlecht einer Person abgelegt wird. Da in einem solchen Fall lediglich die Werte „w" und „m" Merkmal sind, bringt ein Index auf eine solche Tabellenspalte keinen Performance-Gewinn.

Im Gegensatz zu dBase- oder Paradox-Systemen wird bei SQL-Datenbanken der Hinweis auf das Vorhandensein eines Index nicht direkt in der Struktur der jeweiligen Tabelle eingetragen. Auf einer Oracle-Datenbank erfolgt das Anlegen eines Index durch Eintrag der Indexdefinition in Systemtabellen der Datenbank Das Erzeugen von Indizes ist für die Einsatzfähigkeit einer SQL-Datenbank nicht zwingend notwendig, in vielen Fällen jedoch erforderlich, um eine akzeptable Ablaufgeschwindigkeit und geringe Antwortzeiten zu erlangen.

8.3.2 Index erzeugen

Die allgemeine Syntax zum Erzeugen eines Index lautet wie folgt:

```
CREATE [UNIQUE] INDEX Indexname ON Tabellenname
    (Liste der Tabellenspalten) [ASC/DESC]
```

Indexname gibt dabei einen Namen an, über den der Index identifiziert wird. *Tabellenname* definiert die Tabelle, für die der Index erzeugt werden soll. Die *Liste der Tabellenspalten* markiert, welche Spalten für die Indizierung herangezogen werden sollen. Optional kann der Zusatz UNIQUE verwendet werden. Dadurch wird der Index als *eindeutig* gekennzeichnet. Die *Liste der Tabellenspalten* identifiziert dabei eindeutig jeden einzelnen Satz. ASC definiert eine aufsteigende Sortierung und DESC eine absteigende Sortierung, wobei die aufsteigende Sortierung der Default-Wert ist.

Bezogen auf die Kundentabelle wäre ein eindeutiger Index auf der Spalte *Kundennr* sinnvoll, während ein zweiter Index auf dem Feld *Name* nicht als UNIQUE definiert werden sollte, da nicht ausgeschlossen werden kann, dass die Kundentabelle zwei gleichnamige Kunden beinhaltet. Der Vorteil von UNIQUE-Indizes ist, dass Sie damit automatisch eine Duplizitätskontrolle der Datensätze in der Datenbank implementiert haben, ohne auch nur eine Zeile Programmcode schreiben zu müssen. Die folgende Anweisung zeigt das Erzeugen beider Indizes:

```
SQL> CREATE UNIQUE INDEX nr on kunden (kundennr);

Index created.

SQL> CREATE INDEX name on kunden (name);

Index created.

SQL>
```

Ein Index wird als *einfacher* Index bezeichnet, wenn er sich aus lediglich einer Tabellenspalte zusammensetzt. Man spricht von einem *zusammengesetzten* Index, wenn seine Definition mehrere Tabellenspalten enthält. Das folgende Beispiel zeigt einen solchen zusammengesetzten Index:

```
SQL> CREATE INDEX iname on kunden (name, vorname);

Index created.

SQL> CREATE INDEX name on kunden (name);

Index created.

SQL>
```

Die obige Definition erzeugt einen Index, der nicht zwingend eindeutig sein muss. Wird er mit dem Zusatz `UNIQUE` definiert, muss jede Kombination aus *name* und *vorname* eindeutig sein. Jeweils eine Kombination aus *name* und *vorname* identifiziert also genau einen Datensatz. Einen solchen `UNIQUE`-Index zeigt das folgende Beispiel:

```
SQL> CREATE UNIQUE INDEX iname on kunden (name, vorname);

Index created.

SQL> CREATE INDEX name on kunden (name);

Index created.

SQL>
```

Hinweis:
Wenn eine Tabelle auf einem Objekt-Typ von Oracle 8 bzw. eine Tabellenspalte auf demselbigen basiert, sind einige Einschränkungen zu beachten. Indizes können lediglich auf den skalaren Datentypen definiert werden. Bei Objekten kann also nur auf einzelne Eigenschaften ein Index gelegt werden, nicht auf das Objekt als Ganzes.

Wie schon in Kapitel 5 beschrieben, werden die Index-Definitionen innerhalb einer System-Tabelle von Oracle verwaltet. Über eine entsprechende Selektion der Tabelle `USER_INDEXES` kann man sich genau über die im System vorhandenen Indizes informieren. Abbildung 8-5 zeigt eine tabellarische Darstellung aller Indizes die vom Benutzer *Heitsiek* angelegt wurden.

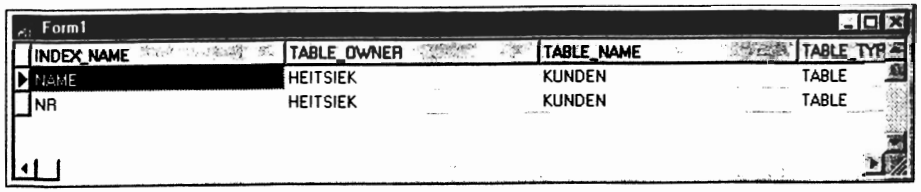

Abbildung 8-5: *Vorhandene Indizes des Systems*

Die Indizierung eines oder mehrerer Felder hat zur Folge, dass innerhalb der Datenbank eine Indextabelle angelegt wird, in der die wichtigen Informationen, also die indizierten Feldinhalte, abgelegt sind. Nach dem Erzeugen der Indizes für die Tabelle `kunden` hat dies auf die eigentliche Tabelle keine Auswirkungen. Abbildung 8-6 zeigt die gesamte Kundentabelle.

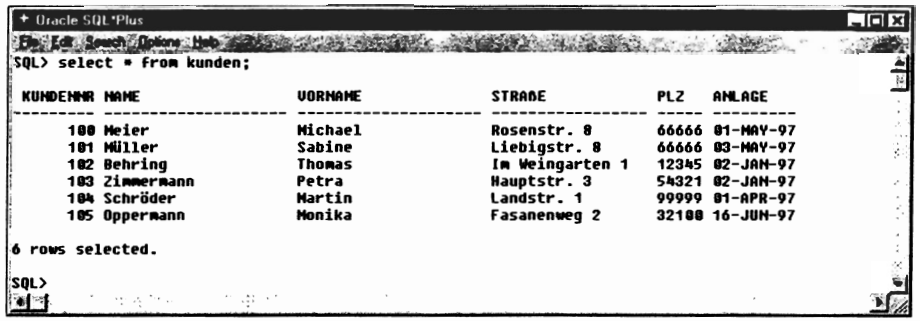

Abbildung 8-6: *Kundentabelle*

Neben dieser eigentlichen Datentabelle gibt es noch weitere Tabellen, in denen die Indexinformationen abgelegt sind. Tabelle 8-1 zeigt den Aufbau der Indextabelle für den Index *Name*.

Satznummer	Name
3	Behring
1	Meier
2	Müller
6	Oppermann
5	Schröder
4	Zimmermann

Tabelle 8-1:
Indextabelle für Name

Da diese Indextabelle alphabetisch sortiert ist, kann hier über bestimmte Mechanismen enorm schnell gesucht werden. Über die Information *Satznummer* kann das DBMS danach den kompletten Satz in der Kundentabelle über einen Zugriff finden. Dadurch reduziert sich die Zahl der Festplattenzugriffe, und die Performance steigt.

Hieran erkennen Sie auch die Nachteile, die eine Vielzahl von Indizes mit sich bringt. Soll beispielsweise ein neuer Kunde mit Namen *Meyer* in die Kundentabelle eingefügt werden, so muss die Indextabelle ebenfalls aktualisiert werden. Da diese nach Namen sortiert ist, muss der neue Kunde in der Mitte eingefügt und alle folgenden um 1 nach unten versetzt werden.

Solche Indextabellen bieten außerdem noch einen weiteren Vorteil. Bei einer gut überlegten Definition von Indizes kann das DBMS sogar auf den Zugriff auf die eigentliche

Datentabelle verzichten. Bei dem in Tabelle 8.1 angelegten Index kann beispielsweise die folgende Abfrage völlig auf die eigentliche Datentabelle verzichten:

```
SQL> SELECT name FROM kunden
  2  WHERE  name='Oppermann';

NAME
----------------------------------------
Oppermann
```

Wenn also die selektierten Informationen einer Anweisung komplett in der Indexdatei enthalten sind, wird nicht auf die eigentliche Datentabelle zurückgegriffen. Diesem Performance-Gewinn steht allerdings ein erhöhter Pflegeaufwand bei INSERT- oder UPDATE-Anweisungen für die zusammengesetzten Indizes entgegen.

Hinweis:
Man spricht bei solchen Vorgängen, bei denen lediglich die Indextabelle und nicht mehr die Datentabelle gelesen werden muss, von einem *key only read*.

Vor dem Erzeugen von Indizes sollten Sie sich als Datenbankadministrator also genau überlegen, welche Felder bzw. welche Kombinationen an Feldern häufig in WHERE-Klauseln erscheinen. Es ist oftmals sinnvoll, hierfür einen zusammengesetzten Index zu formulieren. Bei großen Datenmengen kann man sich bzw. dem DBMS unter Umständen dadurch eine Reihe von (langsamen) Festplattenzugriffen sparen, da evtl. schon alle selektierten Felder in der Indextabelle enthalten sind.

Hinweis:
Die Verwendung der LOWER()- und UPPER()-Funktionen (und weiterer Manipulationen an den Selektionsspalten) setzen die Indizes außer Kraft bzw. werden evtl. vorhanden Indizes nicht herangezogen. Eine Abfrage wie die folgende würde die Tabelle beispielsweise sequentiell durchsuchen:

```
SQL> SELECT name FROM kunden
  2  WHERE  lower(name)='oppermann';

NAME
----------------------------------------
Oppermann
```

Allgemein ist zu sagen, dass die indizierten Spalten in der WHERE-Klausel in "reiner" Form zu formulieren sind. Jegliche Modifikation wie beispielsweise die LOWER()- und UPPER()-Funktionen verhindern den Einsatz von Indizes!

8.3.3 Doppelte Datensätze finden

Unique Indizes können nur auf solchen Spalten definiert werden, die auch wirklich satzidentifizierend wirken. Gerade bei schon gefüllten Tabellen kann es vorkommen, dass die Datenbank sich weigert, einen solchen Index anzulegen. In einem solchen Fall sollte man die entsprechenden Feldinhalte der zu indizierenden Spalten der Tabelle auf evtl. Duplizitäten hin überprüfen. Das folgende Script zeigt ein solches Beispiel:

```
SQL> SELECT empno, ename, job FROM emp;

     EMPNO ENAME      JOB
---------- ---------- ---------
      7369 SMITH      CLERK
      7499 ALLEN      SALESMAN
      7521 WARD       SALESMAN
      7566 JONES      MANAGER
      7654 MARTIN     SALESMAN
      7698 BLAKE      MANAGER
      7782 CLARK      MANAGER
      7788 SCOTT      ANALYST
      7839 KING       PRESIDENT
      7844 TURNER     SALESMAN
      7876 ADAMS      CLERK
      7900 JAMES      CLERK
      7902 FORD       ANALYST
      7934 MILLER     CLERK

14 rows SELECTed.

SQL> CREATE UNIQUE INDEX emp_job ON emp (job);
CREATE UNIQUE INDEX emp_job ON emp (job)
                             *
FEHLER in Zeile 1:
ORA-01452: CREATE UNIQUE INDEX nicht ausführbar; doppelte Schlüssel gefunden

SQL>
```

Das Anlegen eines eindeutigen Index scheitert in diesem Fall, weil die Tabellenspalte *JOB* keine eindeutige Satzidentifizierung liefert. In diesem Fall ist das auch beabsichtigt.

Oftmals besteht die Tätigkeit eines Datenbankadministrator darin, doppelte Datensätze einer Tabelle zu identifizieren, diese zu eliminieren um dann im nächsten Schritt einen UNIQUE INDEX auf dieser Tabellenspalte zu definieren. Als Beispiel soll uns jetzt die Tabelle EMP dienen. Zunächst wird eine temporäre Tabelle erzeugt, die mindestens zwei Spalten beinhaltet, einmal die Tabellenspalte, die die Duplizitäten beinhaltet, zum anderen eine Tabellenspalte, die die Anzahl der einzelnen Ausprägungen beinhaltet. Wichtig beim Erzeugen dieser Tabelle ist die Gruppierungsfunktion; ansonsten wird die COUNT()-

Funktion ihren Dienst verweigern. Mit einer zweiten Abfrage haben Sie jetzt die Möglichkeit, die Häufigkeit der einzelnen Merkmale abzufragen. Das folgende Script zeigt das entsprechende Beispiel:

```
SQL> CREATE TABLE temp as (
  2  SELECT job, count(job) anzahl FROM emp group by job);

Tabelle wurde angelegt.

SQL> SELECT * FROM temp;

JOB           ANZAHL
--------- ---------
ANALYST           2
CLERK             4
MANAGER           3
PRESIDENT         1
SALESMAN          4

SQL>
```

8.3.4 GROUP BY in Verbindung mit HAVING

Dieses Verfahren über eine Zwischentabelle eignet sich besonders dann, wenn man die doppelten Werte noch weiter verarbeiten muss und sie nicht einfach gelöscht werden dürfen. Möchte man sich lediglich einen Überblick über die doppelten Werte innerhalb einer Tabelle verschaffen, kommt die HAVING-Klausel zu Hilfe. Das Schlüsselwort HAVING tritt immer nur im Zusammenhang mit der Gruppierungsfunktion GROUP BY auf. Man definiert hierbei eine Bedingung, der das durch GROUP BY erzeugte Gruppenergebnis genügen muss. Man kann sich die Funktion vorstellen wie eine zusätzliche WHERE-Bedingung für die Gruppe:

```
SQL> SELECT job, count(job) Anzahl
  2  FROM   emp
  3  GROUP  BY job
  4  HAVING count(job)>3;

JOB           ANZAHL
--------- ---------
CLERK             4
SALESMAN          4

SQL>
```


Leider unterliegt diese recht nützliche Funktion jedoch einigen Einschränkungen. So kann HAVING nur mit den folgenden Aggregatfunktionen zusammenarbeiten:

- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

8.4 Constraints

Bisher haben wir zwei Tabellen immer über eine SELECT-Anweisung verknüpft, wobei die Verknüpfung selbst immer auf einem gemeinsamen Datenfeld beruhte. Darüber waren Abfragen möglich wie beispielsweise

„Zeige mir die Telefon- und Faxnummer des Kunden mit der Kundennummer 100 an."

In unserem Beispiel mit der Kunden- und der Telefontabelle erfordert das Neuanlegen eines Kunden das Erzeugen eines Satzes in der Kundentabelle und evtl. mehrerer Sätze in der Telefontabelle. Die Telefontabelle wird also erst im zweiten Schritt beschrieben. Beide Tabellen bilden dabei eine logische Einheit. Es macht wenig Sinn, Sätze in der Telefontabelle anzulegen, ohne dafür einen korrespondierenden Satz in der Kundentabelle zu haben. Genau diese logische Überprüfung wird durch die Definition von Nachschlagetabellen erledigt. Für die Telefontabelle ist die Kundentabelle die Nachschlagetabelle erster Wahl, weil hier zuerst nachgeschaut werden muss, ob ein passender Satz vorhanden ist. Diese Funktionalität wird innerhalb der Oracle-Umgebung als *Constraint* bezeichnet. Über die Definition solcher Constraints kann man sicherstellen, dass die Logik eines Datenbankmodells nicht gefährdet wird. Constraints stellen also die *referentielle Integrität* der Datenbank sicher. Am Beispiel der Kunden- und Telefontabelle werden wir jetzt Constraints definieren.

Um eine referentielle Integrität zwischen zwei Tabellen zu definieren, müssen Sie folgendes tun:

- Für die Nachschlagetabelle (in unserem Beispiel ist das die Kundentabelle) muss ein Primärschlüssel, ein sog. *primary key* definiert werden.
- Für die Tabelle, die auf die Nachschlagetabelle zurückgreift, also die Telefontabelle, muss ein Sekundärschlüssel, ein *foreign key* definiert werden.

Hinweis:
Primärschlüssel sollten immer nur bei numerischen Tabellenfeldern erzeugt werden, niemals auf Textfeldern des Typs CHAR oder VARCHAR. Grund hierfür ist die hohe Performance, mit der numerische Werte miteinander verglichen werden können. Gerade bei Primary Keys, die häufig für Vergleiche herangezogen werden, spürt man diesen Geschwindigkeitsgewinn sehr deutlich. Im übrigen liegen auf Oracles Systemtabellen auch nur auf numerischen Tabellenspalten Primärschlüssel und wer, wenn nicht Oracle selbst, müsste um die Vorteile wissen.

8.4.1 Primärschlüssel erzeugen

Die Definition solcher Schlüssel kann einmal direkt beim Erzeugen der Tabelle oder aber auch nachträglich erfolgen. Aus diesem Grund hat der Navigator beim Erzeugen einer Tabelle in Kapitel 3 nach diesen Schlüsseln gefragt. Es ist natürlich ebenfalls möglich, solche Schlüssel mit Hilfe von SQL-Anweisungen aus SQL*Plus heraus zu generieren. Die Definitionen sollen in unserem Beispiel nachträglich der Tabelle hinzugefügt werden. Geben Sie dazu in SQL*Plus folgende Anweisung ein:

```
ALTER TABLE kunden ADD
   CONSTRAINT kunden_nr PRIMARY KEY (kundennr);
```

Über ALTER TABLE leiten Sie die Anweisung zur Modifizierung der Tabellendefinition ein. In diesem Fall wird aber nicht der Datentyp einer Spalte verändert, es wird eine Definition hinzugefügt. CONSTRAINT leitet dabei den Zusatz ein, *kunden_nr* bezeichnet dabei den Constraint. Wichtig ist, dass hier ein eindeutiger Name über die gesamte Datenbank hinweg vergeben wird. PRIMARY KEY kennzeichnet den entsprechenden Constraint als Primärschlüssel. Als letzte Angabe erfolgt hier die Definition der Felder, die den Primärschlüssel bilden. Dieser kann sich aus einem oder mehreren Datenbankfeldern zusammensetzen. Wichtig ist, dass es sich hier um einen eindeutigen Schlüssel handeln muss. Bei der obigen Definition darf die Kundennummer jeweils nur einmal in der Tabelle vorkommen. Wäre der Schlüssel beispielsweise aus Kundennummer und Nachname zusammengesetzt, so muss die Kombination aus Kundennummer und Nachname eindeutig sein. Über die obige Anweisung fügen Sie kein neues Tabellenfeld hinzu:

```
SQL> DESC kunden;
 Name                           Null?    Type
 ------------------------------ -------- ----
 KUNDENNR                       NOT NULL NUMBER(3)
 NAME                                    VARCHAR2(100)
 VORNAME                                 VARCHAR2(100)
 STRAßE                                  VARCHAR2(100)
 PLZ                                     CHAR(5)
 ANLAGE                                  DATE
SQL>
```


Hinweis:
Wie erwähnt, kann die Definition von Constraints auch gleich beim Anlegen der Tabelle geschehen. In einem solchen Fall wäre lediglich die Zeile

```
CONSTRAINT kunden_nr PRIMARY KEY (kundennr);
```

der CREATE-Anweisung hinzuzufügen. Mit dieser Möglichkeit der Definition von Constraints kennen Sie zwei Arten, eine Duplizitätskontrolle auf ein bzw. mehrere Tabellenfelder zu legen. Die erste Möglichkeit ist das Anlegen eines UNIQUE INDEX auf die entsprechenden Spalten, die zweite ist die Definition eines Primärschlüssels über die Constraint-Bedingung.

8.4.2 Sekundärschlüssel erzeugen

Im nächsten Schritt muss für die Telefontabelle ein Sekundärschlüssel definiert werden, der konform zu dem Primärschlüssel der Nachschlagetabelle ist. Geben Sie dazu in SQL*Plus folgende Anweisung ein:

```
ALTER TABLE telefon add
CONSTRAINT  kunden_nr_tel FOREIGN KEY (kundennr)
REFERENCES  kunden;
```

Die Definition des Sekundärschlüssels erfolgt analog zum Erzeugen des Primärschlüssels. Es ist zu beachten, dass die Definition des Sekundärschlüssels in Art und Anzahl der Tabellenfelder mit der des Primärschlüssels übereinstimmt. Hier wäre also eine zusammengesetzte Definition des Schlüssel beispielsweise aus Kunden- und Telefonnummer nicht möglich. Über REFERENCES geben Sie die Tabelle an, die den korrespondierenden Primärschlüssel beinhaltet.

Damit ist die Verbindung zwischen beiden Tabellen geschaffen. Ab sofort können in die Telefontabelle nur solche Datensätze eingetragen werden, die in der Kundentabelle schon ein Pendant besitzen. Vor dem eigentlichen Einfügen wird also zunächst in der Kundentabelle nachgeschlagen, ob die Kundennummer dort existiert. Ist das nicht der Fall, wird die Anweisung zurückgewiesen. Geben Sie zur Überprüfung folgende Anweisung ein:

```
INSERT INTO telefon (kundennr, fon)
VALUES (106, 72348);
```

Oracle weist die Anweisung mit einer Fehlermeldung zurück, weil sich in der Kundentabelle kein Satz mit der Kundennummer 106 befindet:

```
SQL> INSERT INTO telefon (kundennr, fon)
  2  VALUES (106, 72348);
INSERT INTO telefon (kundennr, fon)
            *
```

```
FEHLER in Zeile 1:
ORA-02291: Verstoß gegen Integritätsregel (HEITSIEK.KUNDEN_NR_TEL). Übergeordn.
Schlüssel nicht gefunden

SQL>
```

Die Überprüfung der Constraints erfolgt aber nicht nur beim Einfügen neuer Sätze. Auch Modifikationen an bestehenden Sätzen werden vor der eigentlichen Ausführung auf ihre Integrität hin überprüft:

```
SQL> UPDATE telefon
  2  SET    kundennr=106
  3  WHERE  kundennr=105;
UPDATE telefon
       *
FEHLER in Zeile 1:
ORA-02291: Verstoß gegen Integritätsregel (HEITSIEK.KUNDEN_NR_TEL). Übergeordn.
Schlüssel nicht gefunden

SQL>
```

Ebenso ist es nicht mehr so einfach möglich, die Kundentabelle zu löschen, da eine Verbindung zu der Telefontabelle besteht. Der Löschversuch wird mit einer Fehlermeldung zurückgewiesen:

```
SQL> DROP TABLE kunden;
DROP TABLE kunden
           *
FEHLER in Zeile 1:
ORA-02449: Eindeutige und Primärschlüssel in Tabelle von Fremdschlüsseln referenziert
```

8.4.3 Primär- und Sekundärschlüssel abschalten oder löschen

Mit der Definition solcher Constraints haben Sie sich als Entwickler aber auch einer Reihe von Restriktionen unterworfen. Man kann jetzt eben nicht mehr so einfach eine Tabelle löschen und vielleicht mit anderen Feldern neu erzeugen. In bestimmten Fällen kann es also sinnvoll sein, die Constraints und damit die Schlüssel auf Tabellen wieder zu löschen bzw. zu inaktivieren. Gerade im Entwicklungsstadium einer Datenbankanwendung und dem dazugehörigen Datenbankmodell wird Ihnen diese Problematik sicherlich noch häufiger begegnen.

Zunächst sollen die Schlüssel inaktiviert werden. Hierbei ist zu beachten, dass zunächst der Sekundärschlüssel inaktiviert werden muss, bevor es an den Primärschlüssel geht:

```
SQL> ALTER TABLE telefon
  2  DISABLE CONSTRAINT kunden_nr_tel;

Tabelle wurde geändert.

SQL>
```

Die Deaktivierung des Primärschlüssels erfolgt analog. Über die DISABLE-Anweisung sind die Constraints immer noch in der Datenbank definiert. Sie werden bei der Ausführung von SQL-Anweisungen jedoch nicht weiter beachtet. Über die ENABLE-Anweisung kann man sie jederzeit wieder aktivieren:

```
SQL> ALTER TABLE telefon
  2  ENABLE CONSTRAINT kunden_nr_tel;

Tabelle wurde geändert.

SQL>
```

Neben der einfachen Deaktivierung besteht außerdem die Möglichkeit, Constraints wieder komplett zu löschen. Dies ist insbesondere dann sinnvoll, wenn man erkannt hat, dass die vorhandenen Constraints nicht das gewünschte Ergebnis bringen. Auch beim Löschen muss zunächst der Sekundärschlüssel entfernt werden. Das Löschen der verschiedenen Schlüssel erfolgt dabei über die ALTER TABLE-Anweisung:

```
SQL> ALTER TABLE telefon
  2  DROP CONSTRAINT kunden_nr_tel;

Tabelle wurde geändert.

SQL>
```

Über den Einsatz von Constraints kann man also wieder ein Stück Intelligenz aus dem Datenbankfrontend in die Datenbank verlagern. Wie bei den Sequenzen, Triggern und Stored Procedures liegt auch hier der Vorteil auf der Hand:

- Performance-Gewinne
- sinkende Netzbelastung
- Verwaltung logische Zusammenhänge innerhalb der Datenbank und nicht im Frontend

8.4.4 Check-Bedingungen

Neben der Möglichkeit, logische Tabellenverknüpfungen über Constraints zu definieren, gibt es noch weitere. Eine besteht darin, Constraints auf eine Tabellenspalte zu definieren. Über eine CHECK-Anweisung kann eine Menge zulässiger Werte für eine Tabellenspalte definiert werden. Das folgende Listing zeigt verschiedene Varianten der CHECK-Bedingung:

```
CREATE TABLE test
(kundennr number(3) CONSTRAINT check_kundennr
    CHECK (kundennr BETWEEN 1 and 999),
 name VARCHAR2(100) CONSTRAINT check_name
    check (name=UPPER(name)),
 vorname VARCHAR2(100) CONSTRAINT check_vorname
    CHECK (vorname IN ('Heidi', 'Peter')),
 straße VARCHAR2(100) CONSTRAINT check_straße
    CHECK (straße=UPPER(straße)),
 PLZ char(5),
 anlage date);
```

Diese Anweisung erzeugt die Tabelle *test* auf der Datenbank. Für die Felder *kundennr*, *name*, *vorname* und *straße* sind Constraints definiert worden. Die Check-Bedingung für das Feld *kundennr* lässt nur Werte zwischen 1 und 999 zu. Die folgende INSERT-Anweisung wird mit einer Fehlermeldung zurückgewiesen:

```
SQL> INSERT INTO test (kundennr) VALUES (0);
INSERT INTO test (kundennr) VALUES (0)
*
FEHLER in Zeile 1:
ORA-02290: Verstoß gegen CHECK-Regel (HEITSIEK.CHECK_KUNDENNR)

SQL>
```

Die Menge der zulässigen Werte ist im Prinzip schon durch die Wahl des Feldtyps bestimmt. Über eine solche Bedingung kann die Menge nochmals eingeschränkt werden. Sehr interessant ist die CHECK-Bedingung für die Felder *name* und *straße*. Über diese Definition werden hier nur Werte zugelassen, die komplett in Großbuchstaben geschrieben sind. Eine INSERT-Anweisung mit Kleinbuchstaben wird hier ebenfalls zurückgewiesen:

```
SQL> INSERT INTO test (name) VALUES ('Meier');
INSERT INTO test (name) VALUES ('Meier')
*
FEHLER in Zeile 1:
ORA-02290: Verstoß gegen CHECK-Regel (HEITSIEK.CHECK_NAME)

SQL>
```


Für das Feld *vorname* ist eine weitere Bedingung definiert worden. Während die Menge der zulässigen Werte für das Feld *kundennummer* durch eine Ober- und Untergrenze eingeschränkt wurde, sind für den *vornamen* explizit gültige Werte angegeben worden. In diese Tabellenspalte können lediglich die Werte *Heidi* oder *Peter* eingefügt werden:

```
SQL> INSERT INTO test (vorname) VALUES ('Laura');
INSERT INTO test (vorname) VALUES ('Laura')
*
FEHLER in Zeile 1:
ORA-02290: Verstoß gegen CHECK-Regel (HEITSIEK.CHECK_VORNAME)

SQL>
```

Hinweis:
Es ist außerdem möglich, mehrere Constraints nebst zugehöriger CHECK-Bedingung auf einer Tabellenspalte zu definieren. Die einzelnen Constraints und die CHECK-Bedingungen sind dabei einfach hintereinander zu schreiben; sie werden nicht durch Komma getrennt. Auch diese Constraints können durch einfache ENABLE- oder DISABLE-Anweisungen aktiviert bzw. deaktiviert werden.

8.5 Self Joins

Nachdem im Kapitel 4 die Inner und Outer Joins erläutert wurden, möchte ich an dieser Stelle auf eine weitere Möglichkeit der Verknüpfung von Tabellen, genauer von ein und derselben Tabelle, eingehen. Die Verknüpfung einer Tabelle mit sich selbst wird auch als *Self Join* bezeichnet. Abbildung 8-7 verdeutlicht die Beziehung einer Tabelle zu sich selbst.

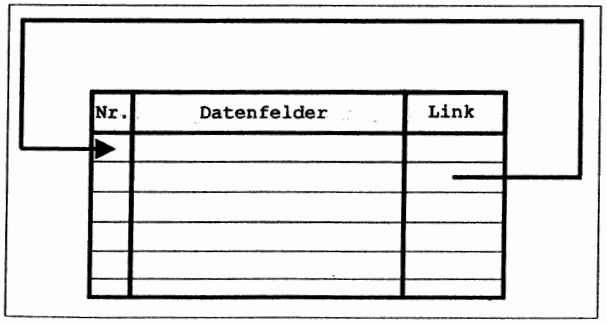

Abbildung 8-7:
Verknüpfung innerhalb einer Tabelle

Ich möchte diese Art der Verknüpfung am Beispiel einer Tabelle erklären, in der verschiedene Arbeitsschritte mit entsprechenden Bezeichnungen abgelegt sind. Tabelle 8-2 listet diese Arbeitsschritte auf.

Nr	Beschreibung	te	tr	Werkstück	Nr_Grob
1	Bohren	1	2	17	0
2	Bohrer holen	0	1	17	1
3	Bohrer einspannen	0	1	17	1
4	Drehen	3	2	19	0
5	Werkstück aufnehmen	0	1	19	4
6	Werkstück einspannen	0	1	19	4
7	Fräsen	10	5	17	0
8	Werkstück einspannen	0	2	17	7
9	CNC-Programm laden	0	3	17	0

Tabelle 8-2: *Arbeitsgangbeschreibungen*

In dieser Tabelle sind einzelne Arbeitsgänge der Metallbearbeitung jeweils für ein Werkstück hinterlegt. Dabei existiert einmal eine Beschreibung des Hauptarbeitsganges (Bohren, Drehen, Fräsen) sowie eine feinere Untergliederung. Zu dem Arbeitsgang *Bohren* gehören beispielsweise die Datensätze 2 und 3. Das folgende Script erzeugt die Tabelle *Liste* auf der Datenbank und füllt sie mit dem Inhalt aus Tabelle 8-2:

```
DROP TABLE liste;

CREATE TABLE liste (
   Nr    NUMBER NOT NULL,
   Beschreibung          VARCHAR2(20),
   te             NUMBER NOT NULL,
   tr             NUMBER NOT NULL,
   Werkstück      NUMBER NOT NULL,
   Nr_Grob        NUMBER NOT NULL);


INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (1, 'Bohren', 1, 2, 17, 0);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (2, 'Bohrer holen', 0, 1, 17, 1);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (3, 'Bohrer einspannen', 0, 1, 17, 1);
```



```
INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (4, 'Drehen', 3, 2, 19, 0);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (5, 'Werkstück aufnehmen', 0, 1, 19, 4);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (6, 'Werkstück einspannen', 0, 1, 19, 4);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (7, 'Fräsen', 10, 5, 17, 0);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (8, 'Werkstück einspannen', 0, 2, 17, 7);

INSERT INTO liste ( Nr, Beschreibung, te, tr, Werkstück, Nr_Grob)
VALUES (9, 'CNC-Programm laden', 0, 3, 17, 7);
```

Im Internet finden Sie dieses Script unter dem Dateinamen LISTE.SQL. Starten Sie dieses Script in SQL*Plus. Abbildung 8-8 zeigt die erzeugte Tabelle.

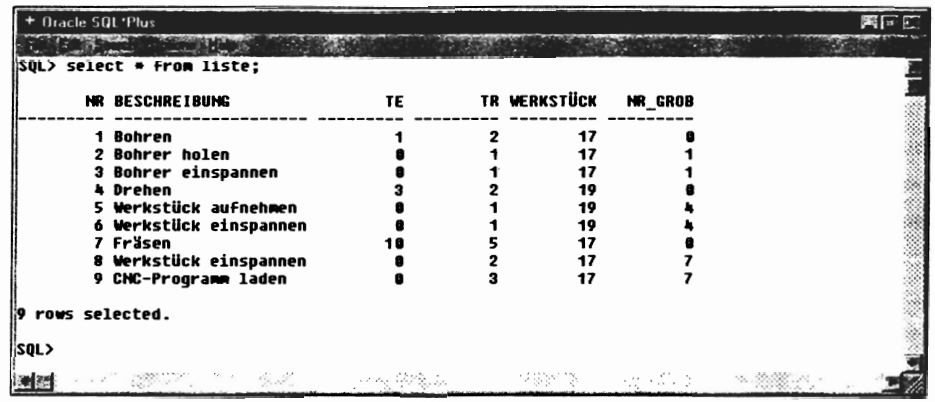

Abbildung 8-8: *Arbeitsgangbeschreibungen*

Die Hauptarbeitsgänge in dieser Tabelle zeichnen sich dadurch aus, dass der Feldinhalt von *NR_GROB* „0" ist. Die folgende Abfrage zeigt beispielsweise alle Hauptarbeitsgänge, die für die Bearbeitung des Werkstücks 17 notwendig sind:

```
SQL> SELECT * FROM liste
  2  WHERE  werkstück = 17
  3  AND    nr_grob = 0;
```

```
        NR BESCHREIBUNG    TE         TR         WERKSTÜCK NR_GROB
---------- --------------- ---------- ---------- --------- --------
         1 Bohren          1          2          17        0
         7 Fräsen          10         5          17        0

SQL>
```

Es stellt sich jetzt die Frage, wie man eine Abfrage formuliert, die zu einem Hauptarbeitsgang die zugehörigen Arbeitsschritte auflistet. Hier hilft uns das Feld *NR_GROB* weiter. Innerhalb der Tabelle zeigt das Feld *NR_GROB* auf das Feld *NR* des Hauptarbeitsgangs. Hierüber kann also eine Verknüpfung innerhalb einer Tabelle erfolgen. Dies machen wir uns zu Nutze und formulieren die folgende Abfrage:

```
SELECT beschreibung, tr FROM liste
WHERE  nr = 1
AND    nr=nr_grob;
```

Wenn Sie diese Abfrage so absetzen, wird Oracle keinen Datensatz zurückliefern. Das liegt am letzten Teil der Abfragebedingung. Innerhalb der Tabelle gibt es keinen Datensatz, für den die Bedingung

```
nr = nr_grob
```

gleichzeitig gilt. Die Problematik liegt darin, dass Oracle die Tabelle Zeile für Zeile durchgeht und die entsprechenden Bedingungen prüft. Die obige SELECT-Abfrage funktioniert also nicht, weil die Tabelle noch nicht mit sich selbst verknüpft ist. Logisch richtig wäre es, wenn die Datenbank die Tabelle Satz für Satz durchgeht und dabei die Bedingungen mit allen anderen Sätzen der Tabelle vergleicht. Hierzu bedienen wir uns wieder der Aliase, wie wir sie in Kapitel 4 kennengelernt haben. Die folgende Abfrage zeigt die korrekte Selektionsbedingung:

```
SELECT l1.nr, l1.beschreibung, l2.beschreibung, l2.tr
FROM   liste l1,
       liste l2
WHERE  l1.nr=1
AND    l2.nr_grob = l1.nr;
```

Hier wird die Tabelle *liste* über zwei verschiedene Aliase angesprochen, im Prinzip analog zu der Verwendung von Aliasen bei mehreren Tabellen. Abbildung 8-9 zeigt das Ergebnis dieser Abfrage.

Vielleicht fragen Sie sich jetzt, weshalb für die Hauptarbeitsgänge und deren Arbeitsschritte nicht zwei einzelne Tabellen erzeugt werden, die über ein gemeinsames Feld verknüpft werden. Bezogen auf die Arbeitsgänge und -schritte kann es passieren, dass man bei bestimmten Abfragen zwischen Arbeitsgängen und -schritten unterscheiden möchte, bei anderen Abfragen aber keine Unterscheidung erfolgen soll. Im letzteren Fall wäre eine Abfrage über zwei oder mehr Tabellen wesentlich umständlicher zu handhaben.

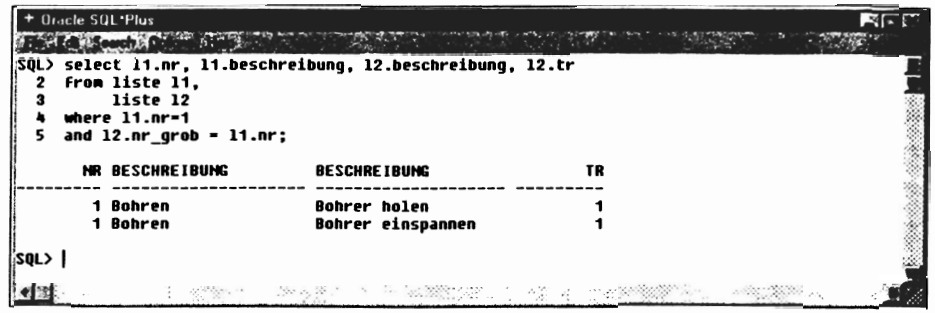

Abbildung 8-9: *Self Join*

8.6 UNION, INTERSECT und MINUS-Operatoren

Die Operatoren UNION, INTERSECT und MINUS dienen in SQL dazu, Ergebnismengen aus zwei SELECT-Statements miteinander zu verknüpfen bzw. zu vergleichen. Als Ergebnis dieses Vergleichs wird *eine* Ergebnismenge zurückgeliefert. Der Vergleich der Ergebnismengen zweier SELECT-Anweisungen setzt natürlich voraus, dass diese überhaupt miteinander zu vergleichen sind. Dies ist dann der Fall, wenn sich die Ergebnismengen in Anzahl und Typ der Ergebnisspalten gleichen. Die allgemeine Syntax für alle drei Operatoren lautet:

SELECT-Statement_1
UNION *oder* INTERSECT *oder MINUS*
SELECT-Statement_2

SELECT-Statement_1 und *SELECT-Statement_2* stellen dabei zwei gültige SELECT-Anweisungen dar. Beide Anweisungen werden mit einem der drei Operatoren verknüpft.

Anhand der Kunden- und der Lieferantentabelle soll die Funktionsweise der drei Operatoren erläutert werden. Dazu sind jedoch einige Vorbereitungen notwendig. Zunächst einmal muss eine Lieferantentabelle erzeugt werden, sofern sie nicht schon vorhanden ist. Hier können Sie sich des AS-SELECT-Konstruktes bedienen, welches in Kapitel 4 beschrieben wurde. Die Lieferantentabelle sollte genau die Datensätze umfassen, die in Abbildung 8-10 abgebildet sind. Allein durch die Anwendung von AS-SELECT werden Sie die benötigten Datensätze allerdings nicht erhalten. Es ist außerdem notwendig, die Datensätze mit den Kundennummern 108 und 109 manuell nachzupflegen. Weiterhin müssen Sie noch die Sätze 100, 102 und 103 aus der Lieferantentabelle löschen.

```
SQL> select * from lieferanten;

 KUNDENNR NAME                 VORNAME              STRAßE               PLZ   ANLAGE
--------- -------------------- -------------------- -------------------- ----- ---------
                                                                              28-JUN-97
      109 Holzmann
      101 Müller               Sabine               Liebigstr. 8         66666 03-MAY-97
      104 Schröder             Martin               Landstr. 1           99999 01-APR-97
      105 Oppermann            Monika               Fasanenweg 2         32100 16-JUN-97
      108 Schmidt                                                             28-JUN-97

SQL>
```

Abbildung 8-10: *Lieferantentabelle*

Abbildung 8-11 zeigt noch einmal die Kundentabelle.

```
SQL> select * from kunden;

 KUNDENNR NAME                 VORNAME              STRAßE               PLZ   ANLAGE
--------- -------------------- -------------------- -------------------- ----- ---------
      100 Meier                Michael              Rosenstr. 8          66666 01-MAY-97
      101 Müller               Sabine               Liebigstr. 8         66666 03-MAY-97
      102 Behring              Thomas               Im Weingarten 1      12345 02-JAN-97
      103 Zimmermann           Petra                Hauptstr. 3          54321 02-JAN-97
      104 Schröder             Martin               Landstr. 1           99999 01-APR-97
      105 Oppermann            Monika               Fasanenweg 2         32100 16-JUN-97

6 rows selected.

SQL>
```

Abbildung 8-11: *Kundentabelle*

Die Kunden- und die Lieferantentabelle ähneln sich sehr. Die Abbildungen 8-10 und 8-11 zeigen, dass sich in der Kundentabelle Datensätze befinden, die nicht gleichzeitig in der Lieferantentabelle enthalten sind und umgekehrt. Gleichzeitig gibt es aber auch gemeinsame Datensätze. Abbildung 8-12 verdeutlicht die Beziehung dieser Datenmengen zueinander. Die Zahlen in der Abbildung geben dabei die Kundennummer bzw. die Lieferantennummer an. Mit Hilfe der drei Mengenoperatoren kann jetzt auf einfache Weise ein Vergleich durchgeführt werden.

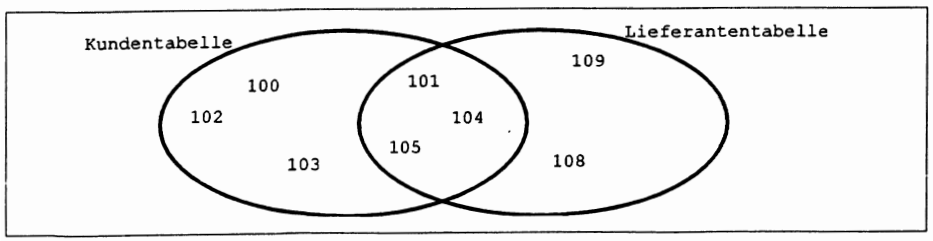

Abbildung 8-12: *Mengendarstellung der Datensätze*

Hinweis:
Die im folgenden beschriebenen Operatoren können auch dazu verwendet werden, Views zu definieren. So ist beispielsweise folgende Anweisung durchaus zulässig:

```
CREATE view unionview as (

SELECT kundennr, name FROM kunden
union
SELECT kundennr, name FROM lieferanten)
```

Außerdem ist es möglich, mehr als zwei Selektionen über die folgenden Operatoren zu verknüpfen.

8.6.1 Der UNION-Operator

Der `UNION`-Operator vereint die Ergebnismengen zweier `SELECT`-Anweisungen derartig, dass er alle Datensätze sowohl aus der einen Selektion als auch aus der zweiten auflistet. Doppelte Datensätze werden dabei unterdrückt. Er erzeugt also eine *Vereinigungsmenge* aus beiden Ergebnismengen. Folgende Anweisung zeigt eine solche `UNION`-Operation auf die Kunden- und die Lieferantentabelle:

```
SQL> SELECT kundennr, name
  2  FROM   kunden
  3  UNION
  4  SELECT kundennr, name
  5  FROM   lieferanten;

 KUNDENNR NAME
--------- --------------------
      100 Meier
      101 Müller
      102 Behring
      103 Zimmermann
      104 Schröder
      105 Oppermann
      108 Schmidt
      109 Holzmann

8 rows SELECTed.

SQL>
```

Abbildung 8-13 verdeutlicht die Mengenbildung noch einmal.

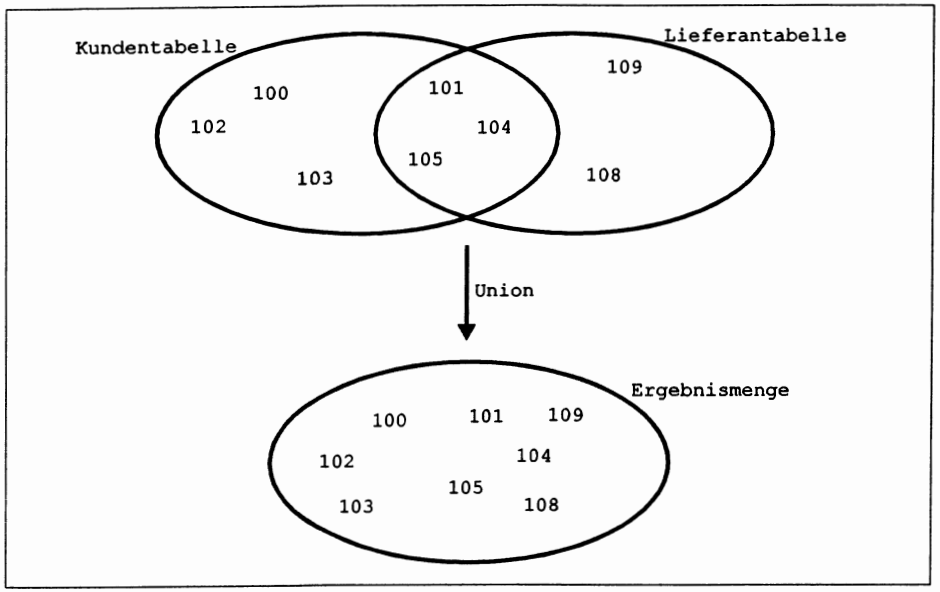

Abbildung 8-13: *UNION-Operation*

Hinweis:
Oracle kennt neben dem einfachen UNION-Operator, den Sie in dieser Form auch auf anderen Datenbanken finden, noch zusätzlich den UNION ALL-Operator. Im Gegensatz zum einfachen UNION-Operator werden bei UNION ALL keine Duplikate unterdrückt. Wenn sich also ein Satz in beiden Tabellen befindet, erscheint er auch zweimal in der Gesamtergebnismenge.

8.6.2 Der INTERSECT-Operator

Im Gegensatz zum UNION-Operator liefert der INTERSECT-Operator die Schnittmenge beider Ergebnismengen. Man erhält also nur die Datensätze, die sich in beiden Ergebnismengen befinden:

```
SQL> SELECT     kundennr, name
  2  FROM       kunden
  3  INTERSECT
  4  SELECT     kundennr, name
  5  FROM       lieferanten;

 KUNDENNR NAME
--------- --------------------
       101 Müller
       104 Schröder
       105 Oppermann

SQL>
```

Abbildung 8-14 zeigt die Schnittmenge beider Ergebnismengen, erzeugt durch die INTERSECT-Operation.

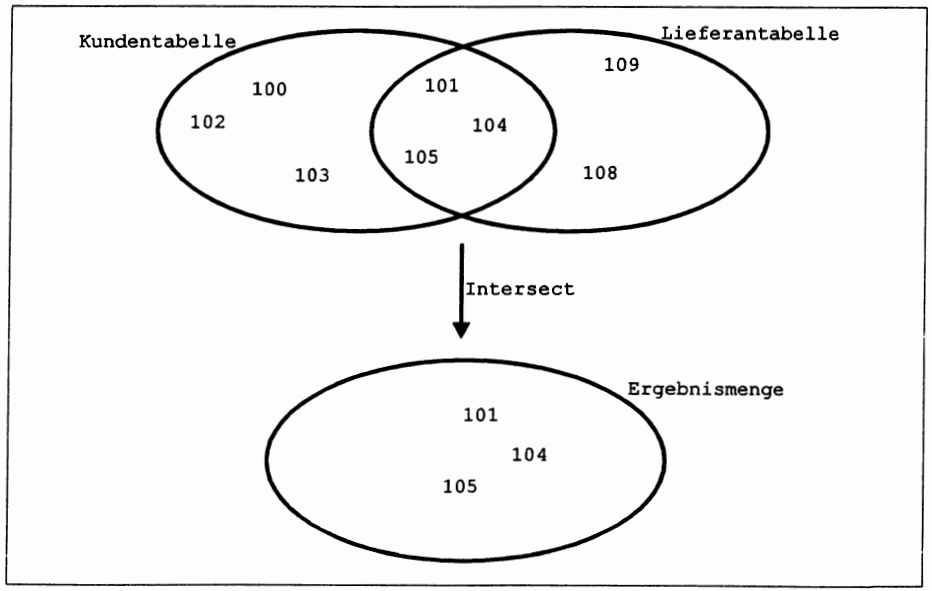

Abbildung 8-14: *INTERSECT-Operation*

8.6.3 Der MINUS-Operator

Der MINUS-Operator subtrahiert zwei Ergebnismengen voneinander.

```
SQL> SELECT kundennr, name
  2  FROM   kunden
  3  MINUS
  4  SELECT kundennr, name
  5  FROM   lieferanten;

 KUNDENNR NAME
--------- --------------------
      100 Meier
      102 Behring
      103 Zimmermann

SQL>
```

Bei diesem Operator ist natürlich darauf zu achten, welche Ergebnismenge von welcher „subtrahiert" wird. Abbildung 8-15 zeigt die Mengen der MINUS-Operation, wenn das Ergebnis der Lieferantenselektion vom Ergebnis der Kundenselektion subtrahiert wird. Abbildung 8-16 zeigt genau den umgekehrten Fall. Hier wird das Ergebnis der Kundenselektion vom Ergebnis der Lieferantenselektion subtrahiert.

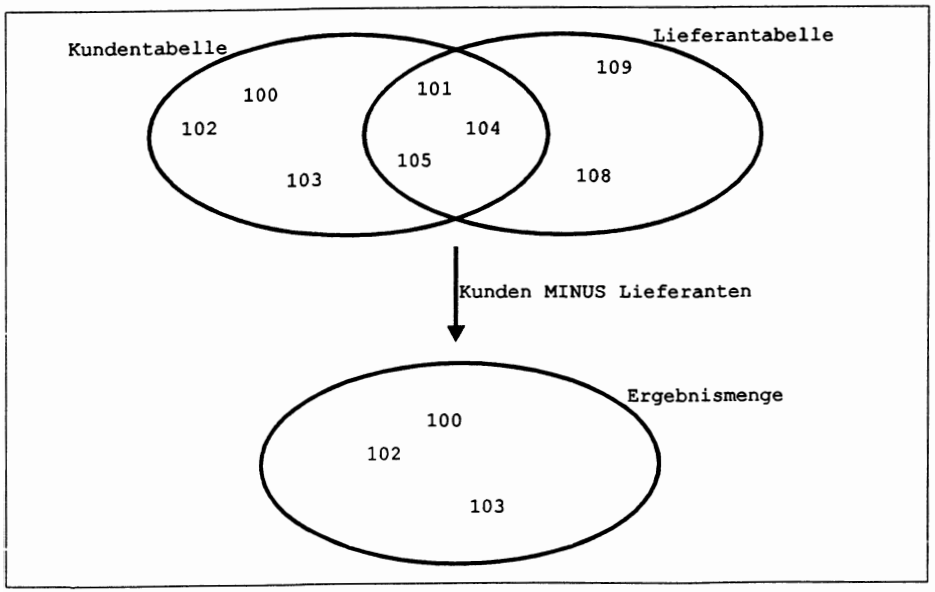

Abbildung 8-15: *Kundentabelle MINUS-Lieferantentabelle*

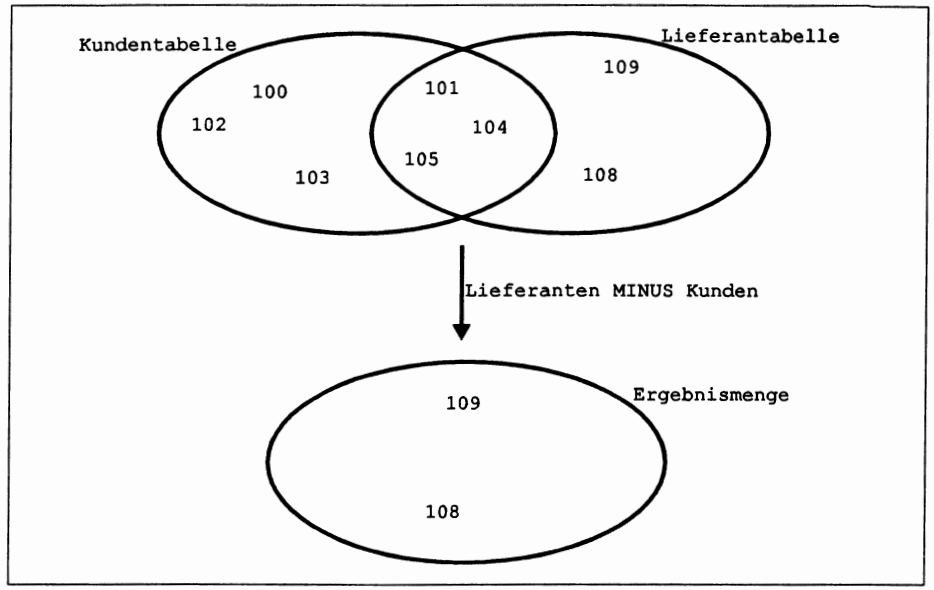

Abbildung 8-16: *Lieferantentabelle MINUS Kundentabelle*

Die folgende Anweisung selektiert zunächst die Daten der Lieferantentabelle, bevor in der zweiten Anweisung die Kundentabelle selektiert wird:

```
SQL> SELECT kundennr, name
  2  FROM   lieferanten
  3  MINUS
  4  SELECT kundennr, name
  5  FROM   kunden;

 KUNDENNR NAME
--------- --------------------
      108 Schmidt
      109 Holzmann

SQL>
```

Diese Verknüpfungsoperatoren können beispielsweise dazu verwendet, verschiedene Datenbanken miteinander zu vergleichen. Dazu ist zunächst ein Datenbank-Link auf die zweite Instanz zu erzeugen, so dass man aus SQL*Plus heraus Zugriff darauf hat. Die folgen-

den Anweisungen vergleichen die für eine Instanz vorhandenen Tabellen und geben eine Differenzliste aus:

```
SELECT table_name FROM ALL_TABLES
MINUS
SELECT table_name FROM ALL_TABLES@Datenbank-Link;

SELECT table_name FROM ALL_TABLES@Datenbank-Link
MINUS
SELECT table_name FROM ALL_TABLES;
```

8.7 Unterabfragen (SUB-SELECT)

8.7.1 Allgemeines

Als Unterabfrage bezeichnet man eine SELECT-Anweisung, die innerhalb einer SELECT-Anweisung ausgeführt wird. Die Ergebnismenge der Unterabfrage wird dann als Vergleichskriterium für die eigentliche Abfrage herangezogen werden. Hierbei ist es zulässig, dass die Unterabfrage genau einen oder mehrere oder keinen Satz zurückliefert. Eine entsprechende Behandlung der unterschiedlichen Ergebnismengen wird über verschiedene Schlüsselwörter eingeleitet, die in den folgenden Abschnitten erläutert werden. Solche Sub-SELECTs werden außerdem noch in *abhängige* und *unabhängige* Unterabfragen unterschieden. Unabhängige Unterabfragen sind – wie der Name schon sagt – voneinander unabhängig, d.h. die Haupt- und die Unterabfrage hängen datentechnisch nicht miteinander zusammen. Anders ist es hingegen mit den abhängigen Unterabfragen. Sie enthält mindestens ein Feld aus einer Tabelle der Hauptabfrage. Das Ergebnis der Unterabfrage ist also abhängig vom aktuellen Satz der Hauptabfrage.

Hinweis:
Es gilt zu beachten, dass die Performance bei der Abarbeitung von verschachtelten Abfragen oftmals schlecht ist. Letztendlich können aber (fast) alle Unterabfragen durch eine intelligente Verknüpfung der verwendeten Tabellen ersetzt werden, wodurch man eine erhebliche Performancesteigerung erreichen kann.

Die allgemeine Syntax solcher Unterabfragen lautet:

```
SELECT Spaltennamen
FROM Haupttabelle
WHERE vergleichspalte VERGLEICHSOPERATOR (
        SELECT vergleichsspalte
        FROM Untertabelle
        [WHERE bedingung])
```

Als *VERGLEICHSOPERATOR* sind die bekannten Vergleichsoperatoren erlaubt.

8.7.2 ANY-Operator

Zunächst soll eine Abfrage realisiert werden, die die Namen aller Kunden aus der Kundentabelle auflistet, die ebenfalls in der Telefontabelle einen Eintrag besitzen. Zunächst sollte folgende Anweisung das Ergebnis liefern; allerdings quittiert Oracle die Anweisung lediglich mit einer Fehlermeldung:

```
SQL> SELECT kundennr, name
  2  FROM   kunden
  3  WHERE  kundennr = (SELECT kundennr FROM telefon);
FEHLER:
ORA-01427: Unterabfrage für eine Zeile liefert mehr als eine Zeile
```

Das Problem bei dieser Art der Formulierung besteht darin, dass diese Unterabfrage mehr als einen Satz zurückliefert. In einem solchen Fall (Anzahl Sätze der Ergebnismenge > 1) muss mit zusätzlichen Operatoren gearbeitet werden!

Die folgende Anweisung liefert das korrekte Ergebnis:

```
SQL>  SELECT kundennr, name
  2   FROM   kunden
  3   WHERE  kundennr = ANY (SELECT kundennr FROM telefon);

 KUNDENNR NAME
--------- ------------------------------
      100 Meier
      101 Müller
      102 Behring
      105 Oppermann

SQL>
```

Wie geht Oracle bei der Bearbeitung eines solchen Statements vor? Zunächst arbeitet der SQL-Parser die innere SELECT-Anweisung ab. Als Ergebnis erhält er eine einspaltige Tabelle mit allen Kundennummern aus der Telefontabelle. Wiederholungen werden in diesem Fall nicht ausgeschlossen. Danach führt er die äußere Anweisung aus und vergleicht dabei den aktuellen Satz der Hauptabfrage mit der Ergebnismenge der Unterabfrage. Findet er **mindestens eine** (ANY) Übereinstimmung, ist die WHERE-Bedingung erfüllt, und die selektierten Spalten werden zur Anzeige gebracht.

Hinweis:
Es ist durchaus erlaubt, Unterabfragen über mehr als eine Ebene zu verschachteln. Bei komplizierteren Abfragen sollte man allerdings aus Performancegründen den Weg über eine oder mehrere Zwischentabellen wählen.

8.7.3 ALL-Operator

Die gleiche Abfrage soll jetzt einmal mit der ALL-Anweisung ausgeführt werden:

```
SQL>  SELECT kundennr, name
  2   FROM   kunden
  3   WHERE  kundennr = ALL (SELECT kundennr FROM telefon);

Es wurden keine Zeilen ausgewählt
```

Auch in diesem Fall wird zunächst die innere Selektion durchgeführt. Bei dem Vergleich zwischen Haupt- und der Unterabfrage muss aber **jeder** jeder Ergebnissatz die WHERE-Bedingung erfüllen. Bei der vorliegenden Abfrage ist das natürlich nicht möglich, denn in der Ergebnismenge befinden sich zur gleichen Zeit verschiedene Kundennummern.

8.7.4 IN-Operator

Der IN-Operator wird verwendet, um zu überprüfen, ob in einer Unterergebnismenge Werte enthalten sind, die die Bedingung der Hauptabfrage erfüllen. Er besitzt damit die gleiche Funktion wie = **ANY**:

```
SQL> SELECT kundennr, name
  2  FROM   kunden
  3  WHERE  kundennr IN (SELECT kundennr FROM telefon);

 KUNDENNR NAME
--------- ------------------------------
      100 Meier
      101 Müller
      102 Behring
      105 Oppermann

SQL>
```


Allerdings ist es (im Vergleich zu ANY) mit dem IN-Operator ein wenig einfacher, die Komplementärmenge einer Selektion zu bilden. Hierzu kennt IN die Negation NOT IN:

```
SQL>  SELECT kundennr, name
  2   FROM   kunden
  3   WHERE  kundennr NOT IN (SELECT kundennr FROM telefon);

 KUNDENNR NAME
--------- ------------------------------
      103 Zimmermann
      104 Schröder

SQL>
```

Hier werden also all die Sätze zurückgeliefert, die in der Telefontabelle keinen Eintrag besitzen.

8.7.5 EXISTS-Operator

Der EXISTS-Operator ist ein Boolescher Operator, der als Ergebnis TRUE oder FALSE zurückliefert. TRUE wird geliefert, wenn die Ergebnismenge der Unterabfrage mindestens einen Satz zurückliefert. Entsprechend erhält man FALSE, wenn als Ergebnis die leere Menge geliefert wird. Im Gegensatz zu den Operatoren ANY, ALL und IN kann der EXISTS-Operator lediglich in abhängigen Unterabfragen verwendet werden, da innerhalb der Unterabfrage auf den aktuellen Satz der Hauptabfrage zurückgegriffen wird:

```
SQL> SELECT name, vorname FROM kunden k
  2  WHERE  EXISTS (
  3  SELECT * FROM telefon t
  4  WHERE  k.kundennr = t.kundennr);

NAME                           VORNAME
------------------------------ ----------
Müller                         Sabine
Meier                          Michael
Behring                        Thomas
Oppermann                      Monika

SQL>
```

Auch hier kann über NOT die Abfrage negiert werden:

```
SQL> select kundennr, name, vorname from kunden k
  2  where  not exists (
  3  select * from telefon t
  4  where  k.kundennr=t.kundennr);

 KUNDENNR NAME                           VORNAME
--------- ------------------------------ ----------
      103 Zimmermann                     Petra
      104 Schröder                       Martin

SQL>
```

Hinweis:
Die einfache EXISTS-Abfrage funktioniert unter Oracle auch, wenn in der Hauptabfrage die zu vergleichende Spalte nicht selektiert wurde. Versucht man gleiches mit NOT EXISTS verweigert Oracle seinen Dienst. Hier hilft nur, die entsprechende Spalte (in diesem Fall KUNDENNR) mit zu selektieren.

Anhand der Beispiele zu den Operatoren IN und EXISTS kann man erkennen, dass durchaus gleiche Aufgaben mit beiden Operatoren gelöst werden können. Gerade im Hinblick auf große Datenmengen gibt es aber je nach Aufgabe erhebliche Performance-Unterschiede bei der Bearbeitung solcher Abfragen. Zur Entscheidungsfindung, welcher Operator für welchen Zweck der bessere ist, muss man die Arbeitsweise des Oracle-Parsers verstehen. Doch kommen wir zurück zum IN-Operator:

```
SELECT kundennr, name
FROM   kunden
WHERE  kundennr IN (SELECT kundennr FROM telefon);
```

Hier wird zunächst die Unterabfrage ausgeführt, bevor die Hauptfrage zur Bearbeitung ansteht. Erst mit der vorhandenen Ergebnismenge der Unterabfrage kann die Hauptabfrage formuliert werden. Die Anzahl der Selektionen der Hauptabfrage entspricht dabei der Anzahl Datensätze der Ergebnismenge von der Unterabfrage.

Im Vergleich dazu folgt nun der EXISTS-Operator:

```
SELECT name, vorname FROM kunden k
WHERE  EXISTS (
SELECT * FROM telefon t
WHERE  k.kundennr = t.kundennr);
```

Hier wird zunächst die Hauptabfrage ausgeführt, die im ersten Schritt ein Vorergebnis liefert. Dazu wird zunächst die gesamte Tabelle eingelesen und ein *Full Table Scan* ausgeführt. Erst danach wird die Unterabfrage ausgeführt. Die Anzahl der Selektionen der Unterabfrage entspricht dabei allerdings nicht immer der Anzahl an Datensätzen der Hauptabfrage. Sobald die Datenbank einen Satz gefunden hat, bricht sie das weitere Scannen der Ergebnistabelle der Unterabfrage ab und liefert TRUE zurück. Nur für den Fall, dass kein Satz gefunden werden kann, wird auf der Ergebnistabelle der Unterabfrage ein Full Table

Scan ausgeführt, bevor FALSE als Ergebnis geliefert wird. Daraus ergibt sich folgende Faustregel für die Verwendung der Operatoren:

→ IN sollte verwendet werden, wenn die Unterabfrage *wenig* Datensätze, die Hauptabfrage hingegen *viele* liefern würde
→ EXISTS sollte verwendet werden, wenn die Hauptabfrage *wenig* Sätze, die Unterabfrage jedoch vergleichsweise viele Sätze zurückliefert.

Hinweis:
Teilweise liefert die Verwendung von IN bzw. EXISTS-Operatoren bessere Antwortzeiten, als entsprechende Formulierungen der Abfragen als OUTER JOINS. Dieses Verhalten ist jedoch jeweils im Einzelfall zu prüfen.

8.7.6 SUB-SELECTs im FROM-Abschnitt

Bislang wurden die Unterabfragen immer dazu verwendet, die WHERE-Klausel der Hauptabfrage genauer zu spezifizieren bzw. zu definieren. Ein Feature von Oracle, mit dem sich die Datenbankengine von denen anderer Hersteller abhebt, ist die Möglichkeit, Sub-Selects im FROM-Abschnitt einzubetten. Dadurch hat der Entwickler oder DBA die Möglichkeit, die Quelle, an die eine Anfrage gestellt wird, innerhalb der Anweisung bzw. des Scriptes zu wechseln. Gerade im Hinblick auf Script-Verarbeitungen und -Generierung kann diese Möglichkeit sehr hilfreich sein.

Die folgenden Zeilen zeigen die Verwendung dieser Möglichkeit. Es wird, wie auch schon in Kapitel 3 beschrieben, ein Script generiert, mit dessen Hilfe man die Anzahl Sätze der User-Tabellen herausfinden kann. Mit Hilfe eines ähnlichen Scriptes könnten beispielsweise der Füllungsgrad der Tabellen oder andere Merkmale der Benutzertabellen abgefragt werden:

```
SQL> SELECT 'SELECT count(*) FROM '||table_name||';' FROM
  2   (SELECT table_name FROM sys.user_tables);

'SELECTCOUNT(*)FROM'||TABLE_NAME||';'
-----------------------------------------------------
select count(*) from ANIMALS;
select count(*) from HISTORIE;
select count(*) from KUNDEN;
select count(*) from LIEFERANTEN;
select count(*) from STATIST;
select count(*) from TELEFON;
select count(*) from WETTER;

7 Zeilen ausgewählt.

SQL>
```

Seite 767