Oracle SQL

Das umfassende Handbuch

Kapitel 10
Datenmanipulation

Nachdem wir ausführlich den lesenden Zugriff auf Daten in der Datenbank besprochen haben, werden wir uns nun um die Manipulation von Daten kümmern. Hierzu gehört das Einfügen, Ändern und Löschen von Daten ebenso wie eine Diskussion zum Begriff der Transaktion, die eine Datenbank von einem Dateisystem unterscheidet.

So viele Seiten und noch keine Daten geändert ... Man könnte ob der Aufgabe, die vor einem liegt, verzweifeln. Doch wäre dies fehl am Platz, denn was ich bereits weiß (und Sie bald auch): Die Datenmanipulation ist im Vergleich zum Lesen von Daten wirklich einfach, die Optionen überschaubar und vieles aus der Kenntnis der `where`-Klausel wiederverwendbar. Sie werden einige alte Bekannte wiedertreffen, etwa Unterabfragen, und im Kontext der Datenmanipulation einsetzen. Das macht diesen Bereich von SQL leicht zu lernen. Einige Grundlagen müssen Sie sich im Vorfeld allerdings schon noch erarbeiten, zumal in das Gebiet der Datenmanipulation der Begriff der Datenbanktransaktion fällt, den Sie gut verstanden haben sollten, denn er ist Ihr Sicherheitsnetz für alle Arbeiten an der Datenbank.

Grundsätzlich stehen uns zur Manipulation von Daten drei, wenn Sie mögen, vier Befehle zur Verfügung: Mit der `insert`-Anweisung werden Daten in die Datenbank eingefügt, die `update`-Anweisung ändert bestehende Daten, und die `delete`-Anweisung löscht Daten aus der Datenbank. Eine Kombination aller drei Anweisungen können Sie ebenfalls verwenden, die Anweisung hierzu heißt dann `merge`. Zu all diesen Anweisungen habe ich Ihnen in diesem Kapitel separate Abschnitte zur Verfügung gestellt, doch ist bereits jetzt klar, dass für eine Manipulation von Daten in der Datenbank mehrere Befehle ausgeführt werden müssen, um ein bestimmtes Ziel zu erreichen: So könnte es sein, dass eine Änderung auf einem bestehenden Datensatz mit der Löschung eines weiteren und dem Einfügen eines dritten Datensatzes kombiniert werden muss. Diese drei Anweisungen werden durch die Transaktion als Einheit zusammengefasst. So weit vorab, sehen wir uns nun die einzelnen Anweisungen detaillierter an.

10.1 Die INSERT-Anweisung

Wir starten mit dem Einfügen neuer Daten in die Datenbank. Grundsätzlich stehen hierfür zwei Varianten zur Verfügung: zum einen das Einfügen von Daten direkt in der SQL-Anweisung, wobei mit dieser Methode jeweils eine Zeile pro Anweisung erzeugt wird, zum anderen das Kopieren von Daten aus einer Tabelle oder Abfrage in eine andere Tabelle. Bei dieser zweiten Methode können sehr viele Zeilen mit einer Anweisung erzeugt werden. Da die Datenbank besser arbeitet, wenn nicht zeilenweise, sondern mengenorientiert gearbeitet wird, ist die zweite Methode beim Einfügen großer Datenmengen vorzuziehen, falls dies möglich ist.

10.1.1 Allgemeine Syntax

Der Befehl zur Erzeugung neuer Daten lautet `insert`. Nach diesem Schlüsselwort wird zunächst angegeben, in welche Tabelle die Daten eingefügt werden sollen. Zwischen dem Schlüsselwort `insert` und dem Tabellennamen wird, um einen einigermaßen aussagekräftigen englischen Satz zu erhalten, noch das Schlüsselwort `into` eingefügt. Nach der Tabellenangabe folgt normalerweise (allerdings ist dies optional) eine Liste der Spalten, die hier, im Gegensatz zur `select`-Klausel, jedoch in Klammern gesetzt werden muss.

Diese Liste legt die Reihenfolge und die Auswahl der Spalten fest, in die die folgenden Daten eingefügt werden sollen. Anschließend folgt entweder eine Liste der einzufügenden Werte, die durch das Schlüsselwort `values` eingeleitet wird, oder eine `select`-Abfrage, die die benötigten Zeilen liefert.

INSERT-Anweisung mit VALUES-Klausel

Auch hier sehen wir uns zunächst ein einfaches Beispiel an. Wir möchten einen neuen Mitarbeiter in die Tabelle `HR_EMPLOYEES` einfügen. Bevor wir dies tun, wäre die Frage zu klären, welche Spalten unbedingt angegeben werden müssen. Wissen Sie die Antwort? Unbedingt angegeben werden müssen alle Spalten, die in der Tabelle einen Constraint `not null` besitzen. Dieser kann entweder direkt oder als Teil eines Primärschlüssels vergeben worden sein. Abziehen können Sie hiervon die Spalten, die einen Standardwert besitzen, falls Sie diesen verwenden möchten. Das kann zum Beispiel das aktuelle Tagesdatum für das Anlagedatum einer Zeile sein oder ein Primärschlüsselwert, der aus einer Sequenz abgeleitet wird. Eine schnelle Analyse der Tabelle `HR_EMPLOYEES` zeigt uns in Spalte `NULLABLE`, dass außer der Primärschlüsselspalte `EMP_ID` die Spalten `EMP_LAST_NAME`, `EMP_EMAIL`, `EMP_JOB_ID` und `EMP_HIRE_DATE` eingefügt werden müssen (Abbildung 10.1), wobei die Spalten `EMP_ID` und `EMP_HIRE_DATE` über Standardwerte verfügen.

Spalten | Daten | Constraints | Zugriffsberechtigungen | Statistiken | Trigger | Flashback | Abhängigkeiten | Details | Partitionen | Indizes | SQL

Aktionen...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID
1	EMP_ID	NUMBER(6,0)	No	"SQL_BUCH"."HR_EMPLOYEES_SEQ"."NEXTVAL"	1
2	EMP_FIRST_NAME	VARCHAR2(20 CHAR)	Yes	(null)	2
3	EMP_LAST_NAME	VARCHAR2(25 CHAR)	No	(null)	3
4	EMP_EMAIL	VARCHAR2(25 CHAR)	No	(null)	4
5	EMP_PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5
6	EMP_HIRE_DATE	DATE	No	trunc(sysdate)	6
7	EMP_JOB_ID	VARCHAR2(10 CHAR)	No	(null)	7
8	EMP_SALARY	NUMBER(8,2)	Yes	(null)	8
9	EMP_COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9
10	EMP_EMP_ID	NUMBER(6,0)	Yes	(null)	10
11	EMP_DEP_ID	NUMBER(4,0)	Yes	(null)	11

Abbildung 10.1 NOT-NULL-Constraints der Tabelle HR_EMPLOYEES

Um einen Primärschlüsselwert anzugeben, existieren verschiedene Möglichkeiten, die ich in Abschnitt 10.6, »Sequenzen und Trigger«, näher erläutern werde. Diese Tabelle definiert für die Primärschlüsselspalte einen Standardwert, der sich den neuen Schlüssel aus einer Sequenz holt. Darauf werden wir uns beim Einfügen der neuen Zeile verlassen, ebenso wie auf den Standardwert für das Einstelldatum.

```
SQL> insert into hr_employees(emp_last_name, emp_email, emp_job_id)
  2  values ('Meier', 'PMEIER', 'IT_PROG');
1 Zeile wurde erstellt.

SQL> select emp_id, emp_last_name, emp_email, emp_hire_date, emp_job_id
  2    from hr_employees
  3   where emp_email = 'PMEIER';

EMP_ID EMP_LAST_NAME EMP_EMAIL EMP_HIRE_DATE EMP_JOB_ID
------ ------------- --------- ------------- ----------
   316 Meier         PMEIER    28.07.2024    IT_PROG

SQL> commit;
Transaktion mit COMMIT abgeschlossen.
```

Listing 10.1 Einfache INSERT-Anweisung

Sehen wir uns an, was ich vorab bereits beschrieben habe: Dem Schlüsselwort `insert` folgt in Klammern die Liste der Spalten, die wir mit Werten belegen möchte. Im Beispiel sind nur die Pflichtfelder ohne die Standardfelder belegt worden. Dann folgt das Schlüsselwort `values`, mit dem, ebenfalls in Klammern, eine Liste von Werten übergeben werden kann. Die Reihenfolge, in der ich nun die Werte angebe, entspricht der Spaltenliste in der `insert`-Klausel. Beachten Sie, dass alle Spalten, die von uns keinen

Wert erhalten haben, einen null-Wert anzeigen. Nachdem wir diese Anweisung abgeschickt haben, ist die Datenbank in einem speziellen Modus, sie hat eine *Transaktion* geöffnet. In diesem Modus gelten besondere Regeln:

- Ein anderer Benutzer könnte zu diesem Zeitpunkt die neu eingefügten Daten nicht sehen, nur wir in unserer Session. Dies gilt, bis wir die Transaktion explizit beenden.
- Weitere insert-, update- oder delete-Anweisungen sind möglich und werden ebenfalls nur in unserer Session sichtbar sein.
- Die geänderten Daten können bereits jetzt innerhalb unserer Session durch select-Anweisungen gelesen und damit geprüft werden.
- Abschließend schließen wir unsere Transaktionsklammer durch die Anweisung commit oder rollback. Erst jetzt sind unsere neuen Daten für andere Benutzer sichtbar.

Den Begriff der Transaktion werden wir noch genauer in Abschnitt 10.7, »Ihr Sicherheitsnetz – die Transaktion«, besprechen.

Version 23ai erweitert die Möglichkeiten der values-Klausel dadurch, dass beliebig viele Zeilen, in Klammern gesetzt und durch Komma getrennt, verwendet werden dürfen. Bis zu dieser Version ist die in Listing 10.1 gezeigte Form auf eine Zeile beschränkt. Diese Schreibweise ersetzt den »Trick«, den ich in Abschnitt 5.5, »Mengenoperationen mit UNION, MINUS und INTERSECT«, beschrieben hatte, der auf der Zusammenstellung von Werten über eine select-Anweisung mit union all beruht. Weil die Schreibweise aus Listing 10.2 so einfach zu verwenden ist, gehe ich davon aus, dass sie der neue Standard für das Einfügen mehrerer Werte werden wird.

```
SQL> insert into hr_employees(emp_last_name, emp_email, emp_job_id)
  2  values
  3    ('Meier', 'PMEIER', 'IT_PROG'),
  4    ('Müller', 'FMUELLER', 'IT_PROG');
2 Zeilen eingefügt

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht
```

Listing 10.2 Verwendung der VALUES-Tabellenfunktion mit INSERT

Lassen Sie uns noch einen Moment bei dieser Variante der insert-Anweisung bleiben. Wir haben eine Reihe an Optionen zur gezeigten Schreibweise: Zum einen könnten wir in der insert-Klausel die Spaltenliste weglassen. Als Folge müssen wir alle Spalten mit Werten belegen, die in der Tabelle enthalten sind, und zwar in der Reihenfolge, in der sie in der Tabelle definiert wurden. Diese Reihenfolge können Sie in Abbildung

10.1 erkennen, und zwar durch die Werte der Spalte COLUMN_ID. Mittlerweile kennen Sie mich aber gut genug, um nicht überrascht zu sein, wenn ich Ihnen dieses Verfahren nicht empfehle: Eine insert-Anweisung wird besser dokumentiert und läuft zudem sicherer, wenn Sie explizit die Reihenfolge der Spalten festlegen. Interessanterweise könnte nämlich die Reihenfolge der Spalten zwischen verschiedenen Datenbanken mit gleichem Datenmodell (zum Beispiel einer Test-, Entwicklungs- und Produktionsdatenbank) voneinander abweichen, wenn Spalten in der Entwicklungsdatenbank hinzugefügt, für das Installationsskript auf der Produktionsdatenbank jedoch »aufgeräumt« und in ihrer Reihenfolge optimiert wurden. In solchen Fällen gelingen Ihre insert-Anweisungen auf einer der beteiligten Datenbanken nicht mehr, weil dort die Standardreihenfolge der Spalten mit Ihrer insert-Anweisung nicht übereinstimmt. Übergeben Sie die Spalten als Liste, ist die Reihenfolge in der Tabelle egal.

Als weitere Option könnten wir eine Spalte in der insert-Klausel zwar definieren, ihr aber keinen Wert zuweisen wollen. Dies können wir erreichen, indem wir positional korrekt für diese Spalte den Wert NULL übergeben. Alternativ können wir auch, wenn wir einen Spaltenwert nicht belegen möchten, diesen in der expliziten Spaltenliste weglassen, wie wir das in der Beispielanweisung gemacht haben.

Schließlich können wir in der insert-Anweisung eine Reihe von SQL-Zeilenfunktionen verwenden, wie zum Beispiel die Funktionen sysdate oder user, aber auch to_date, to_number etc. sowie skalare Unterabfragen, die Sie in Kapitel 8, »Unterabfragen«, kennengelernt haben. Listing 10.3 zeigt ein Beispiel für eine solche Anweisung. Sie verwendet einerseits eine komplette Spaltenliste in der insert-Klausel, zum anderen wird die Spalte COMM mit einem null-Wert belegt, da sie in der expliziten Spaltenliste nicht auftaucht, mit einer Zeilenfunktion in der Spalte EMP_HIRE_DATE sowie drei skalaren Unterabfragen, die ein durchschnittliches Gehalt, auf 100 gerundet, den IT-Manager als Vorgesetzten und die Abteilungsnummer der anderen Programmierer erfragen.

```
SQL> insert into hr_employees
  2    (emp_last_name, emp_email, emp_job_id, emp_emp_id,
  3     emp_hire_date, emp_salary, emp_dep_id)
  4  values
  5    ('Müller', 'FMUELLER', 'IT_PROG',
  6     (select emp_id
  7        from hr_employees
  8       where emp_job_id = 'IT_MGR'),
  9     trunc(sysdate - interval '1' day),
 10     (select round(avg(emp_salary), -2)
 11        from hr_employees
 12       where emp_job_id = 'IT_PROG'),
 13     (select min(emp_dep_id)
```

```
 14        from hr_employees
 15       where emp_job_id = 'IT_PROG'));
1 Zeile wurde erstellt.

SQL> select emp_id, emp_last_name, emp_emp_id,
  2         emp_hire_date, emp_salary, emp_dep_id
  3    from hr_employees
  4   where emp_email = 'FMUELLER';

EMP_ID EMP_LAST_NAME EMP_EMP_ID EMP_HIRE_DATE EMP_SALARY EMP_DEP_ID
------ ------------- ---------- ------------- ---------- ----------
   318 Müller               103 27.07.2024          5000         60
```

Listing 10.3 Eine komplexe INSERT-Anweisung

Die einzelnen Bestandteile der Anweisung kennen Sie bereits, doch ist die Zusammenstellung in diesem Kontext möglichweise noch etwas fremd und gewöhnungsbedürftig. Ich kann aus meiner Erfahrung heraus allerdings sagen, dass `insert`-Anweisungen selten so komplex sind wie im zweiten Beispiel. Viel häufiger sind einfache `insert`-Anweisungen wie im ersten Beispiel.

INSERT-Anweisung mit SELECT-Abfrage

Recht häufig ist das zweite Verfahren, das ich eingangs beschrieben habe: die `insert`-Anweisung mit einer `select`-Abfrage. Die `select`-Abfrage hat die Aufgabe, eine Liste von Werten zu generieren, die anschließend in die Zieltabelle eingefügt werden. Der erste Teil der `insert`-Anweisung ist dabei, inklusive aller besprochenen Optionen, identisch mit der ersten Variante. Die Unterschiede liegen also in der Fortführung der Anweisung. Auch hier verwende ich in Listing 10.4 ein etwas seltsames Beispiel, das aber den Vorteil hat, mit unseren Daten direkt zu funktionieren. Im Gegensatz zu vorhin werde ich diese Daten nach dem Einfügen aber direkt wieder entfernen, da diese Daten ansonsten bei weiteren Abfragen stören.

Die `select`-Anweisung ist nichts Besonderes, außer natürlich, dass die Reihenfolge und Anzahl der Spalten der Abfrage in Reihenfolge und Anzahl mit der Spaltenliste der `insert`-Klausel übereinstimmen müssen. Mit dieser Form der Anweisung erreichen wir, dass `108` Zeilen auf einmal erzeugt werden können. Andererseits ist dieser Weg nur dann möglich, wenn die Daten über SQL verfügbar sind.

```
SQL> insert into hr_employees
  2    (emp_id, emp_last_name, emp_job_id,
  3     emp_emp_id, emp_email, emp_hire_date,
  4     emp_salary, emp_commission_pct, emp_dep_id)
  5  select emp_id + 1000, emp_last_name, emp_job_id,
```

```
  6         emp_emp_id, concat(emp_email, 'BAC'), trunc(sysdate),
  7         emp_salary, emp_commission_pct, emp_dep_id
  8    from hr_employees;
108 Zeilen erstellt.

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.4 Eine INSERT-Anweisung mit SELECT-Klausel

Sind die Daten nicht verfügbar, ist dennoch eine mengenorientierte Verarbeitung möglich, indem die Daten ab Version 23ai durch eine `values`-Tabellenfunktion bereitgestellt werden, wie in Listing 10.2 gezeigt, oder in der älteren Variante mit der Kombination aus skalaren Abfragen gegen die Tabelle `DUAL` und der Mengenoperation `union all`, wie in Abschnitt 5.5, »Mengenoperationen mit UNION, MINUS und INTERSECT«, beschrieben.

Als weitere Möglichkeit können Sie Tabellen als sogenannte *externe Tabellen* anlegen. Dies bedeutet, dass die Daten dieser externen Tabellen nicht in der Datenbank, sondern in einer Textdatei im Betriebssystem vorgehalten werden, zum Beispiel in einer CSV-Datei. Sie beschreiben nun die Struktur dieser CSV-Datei, richten den Zugriff der Datenbank auf das externe Verzeichnis ein und können anschließend auf diese Tabellen zugreifen. Externe Tabellen besprechen wir nicht im Detail, sondern lediglich summarisch in Kapitel 12, »Tabellen erstellen«.

Wichtig zu wissen ist, dass eine mengenorientierte Verarbeitung wesentlich für eine schnelle Verarbeitung von Daten ist. Das ist eine der wenigen allgemeinen Wahrheiten bezüglich der Datenverarbeitung in Datenbanken. Um Daten mengenorientiert zu verarbeiten, lohnt ein höherer Aufwand bei der Bereitstellung der Daten. Die Unterschiede in der Ausführungszeit können um den Faktor 100 schneller sein als eine Einzelsatzverarbeitung.

Als letzte Anmerkung können Sie im Rahmen der `select`-Abfrage alles einsetzen, was Sie in SQL können. Oft wird eine solche `insert`-Anweisung mit einer komplexen `select`-Abfrage gewählt, um komplexe Berichte in eigenen Berichtstabellen zu speichern. Auf diese Weise können diese Berichte schneller verfügbar gemacht oder über die Zeit gespeichert werden. Gewissermaßen eine Variante dieses Verfahrens stellt zudem noch die *Materialized View (materialisierte View)* dar, bei der eine `select`-Abfrage nicht nur in der Datenbank gespeichert, sondern die Abfrageergebnisse auf der Festplatte hinterlegt werden. Hier dient die `select`-Abfrage, die Sie in der Datenbank speichern, als Quelle für eine `insert`-Anweisung, die das Abfrageergebnis speichert. Allerdings haben Sie nicht direkt mit den beiden Befehlen zu tun, das wird durch Oracle im Hintergrund erledigt. Materialized Views besprechen wir in Abschnitt 11.4.

10.2 Die UPDATE-Anweisung

Als Nächstes möchte ich Ihnen die update-Anweisung vorstellen. Mit dieser Anweisung ist es möglich, bereits bestehende Daten in Tabellen zu ändern. Die update-Anweisung wird auch genutzt, um Zellwerte einer Zeile zu löschen, denn eine delete-Anweisung löscht stets die gesamte Zeile, niemals jedoch einzelne Zellwerte innerhalb einer Zeile.

10.2.1 Allgemeine Syntax

Auch diese Anweisung beginnt mit einem neuen Schlüsselwort: update. Anschließend wird, ähnlich der insert-Anweisung, zunächst festgelegt, welche Tabelle betroffen ist. Im Gegensatz zur insert-Anweisung ist diesmal allerdings nicht das Schlüsselwort into erforderlich. Nach der Angabe der Tabelle erfolgt die Zuweisung der Spaltenwerte nach dem Schlüsselwort set mit Hilfe einer einfachen Syntax, die nach dem Prinzip alter_spaltenwert = neuer_spaltenwert aufgebaut ist und mittels einer kommaseparierten Liste die zu ändernden Spalten angibt. Anschließend wird die Anweisung durch eine optionale where-Klausel eingeschränkt. Diese where-Klausel ist syntaktisch identisch aufgebaut wie die entsprechende Klausel in der select-Abfrage und dient dem gleichen Zweck: der Auswahl der Zeilen. Hier jedoch dient die Filterung dazu, die Zeilen zu wählen, die durch die update-Anweisung geändert werden sollen.

Auch hier soll Ihnen das Beispiel aus Listing 10.5 den Überblick erleichtern. Ich möchte gern jedem SALESMAN eine Gehaltskürzung von 50 Talern zumuten und gleichzeitig eine Steigerung der Boni von 10 % einräumen.

```
SQL> select emp_last_name, emp_job_id, emp_salary, emp_commission_pct
  2    from hr_employees
  3   where emp_job_id = 'SA_REP';

EMP_LAST_NAME   EMP_JOB_ID EMP_SALARY EMP_COMMISSION_PCT
--------------- ---------- ---------- ------------------
Tucker          SA_REP          10000                ,3
Bernstein       SA_REP           9500               ,25
Hall            SA_REP           9000               ,25
Olsen           SA_REP           8000                ,2
Cambrault       SA_REP           7500                ,2
Tuvault         SA_REP           7000               ,15
30 Zeilen wurden ausgewählt.

SQL> update hr_employees
  2     set emp_salary = emp_salary - 50,
  3         emp_commission_pct = emp_commission_pct * 1.1
```

```
  4   where emp_job_id = 'SA_REP';
30 Zeilen aktualisiert.

SQL> select emp_last_name, emp_job_id, emp_salary, emp_commission_pct
  2    from hr_employees
  3   where emp_job_id = 'SA_REP';

EMP_LAST_NAME   EMP_JOB_ID EMP_SALARY EMP_COMMISSION_PCT
--------------- ---------- ---------- ------------------
Tucker          SA_REP           9950                ,33
Bernstein       SA_REP           9450                ,28
Hall            SA_REP           8950                ,28
Olsen           SA_REP           7950                ,22
Cambrault       SA_REP           7450                ,22
Tuvault         SA_REP           6950                ,17

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.5 Eine einfache UPDATE-Anweisung

Es zahlt sich nun aus, dass Sie bereits Erfahrung mit der `select`-Abfrage und deren `where`-Klausel gesammelt haben, denn diese Anweisung verstehen Sie nun sofort, allerdings sind einige Dinge doch erwähnenswert:

- Ich habe damit begonnen, zunächst eine `select`-Abfrage zu formulieren, die die Daten anzeigt, die ich ändern möchte. Dies kann ich Ihnen, insbesondere zu Beginn, nur dringend empfehlen, denn ich denke, auch Sie werden etwas hektisch, wenn als Antwort auf Ihre `update`-Anweisung `14 Millionen Zeilen aktualisiert` zurückkommt, obwohl Sie doch nur die paar Bestellungen von gestern aktualisieren wollten. Nebenbei: Was machen Sie nun? Ich hoffe, wir sind uns einig: Die nächste Anweisung lautet `rollback`! Dieses Problem können Sie durch eine vorangehende `select`-Abfrage aus der Welt schaffen. Syntaktisch können Sie aus einer `select`-Abfrage leicht eine `update`-Anweisung machen, denn die `select`-Liste wird einfach durch das Schlüsselwort `update` ersetzt, und zwischen Tabellenname und `where`-Klausel fügen Sie die `set`-Klausel ein.
- Nach dem Update wird zurückgemeldet, wie viele Zeilen durch die Anweisung geändert wurden. Prüfen Sie das immer, bevor Sie ein `commit` absetzen. Gerade `update`-Anweisungen sind oftmals schwer reversibel; denken Sie nur daran, wie Sie nach einem `commit` reagieren möchten, wenn nun alle Mitarbeiter in Abteilung `40` arbeiten. In welchen Abteilungen haben die Mitarbeiter vorher gearbeitet? Diese Information ist nun verloren (na ja, jedenfalls im schlechtesten Fall).

10.2.2 Variationen zum Thema

Eigentlich war es das bereits, was Sie über eine `update`-Anweisung wissen müssen: Sie wählen mit der `where`-Klausel aus, welche Zeilen Sie aktualisieren möchten, und teilen in der `set`-Klausel mit, welche Spalten wie verändert werden sollen. Kompliziert wird die `update`-Anweisung, wenn Sie die neuen Werte nicht einfach aus den bestehenden Daten ableiten können, sondern aus anderen Tabellen kopieren müssen.

Ein Beispiel wäre, dass Sie den aktuellen Datenbestand bereits in einer Tabelle vorliegen haben und diese nun auf die lokale Tabelle übernehmen möchten. Leider schlug bis zur Version 23ai der Versuch fehl, in der `update`-Anweisung direkt eine andere Tabelle über eine Join-Bedingung anzubinden, wie etwa in Listing 10.6.

```
SQL> update hr_employees l
  2     set l.emp_salary = e.emp_salary
  3    from hr_employees_extern e
  4   where l.emp_id = e.emp_id;
ORA-00933 SQL-Befehl wurde nicht korrekt beendet.

-- Ab Version 23ai
SQL> update hr_employees l
  2     set l.emp_salary = e.emp_salary
  3    from hr_employees_extern e
  4   where l.emp_id = e.emp_id;
5 Zeilen aktualisiert.
```

Listing 10.6 Neue Syntax für UPDATE-Anweisungen in 23ai

Dann helfen Ihnen skalare Unterabfragen, die oftmals die Form einer harmonisierten Unterabfrage haben. In Listing 10.7 sehen Sie ein Beispiel: Wir möchten allen Mitarbeitern einer Berufsgruppe als neues Gehalt das Mindestgehalt, das in dieser Gruppe gezahlt wird, zuweisen. Sie merken schon: Ich bin bei meinen Beispielen besonders sozial eingestellt. Das täuscht aber, im echten Leben bin ich eigentlich ganz nett ...

Die syntaktischen Details entsprechen dem, was wir in Kapitel 8, »Unterabfragen«, bereits über skalare Unterabfragen gesagt haben, daher verweise ich bei Fragen auf die Besprechung dort. Wir benötigen für eine harmonisierte Unterabfrage ein Tabellenalias auf die zu aktualisierende Tabelle, damit wir diese aus der Unterabfrage referenzieren können.

```
SQL> update hr_employees e
  2     set emp_salary =
  3         (select min(m.emp_salary)
  4            from hr_employees m
```

```
  5           where m.emp_job_id = e.emp_job_id);
107 Zeilen wurden aktualisiert.

Ausführungsplan
----------------------------------------------------------------------
| Id  | Operation             | Name         | Rows  | Cost (%CPU)|
----------------------------------------------------------------------
|   0 | UPDATE STATEMENT      |              |   341 |     8  (25)|
|   1 |  UPDATE               | HR_EMPLOYEES |       |            |
|*  2 |   HASH JOIN OUTER     |              |   341 |     8  (25)|
|   3 |    TABLE ACCESS FULL  | HR_EMPLOYEES |   108 |     3   (0)|
|   4 |    VIEW               | VW_RT_0      |    20 |     4  (25)|
|   5 |     SORT GROUP BY     |              |    20 |     4  (25)|
|   6 |      TABLE ACCESS FULL| HR_EMPLOYEES |   108 |     3   (0)|
----------------------------------------------------------------------

SQL> select emp_last_name, emp_job_id, emp_salary
  2    from hr_employees;

EMP_LAST_NAME             EMP_JOB_ID EMP_SALARY
------------------------- ---------- ----------
OConnell                  SH_CLERK         2500
Grant                     SH_CLERK         2500
Whalen                    AD_ASST          4400
Hartstein                 MK_MAN          13000
Fay                       MK_REP           6000
Mavris                    HR_REP           6500
Baer                      PR_REP          10000
Higgins                   AC_MGR          12008
Gietz                     AC_ACCOUNT       8300
…
```

Listing 10.7 Rosskur: Alle auf Start zurück, nur dem Präsidenten ist's egal.

Ein Punkt, der Fragen aufwirft, könnte sein: Wie schafft es die Datenbank eigentlich, das Minimum zu berechnen, wenn wir die Tabelle gleichzeitig ändern? Skalare Unterabfragen werden immer zuerst berechnet und die Ergebnisse als Konstanten durch die `update`-Anweisung referenziert. Dieses Verhalten können Sie schön am Ausführungsplan der Anweisung sehen.

Um die `update`-Anweisung auszuführen, wird zunächst ab Zeile 4 die Unterabfrage berechnet. Erst danach wird der Rest ermittelt.

Einen Wermutstropfen hat diese Strategie allerdings: Die Performanz kann unter Umständen sehr langsam sein. Das Problem der harmonisierten Unterabfrage ist, dass diese Form der Abfrage nicht in jedem Fall durch die Datenbank so optimiert werden kann, dass eine wirklich performante Aktualisierung sichergestellt ist. Ich erinnere mich an ein Problem bei einer Tabelle mit ca. 75.000 Zeilen. Aus diesem Datenbestand wurden Dubletten entfernt. Dadurch entstanden Lücken in einer vorher lückenlosen Nummerierung, die nun, aufgrund von Geschäftsanforderungen, geschlossen werden sollten. Dabei handelte es sich nicht um die Primärschlüsselinformationen. Wie geht man nun vor? Ich finde, dass diese Diskussion ganz interessant ist, denn sie zeigt einige durchaus witzige Nebeneffekte.

Die folgenden Anweisungen beziehen sich auf eine fiktionale Tabelle, daher habe ich die hierfür verwendeten Daten auch nicht in den Beispielskripten hinterlegt. Sie dienen eher der Darstellung eines Sachverhaltes und können auch ohne konkrete Daten nachvollzogen werden.

```
SQL> select *
  2    from prod_table;

   PROD_ID PROD_NAME          PROD_DESC            PROD_TOTAL_ID
---------- ------------------ -------------------- -------------
        25 5MP Telephoto      Camera 5MP Telephoto            25
        27 HDTV Tuner         17" LCD w/ HDTV Tuner           27
        29 Envoy 256MB - 40GB Envoy 256MB - 40Gb              29
        31 Y Box              Y Box                           31
…
72000 Zeilen ausgewählt
```

Listing 10.8 Beispieltabelle für die UPDATE-Anweisung

Listing 10.8 zeigt eine Tabelle mit vielen Produkten. Spalte `PROD_TOTAL_ID` enthält, wie Sie sehen, Lücken und soll nun aufsteigend nummeriert werden, ohne die Sortierung der Daten nach der `PROD_ID` zu ändern. Wenn wir uns nun die folgenden Lösungsansätze ansehen, bitte ich Sie, sich vor dem Weiterlesen Gedanken darüber zu machen, ob der jeweils eingeschlagene Weg funktionieren wird oder nicht. Für alle `update`-Anweisungen lassen wir uns zunächst nur den geplanten Ausführungsplan ausgeben, um zu sehen, ob unsere Strategie sinnvoll ist oder nicht.

Listing 10.9 zeigt einen ersten Versuch, eine `update`-Anweisung für dieses Problem zu erstellen. Bereits die Schätzung der Datenbank lässt uns nichts Gutes ahnen, denn die Zeit für die `update`-Anweisung wird mit ca. 50 Minuten veranschlagt. Zudem hat die harmonisierte Unterabfrage ein grundsätzliches Problem. Sehen Sie das? Machten wir die `update`-Anweisung auf diese Weise, wären nachher alle `PROD_ID`-Spalten = 1. Warum ist das so? Ganz einfach: Durch den Join in die harmonisierte Unterabfrage

wird für jede Zeile die innere Zeile neu gerechnet. Daher ist überall die Zeilennummer gleich 1. Lösen wir also dieses Problem in der zweiten Anweisung, doch es bleibt beim schlechten Ausführungsplan, den ich daher hier nicht noch einmal ausgebe.

```
SQL> explain plan for
  2  update prod_table p
  3     set prod_total_id =
  4         (select rownum
  5            from prod_table pt
  6           where pt.prod_id = p.prod_id);
EXPLAIN PLAN ausgeführt.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------
| Id  | Operation           | Name       | Cost   | Time
------------------------------------------------------------------
|   0 | UPDATE STATEMENT    |            |15M  (2)| 50:06:23
|   1 |  UPDATE             | PROD_TABLE |        |
|   2 |   TABLE ACCESS FULL | PROD_TABLE |189  (1)| 00:00:03
|   3 |   COUNT             |            |        |
|*  4 |    TABLE ACCESS FULL| PROD_TABLE |189  (1)| 00:00:03
------------------------------------------------------------------

SQL> explain plan for
  2  update prod_table p
  3     set prod_total_id =
  4         (select rn
  5            from (select prod_id, rownum rn
  6                    from prod_table) pt
  7           where pt.prod_id = p.prod_id);
```

Listing 10.9 Erster Versuch

Wenn wir dieses Update ausführen, kommt noch, je nach Umgebung, ein witziges Problem hinzu, das mir klar wurde, nachdem uns von der Fachseite mitgeteilt wurde, dass die Sortierung dieser Aktualisierung nur teilweise gelingt. Etwa 70–80 Zeilen sind korrekt sortiert, dann kommt ein Block anderer Produkte, die in sich korrekt sortiert wurden, als Block jedoch nicht hinter die ersten Produkte gehörten usw. Was ist hier passiert? Das Problem liegt in der Ermittlung der neuen Zeilennummer, denn in aller Regel wird die Datenbank diese `update`-Anweisung auf mehrere Prozesse verteilen und parallel ausführen. Nun wird in jedem Teil die neue `PROD_ID` in der Reihenfolge der Bearbeitung korrekt ermittelt, doch ist der Startwert der neuen `PROD_ID` abhängig davon, welcher Prozess wann mit seinem Teil der Anweisung fertig ist.

Dadurch wird die Sortierung sozusagen »blockweise zufällig«. Wir benötigen zusätzlich noch ein definiertes Sortierkriterium, um sicher zu sein, dass die Sortierung erhalten bleibt. Unsere endgültige `update`-Anweisung sehen Sie in Listing 10.10. Ich habe daher die Anweisung ausgeführt, um zu prüfen, wie genau die Vorhersagen der Zeit tatsächlich sind.

```
SQL> explain plan for
  2  update prod_table p
  3     set prod_total_id =
  4         (select rownum
  5            from (select prod_id
  6                    from prod_table
  7                   order by prod_id) pt
  8           where pt.prod_id = p.prod_id);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------
| Id | Operation            | Name       | Cost (%CPU)| Time
--------------------------------------------------------------------
|  0 | UPDATE STATEMENT     |            |    13M  (1)| 45:21:39
|  1 |  UPDATE              | PROD_TABLE |            |
|  2 |   TABLE ACCESS FULL  | PROD_TABLE |   189   (1)| 00:00:03
|  3 |   COUNT              |            |            |
|  4 |    VIEW              |            |   189   (1)| 00:00:03
|* 5 |     TABLE ACCESS FULL| PROD_TABLE |   189   (1)| 00:00:03
--------------------------------------------------------------------

SQL> set timing on
SQL> update prod_table p
  2     set prod_total_id =
  3         (select rownum
  4            from (select prod_id
  5                    from prod_table
  6                   order by prod_id) pt
  7           where pt.prod_id = p.prod_id);
72000 Zeilen wurden aktualisiert.
Abgelaufen: 00:08:52.86

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
Abgelaufen: 00:00:00.56
```

Listing 10.10 UPDATE-Anweisung zur Renummerierung

Nun ja, das Ergebnis ist nicht ganz so schlecht wie befürchtet, aber für das, was getan wurde, doch recht lang und zudem nicht repräsentativ, es hängt von der Datenbankversion, der Systemausstattung und vielem mehr ab. Erfahrene SQL-Entwickler werden auch andere Strategien vorschlagen, die in diesem konkreten Fall eventuell sogar deutlich schneller sein können. Der Punkt ist aber: Selbst eine relativ einfache Anforderung wie diese hier kann beim »normalen« Einsatz der `update`-Anweisung zu immensen Problemen führen, auch wenn Sie alles in Ihrer (SQL-)Macht Stehende getan haben, um die Gewinnung der neuen Daten zu beschleunigen. Wie schnell 72.000 Zeilen geschrieben werden können, sehen Sie an der Dauer der `rollback`-Anweisung, bei der alle alten Datenzustände wieder zurückgeschrieben werden müssen. Das, kombiniert mit der Ausführungszeit für die Ermittlung der neuen Nummern, deutet darauf hin, dass die Arbeit in etwa 1 Sekunde hätte erledigt werden können. Offensichtlich bleibt das Problem bestehen, dass durch die harmonisierte Unterabfrage die Ermittlung der neuen Zeilennummern jeweils neu angestoßen wird.

Die Frage, die sich stellt, ist, wie man solche Anweisungen schneller ausführen kann. Ich komme im Verlauf des Kapitels noch einmal auf dieses Problem zurück und zeige Ihnen, wie wir dieses Problem einfach und performant lösen können.

Was wäre noch? An sich sind alle Varianten, die ich noch anführen könnte, aus diesen Beispielen ableitbar. Wenn Sie mit der `update`-Anweisung arbeiten, hat die Prüfung der `where`-Klausel absoluten Vorrang vor allem anderen. Anschließend kommt die Prüfung der `set`-Klausel. Beides ist bei Datenänderungen eminent wichtig!

10.3 Die DELETE-Anweisung

Die letzte Grundtechnik ist das Löschen von Zeilen mit Hilfe der `delete`-Anweisung. Das zusätzliche Schlüsselwort zwischen `delete` und dem Tabellennamen heißt `from`. Witzigerweise darf dieses Schlüsselwort weggelassen werden, was grammatikalisch Sinn macht, wenn einfach die gesamte Tabelle gelöscht werden soll. Ich schreibe es aus Gewohnheit allerdings immer hin. Schön an der Anweisung ist, dass es hier keine Komplikation gibt außer der bereits bekannten `where`-Klausel, die steuert, welche Zeilen gelöscht werden sollen. Daher ist es ganz einfach, sich das Beispiel aus Listing 10.11 klarzumachen.

```
SQL> delete from hr_employees
  2   where emp_id >= 315;
1 Zeile wurde gelöscht.

SQL> commit;
Transaktion mit COMMIT abgeschlossen.

SQL> delete hr_employees;
```

```
107 Zeilen wurden gelöscht.

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.11 Eine einfache DELETE-Anweisung

Mehr ist dazu nicht zu sagen. Auch hier steht immer wieder das große ACHTUNG-Schild, das Sie bei der Arbeit mit Datenmanipulationsbefehlen immer berücksichtigen sollten: Drum prüfe, wer auf ewig festschreibt … Machen Sie sich zur Gewohnheit, zunächst einmal mit einer `select`-Abfrage zu prüfen, ob Sie wirklich die richtigen Daten an den Flügeln haben, bevor Sie diese Daten löschen.

Eine Anmerkung hätte ich noch zum Thema *Löschen*, *Löschen* oder *Löschen*. Es gibt nämlich drei Löscharten, von denen die `delete`-Anweisung die erste (und gewissermaßen harmloseste) ist. Die beiden anderen Löscharten heißen `truncate` und `drop`. Im Gegensatz zu den beiden weiteren Varianten ist ausschließlich `delete` durch die Transaktionsklammer geschützt. Die beiden anderen Varianten löschen schnell, gründlich und ohne Nachfrage.

Allerdings gibt es noch weitere Unterschiede: Die `delete`-Anweisung ist ein »logisches Löschen«, will sagen, es wird tatsächlich eine einzelne Zeile einer Tabelle, basierend auf einer `where`-Klausel, gelöscht. Im Bild der Tabelle als Lagerhalle wäre das gleichbedeutend damit, dass ein einzelner Farbeimer aus dem Regal geräumt wird. Der Platz bleibt nun frei, die rote Fahne wird nicht bewegt. Im Gegensatz dazu räumt `truncate` ohne Transaktionsschutz rigoros das gesamte Lager leer und setzt die rote Fahne wieder auf den ersten Lagerplatz. Es handelt sich hierbei also um ein »physisches Löschen« auf der Festplatte, ohne Ansehen der einzelnen Zeilen und ohne mögliche Einschränkung über eine `where`-Klausel.

Als Letztes ist das `drop` das schlichte Entfernen des Lagers von dieser Welt. Es werden nicht nur, wie bei `truncate`, die Lagerplätze physisch gelöscht (und der dadurch allokierte Speicher auf der Platte freigegeben), sondern auch der Eintrag der Existenz der Tabelle im Data Dictionary, die Tabelle gibt es dadurch nicht mehr. Wobei gerade zu dieser letzten Variante tröstend zu sagen ist, dass Tabellen nicht physisch gelöscht, sondern in den *Recycle Bin* (wie das bei Oracle heißt) verschoben werden und aus diesem wieder hervorgeholt werden können. Details sehen wir uns in Kapitel 12, »Tabellen erstellen«, an.

10.4 Die MERGE-Anweisung

Immer noch relativ unbekannt ist die Anweisung `merge`. Allerdings ist diese Anweisung besonders elegant und wird von mir sehr oft verwendet. Es ist im Kern eine Kombination von `insert` und `update` mit einem kleinen `delete`-Anhängsel. Der große Vorteil dieser Kombination liegt darin, dass für viele Anwendungen, die vorher nur

über mehrere Anweisungen möglich waren, nun nur noch eine Anweisung erforderlich ist. Dies spart nicht nur Zeit, sondern ist insbesondere auch unter dem Aspekt der Datenkonsistenz wichtig, denn mehrere Anweisungen, die nacheinander ausgeführt werden, müssen nicht notwendigerweise die gleichen Daten sehen. So kann es passieren, dass sich Fehler einschleichen.

Nehmen wir ein Szenario für eine `merge`-Anweisung: Zwei Datenbanken erfassen Daten und sollen auf einen gemeinsamen Datenbestand gebracht werden. Die erste Tabelle fungiert als Zieltabelle, soll also die Daten beider Datenbanken enthalten, während die zweite Datenbank zur dezentralen Datenerfassung genutzt wird und ihre Daten lediglich abliefern soll. Nun werden sich die Daten in der zweiten Datenbank ändern oder neu hinzugefügt. Um die bestehenden Daten zu ändern, muss eine `update`-Anweisung ausgeführt werden und anschließend eine `insert`-Anweisung für alle neu aufgenommenen Daten. Wenn wir uns dies bei einer Datenbank vorstellen, die gleichzeitig von mehreren Benutzern genutzt wird, ist einsichtig, dass durch den Zeitversatz der beiden Anweisungen Datenänderungen eventuell nicht gesehen werden, die für eine korrekte Ausführung der Anweisung nötig gewesen wären. Zudem kann die betroffene Datenmenge sehr groß sein; zwei Anweisungen erzwingen, dass die Datenbank diese große Datenmenge zweimal bearbeitet. Diese Probleme löst die `merge`-Anweisung, denn sie erkennt innerhalb eines einzigen Durchlaufs über die Daten, ob eine Zeile bereits existiert oder nicht. Existiert eine Zeile in der Zieltabelle, wird ein `update` durchgeführt, anderenfalls ein `insert`. Zudem kann im Fall einer gefundenen Zeile angegeben werden, ob (besser: unter welcher Bedingung) diese gelöscht werden soll.

Ein anderes Szenario: Sie wissen, dass die Zeilen der zweiten Tabelle nicht geändert wurden, sondern lediglich neue Zeilen hinzukamen. In diesem Fall können Sie eine `merge`-Anweisung nutzen, um nur die Zeilen einzufügen, die noch nicht in der ersten Tabelle standen, indem Sie den `update`-Zweig der `merge`-Anweisung weglassen. Diese Art Abfrage hätte auch durch eine einfache `insert`-Anweisung durchgeführt werden können, indem Sie in der `insert`-Anweisung einen Anti-Join auf die Zieltabelle einfügen. Da beide Anweisungen nur dann Daten einfügen, wenn sich diese noch nicht in der Zieltabelle befinden, können Sie die Anweisungen so oft ausführen, wie Sie mögen, denn ab dem zweiten Durchlauf machen diese Anweisungen nichts mehr. Im direkten Vergleich ist die `merge`-Anweisung allerdings, wie Sie noch sehen werden, syntaktisch eleganter. Sie ist durch die Vielfalt der Funktionen etwas komplexer, daher unterteile ich die Besprechung der Optionen und beginne mit einem einfachen Beispiel.

10.4.1 Allgemeine Syntax

Die Syntax der `merge`-Anweisung ist etwas länger und beeindruckt eventuell durch ihre schiere Länge. Allerdings verfliegt dieser Eindruck schnell, wenn wir uns über die

Funktion der einzelnen Klauseln unterhalten haben. Dann wird die Anweisung sehr einfach durchschaubar.

Wir haben in der `merge`-Anweisung mehrere Klauseln. Zunächst beginnen wir mit der `merge`-Klausel, die, ähnlich wie die `insert`-Anweisung nach dem Schlüsselwort `into`, den Namen der Tabelle aufführt, in die unsere Daten hineingeschrieben werden sollen, also den der Zieltabelle. Wir benötigen in jedem Fall ein Tabellenalias für diese Tabelle, für das ich in `merge`-Anweisungen `t` wie `target` verwende:

```
merge into hr_employees t
```

Als Nächstes definieren wir, welche Daten in die Zieltabelle geschrieben werden sollen. Diese Klausel wird durch das Schlüsselwort `using` eingeleitet. Die Datenmenge, die wir einfügen wollen, kann entweder einfach eine Tabelle oder View sein oder aber, was häufiger ist, eine Unterabfrage. Diese Unterabfrage muss dann, wie alle Unterabfragen, in Klammern notiert werden. Auch diese Tabelle benötigt ein Tabellenalias, ich verwende `s` wie `source`. Dann sieht die Anweisung aus wie folgt:

```
merge into hr_employees t
using (<Unterabfrage>) s
```

Als Nächstes müssen wir klären, wann Zeilen der beteiligten Tabellen als zueinandergehörend angesehen werden. Das ist eine Join-Bedingung und Ihnen von daher nicht neu. Ebenso wenig neu ist die syntaktische Formulierung, wir lassen lediglich das Schlüsselwort `join` weg:

```
merge into hr_employees t
using (<Unterabfrage>) s
   on (t.emp_id = s.emp_id)
```

Bitte beachten Sie, dass, im Gegensatz zu einer Join-Bedingung, die `on`-Klausel einer `merge`-Anweisung in Klammern notiert werden muss.

Nun kommen die einzelnen Zweige. Unterschieden wird, ob die Join-Bedingung für die aktuelle Zeile der Quelltabelle eine Zeile in der Zieltabelle finden konnte. Ist das Ergebnis positiv, verzweigt die Anweisung in den `update`-Zweig, der durch die Schlüsselwörter `when matched then update set` eingeleitet wird. Am Ende wird daraus dann eine »normale« `update`-Anweisung, denn nach dem `set` folgt die kommaseparierte Liste der Spaltenzuweisungen, wie bei einer `update`-Anweisung. Ist das Ergebnis nicht positiv, verzweigt die Anweisung analog zum `insert`-Zweig: `when not matched then insert (…) values (…)`. Auch hier wird die Anweisung syntaktisch zu einer normalen `insert`-Anweisung mit einer `values`-Klausel. Tatsächlich können hier Literale, Zeilenfunktionen und sogar skalare Unterabfragen verwendet werden, üblich ist jedoch die Referenz auf die Quelltabelle, die ich wie gesagt normalerweise mit dem Alias `s` bezeichne.

Für unser Beispiel verwenden wir eine Unterabfrage auf die Tabelle HR_EMPLOYEES, die nur die Mitarbeiter der Abteilung 20 zeigt, dort einige Daten ändert und außerdem einen virtuellen Mitarbeiter hinzufügt. Ich denke, die Anweisung brauche ich nicht zu erläutern, sie dient lediglich der Simulation für Arbeiten an der Quelltabelle. Listing 10.12 zeigt diese Abfrage und die resultierenden Daten.

```
SQL> -- Unterabfrage der zu synchronisierenden Daten
SQL> select emp_id, emp_last_name, emp_emp_id, emp_job_id,
  2         case emp_last_name
  3         when 'Hartstein' then emp_salary + 20
  4         when 'Fay' then emp_salary - 50
  5         else emp_salary end emp_salary
  6    from hr_employees
  7   where emp_dep_id = 20
  8   union all
  9  select 320, 'Meier', 201, 'MK_REP', 5800;

EMP_ID EMP_LAST_NAME EMP_EMP_ID EMP_JOB_ID EMP_SALARY
------ ------------- ---------- ---------- ----------
   201 Hartstein            100 MK_MAN          13020
   202 Fay                  201 MK_REP           5950
   320 Meier                201 MK_REP           5800

SQL> -- Stand vor dem MERGE
SQL> select emp_id, emp_last_name, emp_salary
  2    from hr_employees
  3   where emp_dep_id = 20;

EMP_ID EMP_LAST_NAME EMP_SALARY
------ ------------- ----------
   201 Hartstein          13000
   202 Fay                 6000
2 Zeilen ausgewählt.

SQL> -- MERGE-Anweisung
SQL> merge into hr_employees t
  2  using (<Unterabfrage>) s
 13     on (t.emp_id = s.emp_id)
 14   when matched then update set
 15        t.emp_salary = s.emp_salary
 16   when not matched then insert(
 17          emp_id, emp_last_name, emp_emp_id, emp_job_id,
 18          emp_email, emp_hire_date, emp_dep_id, emp_salary)
```

```
 19        values(
 20          s.emp_id, s.emp_last_name, s.emp_emp_id, s.emp_job_id,
 21          s.emp_email, s.emp_hire_date, s.emp_dep_id, s.emp_salary);
3 Zeilen zusammengeführt.

SQL> -- Daten nach dem MERGE
SQL> select emp_id, emp_last_name, emp_salary
  2    from hr_employees
  3   where emp_dep_id = 20;

EMP_ID EMP_LAST_NAME EMP_SALARY
------ ------------- ----------
   201 Hartstein          13020
   202 Fay                 5950
   320 Meier               5800
3 Zeilen ausgewählt.
```

Listing 10.12 Beispiel einer MERGE-Anweisung

Das Besondere der `merge`-Anweisung besteht auch darin, dass Sie diese Anweisung mehrfach ausführen können, ohne Fehler zu provozieren. Dies liegt daran, dass nach dem ersten Aufruf die neue Zeile der Tabelle bekannt ist und beim erneuten Aufruf der Anweisung nicht erneut eingefügt wird.

Eine Anmerkung noch zur `on`-Klausel. Gut, dass in unserem Beispiel ganz zufällig eine Primärschlüsselbeziehung vorliegt (Sie merken, ich bin ein alter Fuchs ...), aber das muss, wie bei jedem Join, nicht sein. Oftmals haben Sie hier andere Beziehungen. Achten Sie aber darauf, dass die Join-Bedingung deterministisch und eindeutig ist. Es dürfen nicht mehrere Zeilen der Quelltabelle die Bedingung der `on`-Klausel erfüllen, ebenso wenig darf eine nicht deterministische Funktion wie `sysdate` verwendet werden. Sollten Sie hier etwas falsch machen, werden Sie mit einem etwas seltsam klingenden Fehler konfrontiert:

```
ORA-30926: Stabile Zeilengruppe in den Quelltabellen
           kann nicht eingelesen werden
```

Dieser Fehler scheint so gar nichts mit der eigentlichen Fehlerursache zu tun zu haben, hat er aber. Mit einer »stabilen Zeilengruppe« ist gemeint, dass die Bedingung der `on`-Klausel auf maximal genau eine Zeile der Quell- und Zieltabelle zeigt. Das ist bei einer `merge`-Anweisung erforderlich, damit insbesondere die `update`-Anweisung weiß, welche Zeile sie für das `update` verwenden soll. Die Klausel, mit der diese Beziehung hergestellt wird, sieht zwar wie eine normale Join-Bedingung aus, muss aber, als zusätzliche Anforderung, eine `1:0..1`-Beziehung zwischen den Tabellen herstel-

len können, einer Zeile der Quelltabelle darf also nur *keine* oder *eine* Zeile der Zieltabelle zugeordnet werden.

Eine Erweiterung hat die `merge`-Anweisung mit Version 23ai erfahren, denn sie unterstützt die neue `values`-Klausel, um Daten ohne Umwege über eine `union all`-Abfrage gegen `DUAL` direkt in die Anweisung zu schleusen. Im Gegensatz zu den bisher gezeigten Varianten hat die `merge`-Anweisung die zusätzliche Anforderung, dass die Spalten der `values`-Klausel durch ein Alias benannt werden müssen, da diese anschließend im `update`- oder `insert`-Zweig der Anweisung referenziert werden können müssen.

Die Spaltenaliasse werden nach der `values`-Klausel über eine in Klammern notierte Liste deklariert, wie das von der `insert`-Anweisung bekannt ist. Listing 10.13 zeigt diese Anwendung an einem Beispiel.

```
SQL> merge into hr_regions t
  2  using (values
  3          (1, 'Europe'),
  4          (2, 'Americas'),
  5          (3, 'Asia'),
  6          (4, 'Middle East and Africa'))
  7        s (reg_id, reg_name)
  8     on (t.reg_id = s.reg_id)
  9   when not matched then insert (reg_id, reg_name)
 10        values (s.reg_id, s.reg_name);
```

Listing 10.13 Verwendung der VALUES-Klausel in einer MERGE-Anweisung

10.4.2 Variationen zum Thema

Nachdem Sie das Grundprinzip verstanden haben, möchte ich Ihnen noch einige wenige, dafür aber häufig anzutreffende Varianten zeigen.

MERGE-Anweisung mit optionalem DELETE-Zweig

Eine syntaktisch andere Variante ist die Option, gleichzeitig noch Daten zu löschen. Wir können uns vorstellen, dass wir gleichzeitig noch einen Mitarbeiter aus der Tabelle entfernen möchten. Die Bedingung ist, dass sein Name `Adams` lauten soll. Dann ändern wir die Anweisung wie in Listing 10.14. Nun ist dieses Löschkriterium vielleicht nicht das spannendste. Öfter werden Sie finden, dass zum Beispiel Daten in einer Tabelle gelöscht werden, die älter als ein Stichpunktdatum sind, andere Daten aktualisiert und wieder andere Daten hinzugefügt werden. Denkbar wäre so etwas zum Beispiel in Tabellen, die speziell nur einen definierten Ausriss aller Daten enthalten sollen. Solche Systeme finden sich in Anwendungen, die Teildaten einer großen Datenbank für eine genauere Untersuchung bereitstellen oder Ähnliches.

```
SQL> merge into hr_employees t
  2  using (<Unterabfrage>) s
 13     on (t.emp_id = s.emp_id)
 14   when matched then update set
 15        t.emp_salary = s.emp_salary
 16        delete where t.emp_last_name = 'Adams'
 17   …
3 Zeilen integriert.
```

Listing 10.14 MERGE-Anweisung mit DELETE-Zweig

MERGE-Anweisung mit konditionalem UPDATE

Eine weitere Option besteht darin, dass wir bei einem Match zwischen den Tabellen das `update` noch an Bedingungen knüpfen können. Es wird etwas schwierig mit Beispieldaten, doch vielleicht können wir ja mit der Formulierung aus Listing 10.15 dafür sorgen, dass nicht das Gehalt mit jedem Aufruf der Anweisung immer kleiner wird. In diesem Beispiel haben wir eine besondere Situation dadurch, dass die Daten der Unterabfrage auf `HR_EMPLOYEES` basieren und sich das Ergebnis der Unterabfrage beim nächsten Aufruf ändert. Daher ist es schwierig, zu verhindern, dass bei erneutem Aufruf der `merge`-Anweisung das Gehalt mehrfach geändert wird. Das ist ein exotischer Sonderfall, denn normalerweise sind die Daten, die in die Zieltabelle integriert werden sollen, konstant und ändern sich nicht. In unserem Beispiel habe ich durch die zusätzliche Prüfung verhindert, dass beim zweiten Durchlauf noch einmal eine Änderung des Gehalts durchgeführt wird.

```
SQL> merge into hr_employees t
  2  using (<Unterabfrage)) s
 13     on (t.emp_id = s.emp_id)
 14   when matched then update set
 15        t.emp_salary = s.emp_salary
 16        where s.emp_salary between 6000 and 13000
 17   when not matched then insert(
 …
3 Zeilen zusammengeführt.

-- Erneute Ausführung
SQL> merge into hr_employees t
  2  using (<Unterabfrage)) s
 13     on (t.emp_id = s.emp_id)
 14   when matched then update set
 15        t.emp_salary = s.emp_salary
 16        where s.emp_salary between 6000 and 13000
```

```
 17   when not matched then insert(
 …
1 Zeile zusammengeführt.
```

Listing 10.15 Konditionales Update in der MERGE-Anweisung

Auch hier hätte ich ein Anwendungsbeispiel aus der Praxis für Sie: Ich musste einmal sicherstellen, dass Stammdaten, die in einer Kopie der Produktionsdatenbank ausgelagert und dort für eine neue Version vorbereitet wurden, wieder in die Produktion zurückkopiert wurden. Sinnigerweise wurde aber nicht unterbunden, dass in dieser Migrationszeit Änderungen an den Stammdaten der Produktionsdatenbank durchgeführt wurden, daher konnten Stammdaten der Kopie Änderungen der Produktion überspielen, die nach der Auslagerung der Daten vorgenommen wurden. Glücklicherweise hatte jede Stammdatentabelle ein Set von Spalten, in denen notiert wurde, wann die letzte Änderung der Daten durchgeführt wurde. Basierend auf dieser Information konnte ich eine `merge`-Anweisung so aufbauen, dass über die Schlüsselbeziehung zunächst einmal nur die Daten betrachtet wurden, die in der Kopie der Produktion geändert oder neu hinzugefügt wurden. Zu einer Aktualisierung von Daten, die auch in der Produktion vorhanden waren, kam es aber nur, wenn die Kopie aktuellere Daten hatte als die Produktion, ansonsten hatte die Produktion Vorrang.

MERGE-Anweisung mit nur einem Zweig

Die nächste Variation zum Thema besteht darin, dass Sie einen der beiden Zweige weglassen. Dadurch entsteht eine Anweisung, die zwar auch mit herkömmlichen Mitteln hätte geschrieben werden können, aber einfacher zu formulieren und zu verstehen ist. Finden Sie nicht? Sehen Sie sich in Listing 10.16 diese beiden Varianten im Vergleich an. Wir haben das Ziel, eine Anweisung zu schreiben, die lediglich fehlende Zeilen einfügen soll. Wir benötigen in der ersten Anweisung eine `insert`-Anweisung mit einem Anti-Join, also eine Unterabfrage mit `not in`. In einer `merge`-Anweisung ergibt sich dies ganz einfach über den Zweig `when not matched`, ohne weitere Unterabfrage.

```
SQL> insert into hr_departments(dep_id, dep_name, dep_loc_id)
  2  select 280 dep_id, 'Research' dep_name, 1700 dep_loc_id
  3   where 280 not in (
  4         select dep_id
  5           from hr_departments);
1 Zeile wurde erstellt.

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

```
SQL> -- Zum Vergleich als MERGE-Anweisung
SQL> merge into hr_departments t
  2  using (select 280 dep_id, 'Research' dep_name, 1700 dep_loc_id) s
  3     on (t.dep_id = s.dep_id)
  4   when not matched then insert (dep_id, dep_name, dep_loc_id)
  5        values (s.dep_id, s.dep_name, s.dep_loc_id);
1 Zeile integriert.
```

Listing 10.16 Vergleich INSERT mit Anti-Join gegen MERGE

Voraussetzung ist, dass Sie sich an die Syntax dieser Anweisung gewöhnt haben, aber dann ist die Gliederung doch recht leicht verständlich und gut selbstdokumentierend. Ein weiteres Anwendungsgebiet solcher `merge`-Anweisungen mit einem `insert`-Zweig sind Installationsskripte für Software. Hier werden die Daten gern über eine Unterabfrage, basierend auf einer `values`-Tabellenfunktion, vorbereitet und in einer `merge`-Anweisung mit `insert`-Zweig in die Tabellen integriert. Gerade Installationsskripte haben eine besondere Verpflichtung, wiederaufsetzbar zu sein, wie man das nennt, also mehrfach ausgeführt werden zu können, ohne Daten doppelt einzufügen oder Constraint-Verletzungen zu provozieren. Das ist mit einer `merge`-Anweisung sichergestellt.

Gerade in Anwendungen, in denen viele Tabellen ineinander *integriert* werden, wie Oracle das so schön nennt, kann es vorteilhaft sein, immer wieder den gleichen Weg über die `merge`-Anweisung zu nehmen. Dadurch wird schneller klar, was die Anweisungen eigentlich tun. Zudem haben Sie die Möglichkeit, jederzeit einen der fehlenden Zweige hinzuzufügen oder zu löschen. Diese Flexibilität ist insbesondere während der Entwicklung nicht schlecht.

Es gibt aber noch einen gewichtigen weiteren Grund, `merge`-Anweisungen mit einem Zweig zu verwenden, und zwar im Umfeld von `update`-Anweisungen. Sie erinnern sich noch an das Problem mit dem lange laufenden Update aus Abschnitt 10.2.1, »Allgemeine Syntax«? Lassen Sie uns einmal nachsehen, wie die Lösung mit der `merge`-Anweisung gelingt. Die Anweisung zeigt Listing 10.17, ich zeige auch gleich den Ausführungsplan. Ich kenne Kollegen, die `update`-Anweisungen nur dann verwenden, wenn Sie trivial einfach zu schreiben und alle Daten bereits vorhanden sind. Sobald die Komplexität ein gewisses (niedriges) Maß übersteigt, wechseln sie auf die `merge`-Anweisung mit nur einem `update`-Zweig und profitieren von einfacherer Schreibweise und besserer Performanz. Wer weiß, vielleicht gehören Sie ja auch bald zu dieser Gruppe?

```
SQL> merge into prod_table t
  2  using (select prod_id,
  3                row_number() over (order by prod_id) rn
  4           from prod_table) s
```

```
  5     on (t.prod_id = s.prod_id)
  6   when matched then update set
  7        t.prod_total_id = s.rn;
72000 Zeilen integriert.
Abgelaufen: 00:00:00.45

Ausführungsplan
----------------------------------------------------------------------
| Id | Operation              | Name       |Cost (%CPU)| Time
----------------------------------------------------------------------
|  0 | MERGE STATEMENT        |            | 985   (1) | 00:00:12
|  1 |  MERGE                 | PROD_TABLE|           |
|  2 |   VIEW                 |            |           |
|* 3 |    HASH JOIN           |            | 985   (1) | 00:00:12
|  4 |     VIEW               |            | 413   (2) | 00:00:05
|  5 |      WINDOW SORT       |            | 413   (2) | 00:00:05
|  6 |       TABLE ACCESS FULL| PROD_TABLE| 189   (1) | 00:00:03
|  7 |     TABLE ACCESS FULL  | PROD_TABLE| 189   (1) | 00:00:03
----------------------------------------------------------------------
```

Listing 10.17 Warum nicht gleich so? Hier die schnelle Lösung für das UPDATE-Problem aus Abschnitt 10.2.1

Optimizer Hint IGNORE_ROW_ON_DUPKEY_INDEX

Eine Anmerkung zur Verwendung der `merge`-Anweisung für konditionale Inserts habe ich noch. Der Grund, nur einen `when not matched`-Zweig zu verwenden, war es ja, Primärschlüsselverletzungen auszuweichen. Hierfür gibt es einen sehr seltsamen, zweiten Weg, den ich nicht empfehle, aber nicht verschweigen möchte. In SQL-Anweisungen können speziell formatierte Kommentare eingefügt werden, die von der Datenbank während der Analyse berücksichtigt werden. Diese speziellen Kommentare werden *Optimizer Hints* genannt, denn der Optimizer ist die Programmkomponente, die den Ausführungsplan berechnet, und diesem Programm werden Hinweise zur Optimierung übergeben. Es führt zu weit, sinnvolle Anwendungen hierfür aufzuzeigen (ich werde in seltenen Fällen erläutern, wann dies sinnvoll ist), doch basiert die angesprochene Option auf einem solchen Hint. Er heißt `IGNORE_ROW_ON_DUPKEY_INDEX` (die Schreibweise ist, wie üblich, egal) und wird als Kommentar unmittelbar hinter der `insert`-Anweisung eingefügt. Wichtig ist, dass unmittelbar hinter der einführenden Kommentaranweisung ein einzelnes Pluszeichen eingefügt wird, daran erkennt der Optimizer einen Hint.

Im Gegensatz zu üblichen Hints, die vom Optimizer berücksichtigt werden können oder auch nicht, wird dieser spezielle Hint zwangsweise verwendet und verändert das Verhalten der `insert`-Anweisung, denn damit werden keine Fehler geworfen,

wenn ein Verstoß gegen einen unique-Constraint ermittelt wird. Wie dieser Hint eingesetzt wird, zeigt Listing 10.18. Sie erkennen, dass die Verwendung die Angabe der Tabelle und des Index erfordert, der ignoriert werden soll. Bei einigem Nachdenken sollte dies verwundern, denn die Tabelle hat einen zweiten unique-Index auf die Spalte EMP_EMAIL, der allerdings nicht angegeben werden muss, obwohl auch dieser verletzt würde. Offensichtlich reicht es, einen Index zu bestimmen, um das Einfügen der Zeile und damit die Verletzung weiterer unique-Indizes zu verhindern.

```
SQL> insert into hr_employees
  2  select *
  3    from hr_employees
  4   where emp_dep_id = 60;
insert into hr_employees
*
FEHLER in Zeile 1:
ORA-00001: Unique Constraint (SQL_BUCH.HR_EMPLOYEES_PK) für Tabelle
SQL_BUCH.HR_EMPLOYEES mit Spalten (EMP_ID) verletzt
ORA-03301: (ORA-00001-Details) Zeile mit Spaltenwerten (EMP_ID:103) ist
bereits vorhanden
Hilfe: https://docs.oracle.com/error-help/db/ora-00001/

SQL> insert /*+ IGNORE_ROW_ON_DUPKEY_INDEX
               (hr_employees, hr_employees_pk)*/
  2    into hr_employees
  3  select *
  4    from hr_employees
  5   where emp_dep_id = 60;
0 Zeilen erstellt.
```

Listing 10.18 Verwendung des Hints IGNORE_ROW_ON_DUPKEY_INDEX

Ich empfehle diesen Weg nicht, weil es mir seltsam vorkommt, dass ein Optimizer Hint, der normalerweise der internen Optimierung der Erzeugung des Ausführungsplans dient, dazu verwendet wird, das Verhalten der Anweisung zu ändern. Da zudem eine »offizielle« Strategie zur Vermeidung solcher Probleme mittels merge-Anweisung existiert, bevorzuge ich diese, da sie klarer beschreibt, was ich eigentlich vorhabe.

MERGE-Anweisung im Umfeld von Datenbanklinks

Eine andere, naheliegende und gerade in diesem Zusammenhang häufig verwendete Variation ist, die Quell- oder Zieltabelle über einen Datenbanklink aus einer anderen Datenbank anzukoppeln.

Ein *Datenbanklink* ist eine Verbindung einer Datenbank mit einer anderen Datenbank. Durch eine solche Verbindung können Sie in SQL die Daten einer anderen Datenbank ansprechen. Dies erfolgt so, dass dem Tabellennamen der Ferndatenbank ein @-Zeichen und der Name des Datenbanklinks angehängt werden. Hätten wir also einen Datenbanklink zur Produktionsdatenbank, den wir PROD nennen, könnten Daten aus der Tabelle HR_EMPLOYEES der Produktionsdatenbank über folgende select-Abfrage gelesen werden:

```
select *
  from hr_amployees@prod
```

Listing 10.19 Verwendung eines Datenbanklinks

Dadurch erreichen Sie, dass Sie Daten aus Subsystemen mit eigener Datenbank schnell und unkompliziert in das zentrale System integrieren können. Zwar stellt Oracle für diese Aufgaben eine ganze Reihe weiterer und mächtigere Optionen zur Verfügung, aber nicht immer benötigen Sie die großen Kanonen, um Ihre lokalen Spatzen zu erlegen. In diesen Fällen wäre eine merge-Anweisung ein eleganter und guter Weg.

Sollten Sie sich für Datenbanklinks näher interessieren, besprechen Sie dies bitte mit Ihrem Datenbankadministrator, oder sehen Sie die Details in der Online-Dokumentation nach, denn die mit diesem Konstrukt verbundenen Optionen kann ich ohne einen weiten Ausflug in die Datenbankadministration nicht recht erklären.

10.5 Exkurs: Flashback

Im Zusammenhang mit den Befehlen zur Manipulation von Daten möchte ich Ihnen eine Abfragestrategie vorstellen, mit der gelöschte oder überschriebene Daten einer Tabelle noch im Nachhinein abgefragt werden können. Es handelt sich um die sogenannte *Flashback-Abfrage*. Alle Änderungen an Daten werden innerhalb der Datenbank protokolliert und in eine externe Datei geschrieben. An diesen Informationsfluss hat Oracle eine Funktionalität gehängt, die auf spezielle Weise die Änderungen der Datenbank für die Datenbank verfügbar macht. Im Endeffekt hat die Datenbank damit die Möglichkeit, für eine kurze Zeit zu ermitteln, wie die Daten vor einer Änderung ausgesehen haben. Natürlich geht so etwas nicht unbeschränkt lange, und die Größe der Datei entscheidet über die Dauer des Gedächtnisses. Details interessieren hier nicht, doch ist es im Regelfall möglich, einen Zeitraum von mindestens 15 Minuten zu schützen. Auf diese Daten können Sie in select-Anweisungen zugreifen, indem Sie die Klausel as of verwenden:

```
SQL> select emp_last_name, emp_job_id, emp_salary
  2    from hr_employees
  3   where emp_id = 100;

EMP_LAST_NAME EMP_JOB_ID EMP_SALARY
------------- ---------- ----------
King          AD_PRES         24000

SQL> update hr_employees
  2     set emp_last_name = 'König'
  3   where emp_id = 100;
1 Zeile wurde aktualisiert.

SQL> commit;
Transaktion mit COMMIT abgeschlossen.

SQL> select emp_last_name, emp_job_id, emp_salary
  2    from hr_employees
  3   where emp_id = 100;

EMP_LAST_NAME EMP_JOB_ID EMP_SALARY
------------- ---------- ----------
König         AD_PRES         24000

SQL> select emp_last_name, emp_job_id, emp_salary
  2    from hr_employees
  3   as of timestamp systimestamp - interval '1' minute
  4   where emp_id = 100;

EMP_LAST_NAME EMP_JOB_ID EMP_SALARY
------------- ---------- ----------
King          AD_PRES         24000
```

Listing 10.20 Anwendung der AS OF-Klausel

Die Klausel wird unmittelbar nach der `from`-Klausel verwendet und erwartet entweder einen Zeitstempel oder eine spezielle, Oracle-interne Transaktionsnummer, die *SCN* (*System Change Number*) , die wir nicht hier, sondern in Kapitel 14, »Aufbau einer Oracle-Datenbank«, näher erläutern. Die Klausel sorgt dafür, dass die Daten ihren historischen Stand von vor 1 Minute zeigen, obwohl aktuell der Datenbestand bereits geändert und festgeschrieben wurde. Mittels dieser Abfragetechnik können zwar auch im normalen Abfragebetrieb historische Abfragen gestellt werden, häufiger ist jedoch die Verwendung nach einem ungewollten (und festgeschriebenen) Löschvor-

gang, denn dann können die Daten einfach über die Anweisung `insert … as select from emp as of …` wiederhergestellt werden. Syntaktisch erkennen Sie wiederum die Verwendung der Intervalle über das Literal `interval`.

Im Übrigen sollten Sie für Ihre Datenbank testen, ob Flashback wirklich zur Verfügung steht, weil der Administrator diese Option einrichten muss (oder andersherum: Sie ist im Regelfall eingerichtet, kann aber deaktiviert werden). Zudem steuert der Administrator die Länge des »Gedächtnisses« sowie die Zuverlässigkeit, mit der Sie sich auf diese Dauer verlassen können. Details erfahren Sie im Gespräch mit Ihrem Administrator.

10.6 Sequenzen und Trigger

Zur Abrundung des Themas möchte ich gern zwei Themen mit Ihnen besprechen: die Verwendung von Sequenzen und Datenbank-Trigger.

10.6.1 Sequenzen

Sequenzen werden verwendet, um neue Primärschlüsselwerte für eine Spalte zu ermitteln. Andere Datenbanken kennen für diesen Zweck (ausschließlich) eine Autowertspalte, die es bei Oracle alternativ gibt. Der Nachteil einer Autowertspalte besteht darin, dass er eindeutige Werte immer nur für diese Spalte garantieren kann. Da und dort kommt es aber vor, dass wir eindeutige Spaltenwerte spalten- oder gar tabellenübergreifend benötigen. Dies ist nur mit einer externen Quelle für solche Schlüsselwerte möglich. Zum anderen sind die externen Sequenzen leichter an spezielle Anforderungen anpassbar. So können leicht Schrittweiten eingestellt werden (zum Beispiel sollen Zahlen in Inkrementen von 10 erzeugt werden) oder Cachegrößen vorgegeben werden, die dafür sorgen, dass in einem Rutsch gleich 20.000 oder mehr Primärschüssel gesichert aus dem Arbeitsspeicher erstellt werden können, um die Performanz zu erhöhen. Schließlich können diese Sequenzen zyklisch angelegt werden, das heißt, sie zählen bis zu einem Maximalwert, um dann wieder von vorn zu beginnen.

Sequenzen und Autowertspalten sind nicht die einzigen Mechanismen, die Primärschlüsselwerte erzeugen, alternativ können Sie eine Sequenz durch eine Tabelle nachbilden. Dies gibt es manchmal, weil nicht alle Datenbanken solche Sequenzen haben und daher Anwendungen, die gegen mehrere Datenbanken einsetzbar sein sollen, nach Auswegen aus diesem Dilemma suchen. Meistens wird eine Tabelle eingerichtet, in der für jede Tabelle, die Primärschlüsselwerte verwalten soll, eine Zeile eingefügt wird. Als Primärschlüssel dieser »Primärschlüsseltabelle« wird dann der Name der Tabelle verwendet, für die der Primärschlüssel verwaltet werden soll. In einer weiteren Spalte wird dann der aktuelle Primärschlüsselwert gespeichert.

Möchte ein Benutzer eine neue Zeile einfügen, muss er zunächst zu dieser zentralen Tabelle, sich dort den aktuell gültigen Schlüssel ansehen, die Zeile der entsprechenden Tabelle um `1` hochzählen und den neuen Wert dort ablegen. Dieser Wert kann anschließend als Primärschlüssel verwendet werden. Der Nachteil dieser Methode: Zu dieser Tabelle wollen alle Benutzer. Schlimmer noch: Alle Benutzer möchten gern in einige wenige Zeilen dieser Tabelle schreiben, nämlich in die Zeilen, die die Primärschlüssel der am häufigsten geänderten Tabellen enthalten. Damit wird diese eine Tabelle zum Flaschenhals der gesamten Anwendung, denn nur, wenn wir dort einen Schlüssel »ergattert« haben, können wir eine Zeile in die Zieltabelle einfügen. Da Datenbanken nur einem Benutzer zur gleichen Zeit erlauben, eine Zeile zu ändern, müssen die anderen Benutzer so lange warten, bis die gewünschte Zeile frei ist.

Es gibt zwar verschiedene Möglichkeiten, dieses Grundproblem zu mildern, aber keine, es zu lösen, außer: Verwenden Sie eine Sequenz! Die Details dieser Diskussion ersparen wir uns und überlassen das meinem Buch »Oracle PL/SQL«, das ebenfalls beim Rheinwerk Verlag erschienen ist, aber diese grundsätzliche Aussage kann ich guten Gewissens so stehen lassen. Wie aber macht Oracle das mit der Verwaltung der Primärschlüssel in Sequenzen? Es mag Sie überraschen, aber Oracle macht es genauso, wie ich es oben als handgemachte Lösung skizziert habe! Auch bei Oracle wird eine zentrale Tabelle geführt, die alle Sequenzen der Datenbank und den jeweils aktuell gültigen Schlüssel enthält. Toll, werden Sie denken, worin soll denn dann der Vorteil liegen? Nun, der Vorteil liegt im Caching. Normalerweise ist die Sequenz so eingestellt, dass die Tabelle nicht den nächsten Sequenzwert, sondern zum Beispiel den aktuellen Wert + 20 dort hinterlegt. Das RDBMS kann dann alle Werte bis zu diesem Wert aus dem Arbeitsspeicher heraus erzeugen, da sichergestellt ist, dass eine andere Datenbankinstanz sich ebenfalls einen Schlüsselbereich bei der zentralen Tabelle besorgt und diesen Bereich verwendet. Das RDBMS kann diese Werte sehr schnell bereitstellen, und zwar, das ist das Besondere, an *alle* Benutzer, die einen Wert dieser Sequenz erfragen. Programmierer könnten so einen Cache auch selbst programmieren, aber es ist sehr schwer (wenn auch nicht unmöglich), diesen Cache mit anderen Benutzern zu teilen. Genau das aber tut der Cache der Sequenz. Zudem kann die Cachegröße pro Sequenz unterschiedlich eingestellt werden. Haben Sie eine Tabelle mit sehr großer Transaktionslast (sehr viele Benutzer fügen gleichzeitig Werte in die Tabelle ein), stellen Sie den Cache entsprechend hoch. Ist die Transaktionslast dagegen klein, stellen Sie den Cache herunter. Durch die große Anzahl reservierter Primärschlüssel muss die Datenbank nun viel seltener zur internen Tabelle, um einen neuen Nummernkreis zu reservieren. Das spart sehr viel Zeit und reduziert die Wahrscheinlichkeit eines Zugriffskonflikts auf die gleiche Zeile der Sequenztabelle.

Tatsächlich können Sie sich die Tabelle, die die Sequenzen verwaltet, ansehen. Sie gehört dem Benutzer `SYS` und heißt `SEQ$`. Hier ist eine `select`-Abfrage gegen diese Sequenz (die Abfrage muss als Benutzer `SYSTEM` oder `SYS` erfolgen):

```
SQL> select obj#, increment$, minvalue, maxvalue,
  2         cycle#, cache, highwater
  3    from sys.seq$
  4   where rownum < 11;

OBJ# INCR MINVALUE   MAXVALUE CYCLE# CACHE HIGHWATER
---- ---- -------- ---------- ------ ----- ---------
  97    1        0 1,0000E+28      0    10         1
 188    1        1 1,0000E+20      1    20         1
 285    1        0 1,0000E+28      0     0         2
 286    1        1  999999999      1    20        23
 290    1        1    4294967      1 10000    130001
 309    1        0 1,0000E+28      0    10         1
 358    1        1 1,0000E+28      0    20     36675
 359    1        1 1,0000E+28      0    20      1195
 361    1        1 2000000000      1 10000    250004
 417    1        1 1,0000E+28      0    20         1
10 Zeilen ausgewählt.
```

Listing 10.21 Abfrage gegen die Tabelle SYS.SEQ$

Typisch für Tabellen des Benutzers SYS ist, dass Sie fast nur Schlüsselwerte sehen, kaum einmal Klartextbezeichnungen. Namen sind jedoch Schall und Rauch, die Objektnummer steht für irgendeine Sequenz. In der Tabelle sehen Sie einige der Optionen, die ich bereits angesprochen hatte; ganz rechts, in der Spalte HIGHWATER, steht der aktuelle Wert der Sequenz (also, um etwas genauer zu sein, der Wert, der derzeit als Maximalwert reserviert wurde, was nicht notwendigerweise dem Wert entspricht, den diese Sequenz als nächsten liefern wird). Nun noch die Frage, wie eine solche Sequenz angelegt und benutzt wird. Listing 10.22 zeigt eine einfache Sequenz, die ab dem Startwert 100 Zahlen liefern soll. Die Sequenz soll nur angelegt werden, falls sie nicht bereits existiert (das ist eine Neuerung in Version 23ai und funktioniert in früheren Datenbankversionen nicht).

```
SQL> create sequence if not exists hr_employees_seq
  2   start with 100
  3   increment by 1
  4   cache 20
  5   nocycle;
Sequenz wurde erstellt.

SQL> select hr_employees_seq.nextval;
   NEXTVAL
----------
       100
```

Listing 10.22 Erzeugen und Verwenden einer Sequenz

Die Sequenz bietet eine Pseudospalte `NEXTVAL` an (neben einer Pseudospalte `CURRVAL`, die den aktuellen Wert enthält), die den nächsten – wahrscheinlich bereits im Cache befindlichen – Wert liefert. Die Pseudospalte wird angesprochen, indem sie dem Namen der Sequenz über einen Punkt angehängt wird, ebenso wie wir eine Tabellenspalte über den Namen der Tabelle und einen Punkt ansprechen.

Nun könnten Sie sich (oder schlimmer ... mir) die Frage stellen, ob wir eine Sequenz pro Tabelle einrichten sollten oder eine Sequenz für alle Tabellen. Alles kann, nichts muss, könnte man hier sagen: Sie haben die Wahl, die Anzahl der Sequenzen auf die Tabellen aufzuteilen. Sie können so viele oder so wenige Sequenzen einrichten, wie Sie mögen. Häufiger sehe ich, dass pro Tabelle eine Sequenz angelegt wird, als dass eine Sequenz für mehrere Tabellen verwendet wird. Allerdings könnte man als Zwischenlösung eine Sequenz für Stammdatentabellen und je eine Sequenz pro Bewegungsdatentabelle einrichten. Eine Empfehlung kann man da nicht so recht geben. Der einzige harte Grund für die Verwendung einer Sequenz für mehrere Tabellen ist die Tatsache, dass Sie aus irgendwelchen Gründen in den Tabellen übergreifend eindeutige Schlüssel benötigen.

Als Einsatzgrenze für Sequenzen können wir darüber hinaus noch festhalten, dass diese nicht geeignet sind, wenn Sie eine *ununterbrochene Folge* von Primärschlüsselwerten benötigen, wie das zum Beispiel für Rechnungsnummern erforderlich ist. Der Grund: Hat eine Sequenz einmal Werte geliefert, werden diese Werte nicht mehr verwendet. Dadurch können Werte verloren gehen, »Löcher« entstehen, und zwar in folgenden beispielhaften Szenarien:

- wenn die Sequenz einen Cache eingestellt hat und die Datenbank neu gestartet wird (In diesem Fall sind die im Cache befindlichen Werte verloren.)
- wenn Sie eine `rollback`-Anweisung geben (Auch in diesem Fall wird die verwendete Zahl nicht an die Sequenz zurückgeliefert.)
- falls eine `insert`-Anweisung aus irgendwelchen Gründen fehlschlägt Auch in diesem Fall wird die aus der Sequenz gewonnene Nummer verworfen und neu angefordert. Das ist beinahe das Gleiche wie der vorherige Fall mit dem `rollback`, jedoch mit einem kleinen Unterschied: Misslingt diese eine `insert`-Anweisung, wird diese Anweisung abgelehnt, nicht aber die gesamte Transaktion. Wird die `insert`-Anweisung korrigiert und erneut eingegeben, kann anschließend die gesamte Transaktion bestätigt werden, der von der fehlerhaften `insert`-Anweisung verwendete Sequenzwert bleibt jedoch verworfen.

Sie mögen dieses Verhalten von Oracle als nicht sehr intuitiv ansehen oder der Meinung sein, Oracle solle sich hier »mehr Mühe geben«, Löcher in den Primärschlüsselwerten zu verhindern, doch spielen diese Löcher keine Rolle: Da ein Primärschlüssel ja definitionsgemäß keinerlei Geschäftsinformation trägt (im Gegensatz zu einer Rechnungsnummer, weil an sie die Forderung der Lückenlosigkeit gestellt wird und

sie dadurch die Zusicherung gibt, dass keine Rechnung fehlt), spielen fehlende Werte keine Rolle. Möchten Sie dennoch eine lückenlose Rechnungsnummer erzeugen, müssen Sie dies durch explizite Programmierung mit anderen Mechanismen erzeugen, dafür sind Sequenzen einfach nicht da. Es versteht sich im Übrigen von selbst, dass Sequenzen nur für numerische Primärschlüssel verwendet werden können, nicht aber für alphanumerische Schlüssel.

Wie die Autowertspalten zu benutzen sind, erkläre ich in Kapitel 12, »Tabellen erstellen«, denn in der Benutzung sind sie insofern unspektakulär, als sie gerade nicht benutzt werden dürfen, sondern in jedem Fall einen entsprechenden Primärschlüsselwert automatisch einfügen. Da Oracle im Hintergrund hierfür eine Sequenz einrichtet, stehen im Grundsatz die gleichen Möglichkeiten und Einschränkungen zur Verfügung, die wir bezüglich der Sequenzen angesprochen haben.

10.6.2 Datenbank-Trigger

Das Thema Datenbank-Trigger wird im PL/SQL-Buch ausführlich besprochen, weil es sich um ein Programmierthema handelt. Zudem werde ich Ihnen einige wenige Beispiele in Kapitel 12, »Tabellen erstellen«, zeigen. Hier sollen nur die grundlegenden Dinge so weit besprochen werden, dass Sie ein Gefühl hierfür bekommen und sich nicht wundern, wenn auf einmal ganz andere Werte in den Tabellen stehen, als Sie einfügen wollten, oder wie von Zauberhand Werte auftauchen, die Sie nicht eingefügt haben.

Ein Trigger ist ein kleines Programm, das immer dann ausgeführt wird, wenn ein definiertes Ereignis stattfindet. Ein solches Ereignis ist typischerweise das Einfügen, Ändern oder Löschen von Daten einer bestimmten Tabelle. Ein solcher Trigger ist stets einer Tabelle zugeordnet und wird ausgeführt, wenn eine Datenänderung auf diese Tabelle durchgeführt wird.

Wozu dienen solche Trigger? Normalerweise werden mit Triggern zwei eng beieinanderliegende Aufgabenbereiche bearbeitet: Es werden Datenkonsistenzregeln und Zugriffsregeln durchgesetzt. Damit ist Folgendes gemeint:

- Datenkonsistenzregeln sind Regeln, die festlegen, dass gewisse Anforderungen an die Daten gestellt werden, die durch einfache Constraints oder Vorgabewerte nicht umgesetzt werden können. So könnte zum Beispiel geregelt sein, dass in einer Spalte Werte nur in Großbuchstaben gespeichert werden dürfen oder dass ein einmal eingefügtes Anlagedatum eines Datensatzes nicht mehr im Nachhinein geändert werden kann.
- Zugriffsregeln dienen dazu, Anforderungen an die Datensicherheit durchzusetzen. Beispiele könnten sein, dass eine Änderung von Daten nur in bestimmten Zeiträumen möglich sein darf oder dass Datenänderungen stets in anderen Tabel-

len protokolliert werden. Es könnte auch geregelt werden, dass Änderungen an Daten nur ermöglicht werden, wenn sich der Anwender aus dem lokalen Netzwerk angemeldet hat und nicht etwa über eine Internetverbindung.

Der Vorteil dieser Trigger ist, dass sie nur dadurch umgangen werden können, dass sie deaktiviert oder gelöscht werden. Sind sie aktiv, kann auch ein Datenbankadministrator die in den Triggern definierten Regeln nicht umgehen. Der Nachteil von Triggern liegt darin, dass, insbesondere wenn sehr weitgehende Geschäftsregeln in diesen Triggern umgesetzt wurden, die Übersicht darüber, wo und was geschieht, sehr schwierig aufrechterhalten werden kann. Zudem gibt es hinterhältige logische Probleme mit Triggern, die zum Teil nur in speziellen Situationen auftauchen und daher die Fehlersuche erschweren.

Eine – aus meiner Sicht – akzeptable Strategie besteht daher darin, lediglich sehr datennahe Arbeiten in Triggern erledigen zu lassen. Dazu gehören die bereits angesprochenen Beispiele, wie etwa Großschreibung von Spaltenwerten, Vergabe von Standarddaten über eine Spalteneinstellung hinaus, Schutz vor nachträglicher Änderung von Daten, Verwaltung von Sequenzen und das Dokumentieren von Datenänderungen in separaten Tabellen, das sogenannte *Logging*. Noch besser ist es, Trigger vollständig zu umgehen. Dies gelingt allerdings nur durch einen zusätzlichen Aufwand, denn hierfür müssen Methoden programmiert werden, die das Schreiben in die Tabellen übernehmen und als Teil ihrer Arbeit die bisherige Arbeit der Trigger übernehmen. Wie das genau geht, ist aber ein Programmierthema und findet sich daher im PL/SQL-Buch.

Halten wir fest, dass es möglich ist, auf Tabellen Trigger einzurichten, die das Einfügen, Ändern und Löschen von Daten beeinflussen können. Zum Teil werden Daten allerdings nicht nur durch Trigger, sondern durch Standardwerte für Spalten in die Tabellen eingefügt. Im Gegensatz zu diesen Standardwerten für einzelne Spalten können Trigger allerdings das Einfügen anderer Werte als die vorgesehenen Standardwerte verhindern. Daher stellen Trigger ein mächtiges Werkzeug der Datenbank dar, um Zusicherungen über die Daten zu machen.

Allgemein ist bei Oracle ein Bestreben spürbar, die Erstellung von Triggern unnötig zu machen. Viele der Funktionen, die bislang noch die Erstellung eines Triggers zumindest nahelegen, wurden von Datenbankversion zu Datenbankversion zunehmend überflüssig, da die entsprechenden Funktionen direkt in SQL integriert wurden. In Version 23ai ist dies noch einmal weitergetrieben worden, weil ab dieser Version Standardwerte auch dann eingefügt werden, wenn eine Änderung der Zeile ansonsten `null`-Werte erzeugen würde. Ein Hauptgrund hierfür liegt darin, dass ein Trigger, der die Werte einer Zeile überwacht, immer einen ressourcenintensiven Umweg zur PL/SQL-Seite der Datenbank unternehmen muss. Gerade bei Tabellen mit hoher Transaktionslast kann dies Schreibprozesse so verlangsamen, dass Alter-

nativen erforderlich werden. Gelingt es, die Standardaufgaben eines Triggers in SQL zu integrieren, ist dieser Umgebungswechsel nicht erforderlich.

10.7 Ihr Sicherheitsnetz – die Transaktion

Vielleicht haben Sie den Begriff der *Transaktion* im Zusammenhang mit Datenbanken schon einmal gehört. Ich erlebe allerdings häufiger, dass die Vorstellung dessen, was eine Transaktion ist, eher nebulös ist. Das ist schade, denn Transaktionen sind für die Arbeit in Datenbanken nicht nur unverzichtbar, sondern geben Ihnen die Sicherheit, Datenänderungen jederzeit und rückstandsfrei widerrufen zu können. Zudem schützen Transaktionen Ihre Änderungen vor parallel arbeitenden Benutzern der Datenbank, die Ihre Änderungen erst dann sehen können, wenn Sie dies möchten. Gründe genug also, sich näher mit diesem Begriff auseinanderzusetzen.

10.7.1 Was ist eine Transaktion?

Zunächst einmal bezeichnet eine Datenbanktransaktion (kurz eine Transaktion) eine Folge von Datenmanipulationsanweisungen, die entweder als Ganzes gelingen oder als Ganzes zurückgenommen werden soll. Für die Datenbank ist die Transaktion ein zentrales Element, denn nur mit diesem Hilfsmittel ist es der Datenbank möglich, zu erkennen, wann eine Datenänderung, die aus mehreren Einzeländerungen besteht, logisch beginnt und endet. Und nur in Kenntnis dieser logischen Grenzen kann die Datenbank einen Zeitpunkt festlegen, zu dem sie zurückgehen kann, falls etwas schiefgeht, denn sie nimmt an, dass vor Beginn und nach Ende einer Transaktion die Daten jeweils konsistent sind und dauerhaft gespeichert werden sollen. So wichtig dieser Begriff ist, so selten müssen Sie mit diesem Thema aktiv umgehen, wenn man einmal davon absieht, dass Sie das Ende einer logischen Datenänderung durch die Anweisung `commit` bestätigen oder durch die Anweisung `rollback` zurücknehmen müssen. Das ist bei anderen Datenbanken anders, weil dort oftmals der Beginn einer Transaktion explizit angefordert werden muss. So etwas benötigen Sie bei Oracle nicht, Sie arbeiten grundsätzlich immer unter Transaktionsschutz. Dieser Schutz beginnt mit der ersten Anweisung, die Daten ändert, und bleibt als Schutz im Hintergrund so lange bestehen, bis Sie eine der oben genannten Anweisungen zum Beenden der Transaktion absetzen (oder die Datenbank das für Sie tut, was, je nach Anweisung, auch einmal passieren kann).

In Anwendungsprogrammen sind diese Gruppen von Anweisungen typischerweise sehr schnell abgearbeitet, eine Transaktion daher kurz. Aber wenn Sie an einer Datenbank im Dialogbetrieb arbeiten, etwa über den SQL Developer, können Transaktionen zeitlich sehr lang sein. Zudem gibt es Anwendungsfälle, die extrem viele Änderungen in einer riesigen Transaktion zusammenfassen. All dies ist möglich und wird von Oracle unterstützt.

In diesem Kapitel möchte ich den Begriff der Transaktion aus logischer Sicht betrachten. In diesem Licht stellt sich eine Transaktion als eine Klammer um mehrere Anweisungen dar, die Daten in der Datenbank ändern. Die technischen Implikationen behalte ich mir dann für Abschnitt 14.1.2, »Änderung von Daten, Transaktion«, vor, wo ich die technischen Hintergründe der Arbeitsweise von Oracle etwas ausleuchten werde.

Wichtig ist zu verstehen, dass eine Transaktionsklammer immer nur im Rahmen unserer aktuellen Session, also der logischen Verbindung, die wir zur Datenbank haben, definiert ist. Eine Transaktion betrifft also stets nur *unsere* Änderungen an der Datenbank. Andere Benutzer können und werden parallel weitere Transaktionen in ihren Sessions haben, allerdings jeweils immer nur eine zur gleichen Zeit, niemals zwei gleichzeitig (hiervon gibt es eine etwas exotische Ausnahme der autonomen Transaktion, die wir bei der `log errors`-Klausel im nächsten Abschnitt kennenlernen werden).

Eine Transaktion, haben wir gesagt, beginnt mit der ersten Anweisung, die einen Datenbestand ändert, etwa einer `insert`-Anweisung. Durch die Transaktion geschützt, sind diese Daten aus anderen Sessions nicht zu sehen, sie sind »under construction«, bis wir die Daten freigeben. Das Beispiel der `insert`-Anweisung ist insofern unproblematisch, als diese Daten nicht bereits von anderen Benutzern zur gleichen Zeit geändert werden können, denn vor der Anweisung existierten die Daten ja nicht. Dies ist anders, wenn Sie Daten mit der `update`-Anweisung ändern oder mit der `delete`-Anweisung löschen möchten, denn nun stellt sich die Frage, was passiert, wenn ein anderer Benutzer genau die Daten, die Sie ändern möchten, ebenfalls gerade ändert. Die Datenbank wird solche Daten immer nur einer Transaktion zur gleichen Zeit zuteilen und für die gleichzeitige Änderung durch andere Benutzer sperren. Dabei sperrt die Datenbank immer eine ganze Zeile einer Tabelle, nie nur eine einzelne Zelle innerhalb der Zeile. Andererseits werden durch die Datenbank nie mehr Zeilen als unbedingt nötig gesperrt, so dass eine Zeile einer Tabelle auch dann noch von einem zweiten Benutzer gesperrt werden kann, wenn alle anderen Zeilen bereits von einem anderen Benutzer gesperrt wurden. Dieses Sperrverhalten ist sinnvoll, aber auch problematisch, wenn Sie zum Beispiel versuchen, nach dem Einfügen Ihrer Daten andere Daten zu ändern, die aktuell durch einen Dritten geändert werden. Tatsächlich sind diese Daten gesperrt und können von Ihnen nicht bearbeitet werden. Sie hängen also mitten in Ihrer Arbeit fest und müssen warten, bis die Daten durch den anderen Benutzer freigegeben werden.

Merke

Version 23ai erweitert diesen Sperrmechanismus durch die Möglichkeit, eine Zeile zu »reservieren«. Damit wird ein Mechanismus bezeichnet, der im Rahmen einer Transaktion Spaltenwerte berechnen kann, ohne diese explizit zu sperren. Dies hat für die

Transaktion direkt keine Auswirkungen und wird nicht durch eine spezielle Syntax innerhalb der `update`-Anweisung erzwungen, sondern ist im Gegenteil eine Option der Tabelle, deren Spalte reservierbar sein soll. Daher bespreche ich diese Erweiterung im Zusammenhang mit der Anlage von Tabellen in Kapitel 12.

Sie kennen jetzt die beiden wichtigsten Aspekte von Transaktionen:

- Sie koordinieren die nebenläufige (durch mehrere Benutzer parallel durchgeführte) Bearbeitung der Daten und stellen sicher, dass der ändernde Zugriff auf die Daten in einer logisch einwandfreien Umgebung so erfolgt, dass Sie stets das Gefühl haben, auf der Datenbank allein zu arbeiten.
- Sie sorgen für die Möglichkeit, eine Reihe von Änderungsschritten zurückzunehmen. Daher sind Transaktionen auch dann erforderlich, wenn Sie allein an der Datenbank arbeiten.

Transaktionen sind – neben anderen – die Dinge, die eine Datenbank zu einer Datenbank machen und von einem Dateisystem unterscheiden. Sie stellen nicht nur ein Sicherheitsnetz für Sie dar, das Ihnen erlaubt, Änderungsschritte vollständig zurückzunehmen und damit die Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand zu überführen, sondern sie steuern zudem die Sichtbarkeit Ihrer Änderungen anderen Benutzern gegenüber.

Im Gegensatz zu normalen Anwendungsprogrammen, die eine Widerrufen-Funktion kennen, um einmal gegebene Anweisungen zurückzunehmen und wieder auszuführen, ist bei einer Datenbank die Rücknahme einer einmal bestätigten Änderungsaktion nicht möglich. Das kann auch nicht so sein, denn diese bestätigte Änderung könnte wiederum Anlass für Änderungen der Daten durch andere Benutzer gewesen sein, die durch eine Rücknahme logisch inkonsistent würden. Durch den Einsatz von Transaktionen haben Sie also das Gefühl, allein auf der Datenbank zu arbeiten, obwohl viele Hundert Benutzer parallel Daten bearbeiten, gleichzeitig nutzen Datenbanken diese Transaktionen, um den konsistenten und geschützten Zugriff auf einzelne Datenbankzeilen zu orchestrieren. Ohne Transaktionen haben Sie keinerlei Schutz. Ärgerlich ist, dass einige Anwendungen ohne Transaktionsschutz arbeiten, denn die Datenbanktreiber, also die Programme, die eine Verbindung zwischen Ihrer Anwendung und der Datenbank herstellen, sind standardmäßig so eingestellt, dass ein automatisches `commit` nach jeder Anweisung ausgeführt wird.

Transaktionen werden bei Oracle stets implizit durch die erste Anweisung aus der Gruppe der sogenannten *DML*-Befehle (*Data Manipulation Language*, *Datenmanipulationssprache*) geöffnet und durch `commit` oder `rollback` beendet. Die Befehle dieser Gruppe lauten `insert`, `update`, `delete` und `merge`. Eine Sonderstellung nimmt der Befehl `select` ein, denn er stört die Kreise der Transaktion nicht, beendet sie insbesondere auch nicht. Alle anderen SQL-Anweisungen haben ein implizites `commit` zur

Folge und sind selbst nicht transaktionsgeschützt. Einen `create`-Befehl zum Beispiel kann man sich also vorstellen als eine Folge folgender Befehle:

```
commit;
create …
commit;
```

Daran müssen Sie denken, wenn Sie eine Reihe von DML-Befehlen abgesetzt haben und dann feststellen, dass Ihnen zum Beispiel eine View fehlt. Legen Sie diese bei offener Transaktion an, wird diese mit Anlegen der View beendet und festgeschrieben. Seien also auch Sie beim Arbeiten mit DML-Anweisungen in einem »Sondermodus«: Die Datenbank ist im Editierungsmodus und erwartet von Ihnen eine *informierte* Entscheidung, *wann* und *wie* sie diesen Modus wieder verlassen soll. Ein einfaches Anlegen einer View, die Vergabe eines Leserechts etc. sind keine informierten Entscheidungen, sondern ein Versehen mit möglicherweise schlimmen Konsequenzen!

10.8 Fehlerbehandlung während der Datenmanipulation

In diesem Abschnitt möchte ich Ihnen eine sehr schöne Erweiterung der DML-Anweisungen erläutern. Oft ist es so, dass während einer Datenänderung oder beim Einfügen vieler Daten aus einer Datenquelle einzelne Daten nicht übernommen werden können, vielleicht, weil die Werte für die Zielspalte zu groß sind, weil Constraints verletzt würden oder weil ein Datumsformat nicht erkannt werden konnte. Eine einzelne, fehlerhafte Zeile in vielen Tausend Datensätzen hätte zur Folge, dass die gesamte Anweisung abgelehnt würde. Schlimmer noch: Als Fehlermeldung erhielten Sie in einem solchen Fall lediglich die Nachricht zurück, dass der Wert zu groß für die Spalte sei. Doch welcher Wert war das? Darüber gibt die Fehlermeldung keine Information. Schwerer noch sind Constraint-Fehler zu entdecken, denn wenn die Daten nicht in die Datenbanktabelle geschrieben sind, sehen Sie keine Constraint-Verletzungen. Genau dieses Einfügen wird aber durch den dann entstehenden Fehler verhindert. Die Folge sind lange händische Suchen nach dem Fehler, oftmals auch aufwendige Trial-and-Error-Kaskaden, bis die Daten endlich eingefügt werden können. Da dies oft nicht tolerabel ist, greifen viele Administratoren oder Entwickler zu einer Programmiersprache wie etwa PL/SQL, um die Daten zeilenweise einzufügen und bei Fehlern diese Datensätze in eine Fehlerdatei zu schreiben. Alles nicht nötig: Oracle kennt eine in SQL integrierte und sehr komfortable Möglichkeit, mit solchen Fehlern umzugehen.

10.8.1 Die Klausel LOG ERRORS

Grundlage dieser Möglichkeit ist die Erweiterung der DML-Anweisungen durch die Klausel `log errors`. Die Idee: Es wird im Hintergrund eine Tabelle eingerichtet, in die fehlerhafte Datensätze geschrieben werden. Auf diese Weise können »die Guten ins

Töpfchen, die Schlechten ins Kröpfchen« gelegt werden, ohne den Erfolg der gesamten Ausführung zu gefährden. Haben wir einmal einen solchen Mechanismus, können wir bei dieser Gelegenheit direkt den Fehlergrund, die Referenz auf die Zeile und andere Informationen mitgeben und speichern. Bei dieser Aktion werden Sie zum ersten Mal eine sogenannte *autonome Transaktion* kennenlernen, was keine Transaktion aus der autonomen Szene beschreibt, sondern eine Transaktion, die unabhängig von einer bereits geöffneten Transaktionsklammer eine eingeschachtelte Transaktion bereitstellt. Verwendet werden autonome Transaktionen, um Fehler zu dokumentieren und in der Datenbank zu hinterlegen, selbst wenn die Transaktion, in der der Fehler auftritt, durch `rollback` zurückgenommen wird. Hätten wir keine autonomen Transaktionen, würden diese Fehlereinträge ja ebenfalls bereinigt.

Diese autonomen Transaktionen erstellen also eine Transaktion innerhalb der umgebenden Transaktion und sind die exotische Ausnahme, über die ich im vorherigen Abschnitts gesprochen hatte. Im Umfeld der Klausel `log errors` sind diese autonomen Transaktionen deshalb praktisch, weil im Fehlerfall die eigentliche Transaktion zurückgerollt werden kann, die aufgetretenen Fehler aber dennoch in der Fehlertabelle stehen und die Daten so auf einfache Weise korrigiert und erneut eingelesen werden können. Alternativ kann die Haupttransaktion bestätigt und können die fehlerhaften Daten nachträglich eingefügt werden. Das hängt vom Zusammenhang ab.

Zusätzlichen Komfort verbreitet die Klausel `log errors` zudem noch durch zwei weitere Optionen: Zum einen kann einer DML-Anweisung mit `log errors`-Klausel ein sogenanntes *Tag* (bitte englisch aussprechen – ein *Anhänger* oder *Etikett*) mitgegeben werden, mit dessen Hilfe in der Fehlertabelle die Zeilen identifiziert werden können, die durch den letzten Lauf erzeugt wurden. Zum anderen kann ein `reject limit` (*Abweisungsgrenze*) festgelegt werden. Übersteigt die Zahl der Fehler die Abweisungsgrenze, wird die Anweisung als Ganzes abgelehnt, wie das ohne `log errors`-Klausel (oder bei fehlendem `reject limit`) bereits bei einem Fehler geschehen würde.

Diese Funktion hat ihre Grenzen bei einigen Datentypen, wie `CLOB` oder `BLOB`, aber auch bei objektorientierten Datentypen. Zudem gibt es einige Einschränkungen, die nicht ganz einfach zu erklären sind, weil sie viel Wissen über Optionen voraussetzen, die wir noch nicht besprochen haben. Im Grunde kann man aber sagen: Für die meisten Anwendungen sind diese Möglichkeiten eine sehr elegante, schnelle und einfach zu handhabende Möglichkeit, mit Fehlern beim Import oder der Migration von Daten umzugehen.

Syntaktisch ist die Verwendung einfach: Es wird der DML-Anweisung die Klausel `log errors` als letzte Klausel angefügt. Allerdings ist ein bisschen Vorarbeit nötig, um die Klausel nutzen zu können. Ich möchte mich hier auf die häufigsten Einsatzwege beschränken und nicht jede Option ausloten. Ihr Datenbankadministrator kann Ihnen bei den weiteren Optionen sicher helfen. Da die Vorarbeiten bereits Einfluss auf die Formulierung der Klausel haben, müssen wir diese zunächst besprechen.

10.8.2 Vorbereitung zum Einsatz

Was wir vor allem benötigen, ist eine Tabelle, in die wir die Fehler von Anweisungen schreiben können. Hierzu stehen uns zwei Varianten zur Verfügung: Wir können diese Tabelle von Oracle erzeugen lassen oder selbst erstellen. Die Tabelle, die von Oracle angelegt wird, um unsere Fehler aufzunehmen, umfasst normalerweise alle Spalten der Ursprungstabelle (also zum Beispiel von `HR_EMPLOYEES`) und fügt fünf weitere Spalten als erste Spalten der Tabelle hinzu, die Sie in Tabelle 10.1 sehen.

Spalte	Typ	Bemerkung
`ORA_ERR_NUMBER$`	`number`	Oracle-Fehlernummer
`ORA_ERR_MESG$`	`varchar2(2000)`	Oracle-Fehlermeldung
`ORA_ERR_ROWID$`	`rowid`	Zeilen-ID
`ORA_ERR_OPTYP$`	`varchar2(2)`	Art der Anweisung: ► `I` (`insert`) ► `U` (`update`) ► `D` (`delete`)
`ORA_ERR_TAG$`	`varchar2(2000)`	Tag des Laufs

Tabelle 10.1 Liste der zusätzlichen Fehlerspalten

Auch wenn wir die Fehlertabelle händisch erzeugen, sind diese fünf Spalten in jedem Fall mit diesen Datentypen und als die ersten Spalten der Tabelle anzulegen. Fehlt Ihnen in Spalte `ORA_ERR_OPTYP$` eventuell eine Option für eine `merge`-Anweisung? Das sollte es nicht, denn gemäß der Arbeitsweise dieser Anweisung werden die Fehler, in den jeweiligen `update`- oder `insert`-Zweig der Anweisung aufgeteilt, in die Fehlertabelle geschrieben.

Legen Sie die Tabelle selbst an, kann diese alle Spalten der Quelltabelle umfassen, aber auch weniger oder mehr. Die `log errors`-Klausel wird mit Werten belegen, was sie finden kann, und keine Fehler werfen, falls eine Spalte nicht vorhanden ist. Das ist auch praktisch, falls Sie eine Spalte der Quelltabelle hinzufügen, denn diese Spalte muss nicht zwingend auch in jeder Fehlertabelle gepflegt werden.

Beginnen wir mit der automatischen Erzeugung der Fehlertabelle. Wir verwenden hierzu ein mitgeliefertes *Package*, also eine Sammlung kleiner Programme in PL/SQL, die Oracle für diesen Zweck programmiert hat. Die einfachste Form, dieses Programm aufzurufen, besteht in der `call`-Anweisung, einer SQL-Anweisung, die es gestattet, Programme direkt aufzurufen. Wenn wir also für die Tabelle `HR_EMPLOYEES` eine Fehlertabelle anlegen lassen möchten, beginnen wir mit folgender SQL-Anwei-

sung, die nur einmal ausgeführt werden muss, um die Fehlertabelle anzulegen, und danach nicht mehr:

```
SQL> call dbms_errlog.create_error_log(
  2         dml_table_name => 'HR_EMPLOYEES',
  3         err_log_table_name => 'HR_EMPLOYEES_ERR');
Aufruf wurde abgeschlossen.
```

Listing 10.23 Aufruf des Packages zur Erzeugung der Fehlertabelle

Der Aufruf dieses Packages erfolgt so, dass zunächst der Name des Packages, `dbms_errlog`, gefolgt von einem Punkt und dem Namen des konkreten Programms innerhalb dieses Packages, `create_error_log`, geschrieben wird. Anschließend werden dem Programm, ähnlich einer Zeilenfunktion, zwei Parameter übergeben, nämlich der Name der Tabelle, für die eine Fehlertabelle erstellt werden soll, und der Name der zu erzeugenden Fehlertabelle.

Diese Prozedur bietet noch weitere Parameter an, doch reicht für unsere Verwendung die Angabe der Quell- und Zieltabellenbezeichner. Anschließend können wir uns die Struktur der erzeugten Fehlertabelle ansehen. Hierfür verwende ich im folgenden Beispiel einen `SQL*Plus`-Befehl `desc` (wie *describe* = *beschreibe*), der die Spalten der neu geschaffenen Tabelle darstellt:

```
SQL> desc hr_employees_err;
 Name                 Typ
 -------------------- ---------------
 ORA_ERR_NUMBER$      NUMBER
 ORA_ERR_MESG$        VARCHAR2(2000)
 ORA_ERR_ROWID$       ROWID
 ORA_ERR_OPTYP$       VARCHAR2(2)
 ORA_ERR_TAG$         VARCHAR2(2000)
 EMP_ID               VARCHAR2(4000)
 EMP_FIRST_NAME       VARCHAR2(4000)
 EMP_LAST_NAME        VARCHAR2(4000)
 EMP_EMAIL            VARCHAR2(4000)
 EMP_PHONE_NUMBER     VARCHAR2(4000)
 EMP_HIRE_DATE        VARCHAR2(4000)
 EMP_JOB_ID           VARCHAR2(4000)
 EMP_SALARY           VARCHAR2(4000)
 EMP_COMMISSION_PCT   VARCHAR2(4000)
 EMP_EMP_ID           VARCHAR2(4000)
 EMP_DEP_ID           VARCHAR2(4000)
```

Listing 10.24 Beschreibung der erzeugten Error-Logging-Tabelle

Natürlich können Sie nach Aufruf dieses Programms (und der Aktualisierung der Liste der Tabellen) auch im SQL Developer diese Tabellendetails einsehen. Sie erkennen, dass die Tabelle einerseits die fünf bereits angesprochenen Spalten enthält. Dann sehen Sie andererseits alle Spalten der Tabelle `HR_EMPLOYEES` wieder, allerdings reduziert (erweitert?) auf den Datentyp `varchar2(4000)`. Das hat folgenden Grund: Sollte ein Fehler beim Einfügen des Datensatzes auftauchen, zum Beispiel weil der Begriff zu lang für eine Spalte ist, muss er immer noch in die Fehlertabelle passen. Gemeinsame Grundlage aller Daten ist der Texttyp, denn in diesen können alle anderen Datentypen konvertiert werden.

Die maximale Länge dieses Datentyps wird verwendet, um möglichst viele fehlerhafte Werte aufnehmen zu können. Wären die Grenzen enger gesteckt, könnte es sein, dass auch die Fehlertabelle die fehlerhaften Daten nicht aufnehmen kann, womit nichts gewonnen wäre. Verwenden Sie die Option der Datenbank, erweiterte Spaltenlängen zu benutzen, sehen Sie hier den Typ `varchar2(32767)`, der dann gültigen Maximallänge für Zeichenkette in Tabellen.

10.8.3 Verwendung der Klausel LOG ERRORS

Nachdem die Fehlertabelle angelegt ist, benötigen wir eine `insert`-Anweisung aus Listing 10.25, die Fehler produziert, um die Klausel `log errors` in Aktion zu erleben. Ich möchte Daten einfügen, die sämtlich Fehler provozieren.

```
SQL> insert into hr_employees (
  2         emp_id, emp_last_name, emp_email, emp_job_id,
  3         emp_salary, emp_dep_id, emp_hire_date)
  4  values -- EMP_ID existiert bereits
  5         (150, 'Peters', 'MPETERS', 'IT_PROG',
  6          6000, 60, trunc(sysdate)),
  7         -- EMP_LAST_NAME zu lang
  8         (280, 'Lahmar-Schadler vom Hohenhorst', 'ALAHMAR', 'AD_VP',
  9          15000, 90, trunc(sysdate)),
 10         -- EMP_DEP_ID existiert nicht
 11         (290, 'Müller', 'TMÜLLER', 'SA_REP',
 12          4500, 350, trunc(sysdate))
 13     log errors
 14    into hr_employees_err ('Testlauf') reject limit 10;
0 Zeilen erstellt.

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.25 Anwendung der Klausel LOG ERRORS

Dass etwas schiefgelaufen ist, sehen wir daran, dass 0 Zeilen eingefügt wurden, nicht mehr jedoch daran, dass die Anweisung zurückgewiesen wurde, wie dies ohne log errors-Klausel der Fall gewesen wäre. Doch bevor wir uns die Fehler ansehen, werfen wir einen Blick auf die Syntax. Ich habe nach der insert-Anweisung die Klausel log errors verwendet. Mit dem Schlüsselwort into geben wir den Namen der Fehlertabelle an, die wir vorhin mit der Funktion dbms_errlog.create_error_log erzeugt haben. In Klammern und als Zeichenkette folgt dann das Tag des Laufs. Unter diesem Namen (den wir durchaus auch dynamisch hätten berechnen können, vielleicht mit Verweis auf die Funktion sysdate) finden wir später die Fehler dieser Anweisung leichter wieder. Schließlich habe ich festgelegt, dass nach spätestens 10 Fehlern jeder weitere Fehler zum Abbruch der Anweisung führt. Damit können wir eine Notbremse einbauen, falls etwas gründlich schiefläuft. Lassen Sie diese Klausel weg, wird als reject limit 0 angenommen, was dazu führt, dass sich die Anweisung verhält, als wäre überhaupt keine Klausel log errors angegeben worden: Der erste Fehler lässt die gesamte Anweisung scheitern (allerdings wird dieser erste Fehler in die Fehlertabelle geschrieben). Nur, wie viele Fehler sind maximal möglich? Vielleicht interessiert Sie die genaue Maximalgrenze jetzt nicht so sehr, aber ich möchte Ihnen anhand dieses Beispiels in Listing 10.26 einmal aufzeigen, wie freches Fragen zu interessanten Antworten führen kann: In der Dokumentation zu diesem Befehl ist die Obergrenze nicht angegeben, dort wird lediglich gesagt, dass eine Ganzzahl als Obergrenze angegeben werden kann.

```
insert into hr_employees (
       emp_id, emp_last_name, emp_email, emp_job_id,
       emp_salary, emp_dep_id, emp_hire_date)
values (150, 'Peters', 'MPETERS', 'IT_PROG', 6000, 60, trunc(sysdate))
   log errors
  into hr_employees_err ('Testlauf') reject limit 100000000000;

SQL-Fehler: ORA-30645: Zurückweisungsgrenze außerhalb des zulässigen Bereichs
            30645. 00000 -  "reject limit out of range"
*Cause:    Reject limit specifies the number of records rejected before
           terminating a table scan. The range is a either a number between
           1..100000 or UNLIMITED if no limit is intended.
*Action:   Change the token representing the reject limit to either a number
           in the range of 0 and 100000 or the keyword UNLIMITED.
```

Listing 10.26 Fehlermeldung bei zu großer Obergrenze

Der Vorteil der Fehlermeldung innerhalb des SQL Developers ist, dass direkt nachgeschlagen wird, was der Grund und was die Lösung dieses Fehlers ist. Diese Information müssten Sie ansonsten in der Online-Dokumentation nachschlagen. Und gerade hier

entdecken wir die interessante Information: Weniger, dass wir maximal 100.000 Fehler explizit tolerieren können, sondern dass es noch das ansonsten verschwiegene Schlüsselwort `UNLIMITED` gibt, das wir verwenden können, wenn wir alle Fehler darstellen möchten. Raum für Fragen lässt zudem die Inkonsistenz zwischen der Beschreibung des Fehlers und der zu wählenden Aktion, denn in der Begründung des Fehlers wird die `0` verschwiegen ...

10.8.4 Darstellung der Fehler

Nun sind also mit der ersten Anweisung unsere Fehler bereits in die Fehlertabelle geschrieben worden. Wenn Sie das Beispiel am Computer nachvollzogen haben, können Sie sich ja einmal die gesamte Fehlertabelle ansehen, hier muss ich mich auf einen Ausschnitt beschränken, weil ich ansonsten kein ordentliches Layout mehr hinbekomme:

```
SQL> select ora_err_number$ err_no,
  2         ora_err_mesg$,
  3         ora_err_tag$ ora_tag$,
  4         emp_last_name
  5    from hr_employees_err;
ERR_NO ORA_ERR_MESG$                                ORA_TAG$ EMP_LAST_NAME
------ ---------------------------------------- -------- -------------
     1 ORA-00001: Unique Constraint                Testlauf Peters
       (SQL_BUCH.HR_EMPLOYEES_PK) für Tabelle
       SQL_BUCH.HR_EMPLOYEES mit Spalten (EMP_ID)
       verletzt
       ORA-03301: (ORA-00001-Details) Zeile mit
       Spaltenwerten (EMP_ID:150) ist bereits
       vorhanden
 12899 ORA-12899: Wert zu groß für Spalte          Testlauf Lahmar-Schadler
       "SQL_BUCH"."HR_EMPLOYEES"."EMP_LAST_NAME"           vom Hohenhorst
       (aktuell: 30, maximal: 25)
  2291 ORA-02291: Integritöts-Constraint           Testlauf Müller
       (SQL_BUCH.EMP_DEP_ID_FK) verletzt -
       übergeordneter Schlüssel nicht gefunden
```

Listing 10.27 Ausgabe der Fehlertabelle

Das Prinzip der Ausgabe wird klar, hoffe ich. Interessant ist, dass die Eintragungen in diese Tabelle die `rollback`-Anweisung überstanden haben, wie erhofft. Zudem erlauben die Angaben eine detaillierte Analyse der fehlerhaften Daten und eine anschließende Korrektur.

Die Daten sind nun in der Tabelle HR_EMPLOYEES_ERR enthalten und können von dort, wie Daten jeder anderen Tabelle auch, abgefragt, geändert oder gelöscht werden. Das erwähne ich nur für den Fall, dass Sie eventuell aufgrund der autonomen Transaktion davon ausgehen, diese Daten seien auf irgendeine Art »anders« als andere Tabellendaten. Der einzige Unterschied besteht darin, dass sie im Zuge einer autonomen Transaktion eingefügt wurden, also nicht Teil der Gesamttransaktion waren. Für die Datenbank macht dies keinen Unterschied, wenn erst einmal die Transaktion beendet wurde. Danach sind das normale Daten.

10.8.5 Einsatzszenarien

Es ist, glaube ich, leicht ersichtlich, in welchen Szenarien diese Klausel eingesetzt werden kann: wenn Daten in die Datenbank eingelesen werden, sei es, dass die Daten über eine externe Schnittstelle zu einer anderen Datenbank, über eine CSV-Datei, einen Webservice oder eine XML-Datei eingelesen wurden. Dann ist diese Klausel bei Migrationsprojekten sehr interessant, nämlich dann, wenn Daten aus mehreren Altsystemen in neue Systeme übernommen werden sollen und sich anschließend erweist, wie gut die Stammdaten für die Übernahme vorbereitet wurden.

Anwendungen profitieren von diesem Automatismus auf besondere Weise, denn dort können Daten über eine Bedienungsseite eingefügt und die Fehler direkt anschließend in tabellarischer Form angezeigt werden. Eine Anwendungsseite könnte die Bearbeitung dieser Daten ermöglichen, so dass interaktiv die offenen Fehler bearbeitet und die Daten in die eigentliche Tabelle umkopiert werden könnten. Anschließend könnten diese Daten aus der Fehlertabelle entfernt werden. Damit stellt die Fehlertabelle also eine Art To-do-Liste für importierte Daten dar.

Natürlich konnte man auch vorher schon so etwas programmieren, doch mit der log errors-Klausel ist die Programmierung auf ein Minimum beschränkt, große Teile der Programmierung können an SQL delegiert werden. Sie können diesen Vorteil gar nicht hoch genug bewerten: Einerseits erspart dies viel Zeit bei der Entwicklung des Codes; dieser Code muss ja nicht nur erstellt, sondern getestet, verwaltet und an neue Rahmenbedingungen angepasst werden. Schlimmer ist jedoch, dass die Programmierungslösung eine zeilenweise Verarbeitung erzwingt. Der Grund: Um sicherzustellen, dass eine fehlerhafte Zeile nicht zum Abbruch der gesamten Anweisung führt, darf diese Zeile nicht mit anderen Zeilen zusammen in einer SQL-Anweisung ausgeführt werden. Wird ein Fehler geworfen, ist dann nur diese eine Zeile betroffen. Dann kann, im Rahmen der Fehlerbearbeitung, diese Zeile kenntlich gemacht oder in eine Fehlertabelle geschrieben werden und beeinflusst die anderen Zeilen nicht. Diese zeilenweise Bearbeitung wird durch die log errors-Klausel unnötig und beschleunigt die Schreibprozesse so enorm.

10.9 Multi-Table-Insert

Als Erweiterung zur insert-Anweisung existiert die Möglichkeit, Daten direkt in mehrere Tabellen einfügen zu lassen. Dabei wird einer insert-Anweisung entweder eine Liste von Tabellen mitgegeben, auf die sich die Anweisung beziehen soll, oder aber es wird optional ein Entscheidungsbaum definiert, der steuert, welche Zeilen in welche Tabelle eingefügt werden sollen. Diese Option erscheint zunächst einmal etwas befremdlich, doch benötigen Sie so etwas durchaus häufiger. Stellen Sie sich vor, eine Textdatei enthielte Daten, die auf mehrere Tabellen aufgeteilt werden müssen. Diese Operation ist mit dieser Option »in einem Rutsch« möglich. Bestehende Daten einer Tabelle könnten mit einer Operation auf mehrere Teiltabellen verteilt werden und vieles mehr.

10.9.1 Kopieren von Daten in mehrere Zieltabellen

Die Syntax für diese Anweisung ist recht einfach: Wir erweitern die insert-Anweisung durch das Schlüsselwort all oder first und schreiben eine into-Klausel für jede Tabelle, die wir ansprechen möchten. Um ein Beispiel zu zeigen, möchte ich die Daten der Tabelle HR_EMPLOYEES auf drei Tabellen aufteilen. Das ist zwar für sich kein besonders sinnvolles Vorgehen, zeigt aber die Syntax. Als erste Aufgabenstellung sollen die Daten der Tabelle HR_EMPLOYEES lediglich auf drei Zieltabellen kopiert werden, das heißt, dass alle Daten ohne Fallunterscheidung übernommen werden sollen. Dies erreichen wir durch die Klausel all. Um Ihnen das Beispiel zu zeigen, benötigen wir zunächst einige Tabellen. Ich habe mehrere leere Kopien der Tabelle HR_EMPLOYEES mit den Namen HR_EMPLOYEES_US, HR_EMPLOYEES_UK, HR_EMPLOYEES_CA und HR_EMPLOYEES_DE erzeugt. Ziel ist es mittelfristig, Mitarbeiter pro Land in eine eigene Tabelle zu verlegen. Im Skript zum Kapitel sehen Sie, wie ich diese Tabellen angelegt habe. Die Anweisung, die ich dort verwende, erkläre ich in Kapitel 12, »Tabellen erstellen«, damit belasten wir uns hier noch nicht. Nun können wir uns die insert-Anweisung ansehen:

```
SQL> insert all
  2    into hr_employees_us
  3    into hr_employees_uk
  4    into hr_employees_ca
  5    into hr_employees_ca
  6  select *
  7    from hr_employees;
428 Zeilen erstellt.

SQL> select count(*)
  2    from hr_employees_us;
```

```
  COUNT(*)
----------
       107

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.28 Eine erste Multi-Table-Insert-Anweisung

Das Schlüsselwort `all` bedeutet hier, dass alle Daten in alle Tabellen geschrieben werden sollen. Es leitet die Multi-Table-Version der `insert`-Anweisung ein, so dass es syntaktisch in jedem Fall erforderlich ist. Danach folgen die `into`-Klauseln, an denen auffällig ist, dass keine Kommata zur Trennung der Tabellenliste verwendet werden. Diese etwas seltsame Schreibweise wird verständlicher, wenn wir die folgenden Erweiterungen der `insert`-Anweisung betrachten, für den Moment nehmen wir das so hin. Anschließend habe ich, wie bereits bei einer »normalen« `insert`-Anweisung, eine `select`-Abfrage formuliert, die die Daten der Tabelle `HR_EMPLOYEES` liefert. Die Kontrollabfrage, die ich im Anschluss gegen eine der neuen Tabellen ausgeführt habe, bestätigt, dass alle `107` Zeilen der Ausgangstabelle in die Tabelle `HR_EMPLOYEES_US` kopiert wurden. Der Vorteil ist vor allem darin zu sehen, dass zum Füllen der Tabellen die Datenmenge nur ein einziges Mal gelesen werden musste. Das schafft, gerade bei sehr großen Tabellen, einen nicht unerheblichen Performanzgewinn.

10.9.2 Fallweises Einfügen in jeweils eine Zieltabelle

Richtig interessant wird die Sache, wenn wir die Daten der Ursprungstabelle auf mehrere Tabellen aufteilen und nicht einfach nur stumpf kopieren. Nun benötigen wir allerdings ein Entscheidungskriterium, um zu steuern, welche Zeilen in welche Tabellen eingefügt werden sollen. Als Entscheidungskriterium dient das Land, in dem die jeweiligen Mitarbeiter arbeiten. Dieses Kriterium müssen wir über eine etwas umfangreichere `select`-Anweisung ermitteln, bevor wir es prüfen können. Da wir wissen, dass ein Mitarbeiter immer nur in einem Land arbeitet, können wir statt der Klausel `all` die Klausel `first` verwenden, um die Arbeit zu beschleunigen. Diese Klausel zeigt an, dass der erste Treffer der Fallunterscheidung ausreicht, um die Zeile zuzuordnen, die anderen Fallunterscheidungen müssen nicht mehr ausgewertet werden.

Bevor wir die Abfrage erstellen können, müssen wir noch ein weiteres Problem beseitigen, denn die Unterabfrage wird das Entscheidungskriterium `COU_ID` liefern, das in den Zieltabellen nicht enthalten ist, da sie als Kopie der Tabelle `HR_EMPLOYEES` angelegt wurden. Daher dürfen nicht alle Spalten für die jeweiligen `insert`-Anweisungen verwendet werden, sondern nur diejenigen, die in der Tabelle auch existieren. Hier hilft

eine values-Klausel, die die Spalte COU_ID nicht referenziert. Listing 10.29 zeigt die resultierende Anweisung.

```
SQL> insert first
  2    when cou_id = 'US' then
  3    into hr_employees_us
  4    values(emp_id, emp_first_name, emp_last_name, emp_email,
  5           emp_phone_number, emp_hire_date, emp_job_id,
  6           emp_salary, emp_commission_pct, emp_emp_id, emp_dep_id)
  7    when cou_id = 'UK' then
  8    into hr_employees_uk
  9    values(emp_id, emp_first_name, emp_last_name, emp_email,
 10           emp_phone_number, emp_hire_date, emp_job_id,
 11           emp_salary, emp_commission_pct, emp_emp_id, emp_dep_id)
 12    when cou_id = 'CA' then
 13    into hr_employees_ca
 14    values(emp_id, emp_first_name, emp_last_name, emp_email,
 15           emp_phone_number, emp_hire_date, emp_job_id,
 16           emp_salary, emp_commission_pct, emp_emp_id, emp_dep_id)
 17    when cou_id = 'DE' then
 18    into hr_employees_de
 19    values(emp_id, emp_first_name, emp_last_name, emp_email,
 20           emp_phone_number, emp_hire_date, emp_job_id,
 21           emp_salary, emp_commission_pct, emp_emp_id, emp_dep_id)
 22  select cou_id, e.*
 23    from hr_countries
 24    join hr_locations
 25      on cou_id = loc_cou_id
 26    join hr_departments
 27      on loc_id = dep_loc_id
 28    join hr_employees e
 29      on dep_id = emp_dep_id;
106 Zeilen erstellt.

SQL> select count(*)
  2    from hr_employees_us;
  COUNT(*)
----------
        68

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 10.29 Aufteilung der Daten auf mehrere Tabellen

Das ist so schon recht beeindruckend, geht aber noch besser. Unsere Anweisung ist noch recht wackelig und das aus mehreren Gründen. Zum einen ist die Frage, was eigentlich passiert, falls die Tabelle HR_EMPLOYEES Mitarbeiter enthielte, die nicht in der Liste der gefilterten Ländern enthalten sind. Wohin gehen diese Zeilen? Werden sie einfach ignoriert? Dann wäre interessant zu wissen, was eigentlich passiert, falls die Zieltabellen nicht exakt gleich gebaut wären wie die Tabelle HR_EMPLOYEES, wenn also zum Beispiel die Spalten in anderer Reihenfolge angelegt wären oder nur ein Teil der Spalten vorhanden wäre.

Zunächst kann ich für den ersten Punkt Entwarnung geben. Falls ein Wert der Spalte COU_ID nicht durch die Fallunterscheidung referenziert wird, wird diese Zeile schlicht ignoriert und nicht in die Zieltabellen kopiert. Anders wäre die Situation, wenn wir solche unvorhergesehenen Werte dennoch kopieren wollten. Dann müssten wir sicherstellen, dass alle Werte tatsächlich kopiert werden können. Sie sehen ja, dass die Fallunterscheidung wie bei einer case-Anweisung aussieht, daher wäre denkbar, dass eine else-Klausel existiert. Und genau das ist auch der Fall. Der zweite Punkt ist ebenfalls relativ einfach zu lösen, denn wir wissen aus der »einfachen« insert-Anweisung, dass es gute Praxis ist, die Spalten, die eingefügt werden sollen, explizit anzusprechen. Dort wird dies durch eine Auflistung der Spalten hinter der into-Klausel gemacht. Und genau dies geht auch mit der Multi-Table-Insert-Anweisung.

Aus den geschilderten Möglichkeiten sollten Sie entscheiden können, welche Optionen für Ihre Anweisung zu wählen sind. Hier einige Stichpunkte zur Entscheidungsfindung:

- Sollen alle Daten der Quelltabelle in mehrere Tabellen übernommen werden, verwenden Sie insert all ohne weitere Fallunterscheidungen.
- Existieren Fälle, wo eine Zeile in mehrere Zieltabellen übernommen werden muss, wo aber dennoch eine Fallunterscheidung benötigt wird, verwenden Sie insert all mit entsprechenden Fallunterscheidungen.
- Ist sichergestellt, dass Ihr Entscheidungskriterium deterministisch ist, verwenden Sie insert first und die entsprechenden Fallunterscheidungen.
- Beinhalten die Quelldaten mehr Spalten als die Zieltabellen, verwenden Sie die values-Klausel für jede Zieltabelle.
- Ist die Reihenfolge der Spalten in der Zieltabelle nicht gesichert gleich wie in den Quelldaten, verwenden Sie eine Aliasliste hinter dem entsprechenden into-Zweig.

Bliebe noch die Frage, auf welche Weise bei einem Multi-Table-Insert neue Primärschlüsselwerte erzeugt werden könnten. Die Lösung liegt wiederum in der Verwendung der values-Klausel, die nun erforderlich ist. Der Grund ist nachvollziehbar: Stellen Sie sich die Arbeitsweise der Anweisung vor: Es wird zunächst durch eine select-Abfrage eine Datenmenge erzeugt. Diese Datenmenge verfügt über mehrere

Spalten. Diese Spalten werden durch die `into`-Klausel der Einfügeoperationen referenziert. Dabei können Umformungen, Typkonvertierungen, Änderungen der Reihenfolge oder was auch immer sonst erforderlich werden. Diese Operationen werden innerhalb der `values`-Klausel jeder einzelnen Einfügeoperation durchgeführt. Wenn Sie einen neuen Primärschlüssel benötigen, ist ebenfalls hier der Ort, dies zu tun, denn es wird ja einfach für eine Spalte ein berechneter Wert benötigt.

10.9.3 Workshop: Schemamigration mittels Multi-Table-Insert

Als Beispiel für die Anwendung möchte ich Ihnen ein Szenario geben, das etwas komplexer ist, von dem ich aber hoffe, dass es etwas realitätsnäher als die bisherigen Beispiele ist: Wir entschließen uns, das in die Jahre gekommene Datenmodell des Schemas `HR` zu modernisieren. Diese Modernisierung wurde erforderlich, weil durch einen Change Request als neues Feature gespeichert werden können muss, welche Berufe und welche Abteilungszugehörigkeit die Mitarbeiter über die Zeit innegehabt haben. Das Problem: Aus den einfachen `1:n`-Beziehungen zwischen der Tabelle `HR_EMPLOYEES` und `HR_JOBS` auf der einen und `HR_DEPARTMENTS` werden nun `m:n`-Beziehungen. Das mag nicht unmittelbar einleuchten, ist aber so: Ein Beruf kann gleichzeitig von mehreren Mitarbeitern ausgeübt werden, ein Mitarbeiter kann (über die Zeit) mehrere Berufe ausüben.

Damit die Beispiele aus den vorhergehenden Kapiteln weiterhin funktionieren, habe ich mich entschlossen, die Tabelle `HR_EMPLOYEES` durch eine neue Tabelle `HR_EMPS` zu ersetzen und die neuen Tabellen mit entsprechend gekürzten Namen einzurichten. Die neuen Tabellen referenzieren die bereits bestehenden Tabellen. Alle Mitarbeiter ziehen also in die neue Tabelle um, behalten aber ihre Primärschlüsselinformationen. Fachlich ist eine Tabelle `HR_EMP_JOBS` und eine Tabelle `HR_EMP_DEPARTMENTS` erforderlich, die im einfachsten Fall lediglich die Primärschlüsselinformationen der beiden Tabellen beinhalten, für unseren Fall aber um ein Gültigkeitsband in Form zweier Datumsspalten (`VALID_FROM` und `VALID_UNTIL`) erweitert werden müssen. Das ist aber noch nicht alles, denn auch der Vorgesetzte eines Mitarbeiters soll über die Zeit gespeichert werden können. Das erfordert ebenfalls eine weitere Tabelle, nämlich `HR_EMP_MANAGERS`, die in den beiden relevanten Spalten die gleiche Spalte aus `HR_EMPS`, `EMP_ID` referenziert und ebenfalls die zwei Zeitspalten beinhaltet.

Im Gegenzug verschwinden die Spalten `EMP_DEP_ID`, `EMP_JOB_ID` und `EMP_EMP_ID` aus der Tabelle `HR_EMPS`, da diese Beziehungen in dedizierten Tabellen gespeichert werden. Da Sie noch nicht die SQL-Anweisungen für das Erstellen von Tabellen kennen, lösen wir dieses Problem, indem Sie einfach das Skript *change_hr_schema.sql* aus dem Beispielmaterial zu diesem Kapitel ausführen.

Das Problem: Wie übernehmen wir die Altdaten des bestehenden Datenmodells? Da wir bislang noch keine Historie pflegen, beginnen wir jetzt damit und setzen als

VALID_FROM immer EMP_HIRE_DATE ein, als VALID_UNTIL ein Datum in der (fernen) Zukunft. Jetzt aber wird es spannend: Alle Mitarbeiter müssen in die neue Tabelle HR_EMPS umziehen und einen Eintrag in den Tabellen HR_EMP_MANAGERS, HR_EMP_JOBS und HR_EMP_DEPARTMENTS erhalten. Alles soll möglichst in einem Rutsch passieren.

Arbeiten wir unsere Liste mit Entscheidungskriterien ab, stellen wir fest, dass wir Zeilen der Quelltabelle in mehrere Tabellen gleichzeitig schreiben möchten, andererseits aber keine besonderen Fallunterscheidungen benötigen. Allerdings differieren die Zieltabellen in Anzahl und Reihenfolge der Spalten, daher ist sowohl eine Spaltenalias-Liste als auch eine values-Klausel erforderlich. Oder haben wir etwas übersehen? Machen wir eine Gegenprüfung mit dem Problem null-Wert im Hinterkopf. Was passiert, wenn zum Beispiel die Spalte EMP_DEP_ID einen null-Wert enthält? In diesem Fall würde eine Zeile in Tabelle HR_EMP_DEPARTMENTS angelegt, die nicht eingefügt werden kann, denn dort ist EDE_DEP_ID Teil des Primärschlüssels. Wir benötigen also doch eine Fallunterscheidung und fügen in diesen Fällen keine Zeile in die Zuordnungstabellen ein. Der gleiche Fall trifft auf Tabelle HR_EMP_MANAGERS zu, denn King hat keinen Vorgesetzten.

Listing 10.30 zeigt die überraschend einfache insert-Anweisung für diesen Anwendungsfall. Im Gegensatz zu meinen ersten Plänen habe ich mich entschlossen, in den Zuordnungstabellen HR_EMP_DEPARTMENTS, HR_EMP_JOBS und HR_EMP_MANAGERS jeweils eigene, technische Schlüssel einzurichten. Dies wird sie in einem späteren Beispiel besser nutzbar machen. Für die insert-Anwendung hat dies die Konsequenz, diesen Schlüsselwert ermitteln zu müssen. Hier wären viele Wege möglich, ich habe mich entschlossen, den Schlüsselwert als Hashcode mittels der Funktion ora_hash aus den drei Angaben EMP_ID, Schlüssel der referenzierten Tabelle und Startdatum zu ermitteln. Das hat den großen Vorteil, dass es so unmöglich ist, die gleiche Kombination von Werten mehrfach einzugeben.

Seltsam mutet die Fallunterscheidung when null is null an, das können Sie gern auch anders formulieren, zum Beispiel when 1 = 1; es geht darum, dass diese Bedingung immer zu wahr evaluiert, da syntaktisch die when-Klausel entweder in allen oder in keinem Zweig genutzt werden darf. Da aber nur zwei Tabellen eine Fallunterscheidung benötigen, haben die anderen diese »always true«-Bedingung. Ich habe mir das angewöhnt, nachdem ich in einer Anweisung eines Mitarbeiters von Oracle einmal die Formulierung where null is not null gesehen habe. Die erschien mir witzig, geradezu philosophisch, daher habe ich das übernommen.

```
SQL> insert all
  2         when null is null then
  3         into hr_emps (
  4           emp_id, emp_first_name, emp_last_name, emp_email, emp_phone_number,
  5           emp_hire_date, emp_salary, emp_commission_pct)
```

```
 6         values (
 7           emp_id, emp_first_name, emp_last_name, emp_email, emp_phone_number,
 8           emp_hire_date, emp_salary, emp_commission_pct)
 9         when emp_dep_id is not null then
10         into hr_emp_departments(
11           ede_id, ede_emp_id, ede_dep_id, ede_valid_from)
12         values(
13           ora_hash(
14             to_char(emp_id) ||
15             emp_dep_id ||
16             to_char(emp_hire_date, 'yyyy-mm-dd')),
17           emp_id, emp_dep_id, emp_hire_date)
18         when null is null then
19         into hr_emp_jobs (
20           ejo_id, ejo_emp_id, ejo_job_id, ejo_valid_from)
21         values (
22           ora_hash(
23             to_char(emp_id) ||
24             emp_job_id ||
25             to_char(emp_hire_date, 'yyyy-mm-dd')),
26           emp_id, emp_job_id, emp_hire_date)
27         when emp_emp_id is not null then
28         into hr_emp_managers (
29           ema_id, ema_emp_id, ema_mgr_id, ema_valid_from)
30         values (
31           ora_hash(
32             to_char(emp_id) ||
33             coalesce(emp_emp_id, emp_id) ||
34             to_char(emp_hire_date, 'yyyy-mm-dd')),
35           emp_id, coalesce(emp_emp_id, emp_id), emp_hire_date)
36  select *
37    from hr_employees;
426 Zeilen erstellt.
```

Listing 10.30 Schema-Migration mit einer einzigen Anweisung

Eine schnelle Gegenprüfung der Zeilenmenge: Aus einer Tabelle wurden vier. In der Quelltabelle sind 107 Zeilen vorhanden, wir erwarten 428 Zeilen abzüglich der beiden nicht erstellten Zeilen für die nicht zugeordnete Abteilung und den fehlenden Manager, macht 426 Zeilen. Listing 10.31 zeigt stellvertretend die Ergebnisse einer Abfrage gegen eine der neuen Zuordnungstabellen.

```
SQL> select *
  2    from hr_emp_managers;

EMA_EMP_ID EMA_MGR_ID EMA_VALID_FROM EMA_VALID_UNTIL
---------- ---------- -------------- ---------------
       101        100     21.09.2005      31.12.9999
       102        100     13.01.2001      31.12.9999
       103        102     03.01.2006      31.12.9999
       104        103     21.05.2007      31.12.9999
       105        103     25.06.2005      31.12.9999
       106        103     05.02.2006      31.12.9999
…
106 Zeilen ausgewählt
```

Listing 10.31 Kontrollabfrage gegen die Tabelle HR_EMP_MANAGERS

Vielleicht vermissen Sie die Spalten …VALID_UNTIL in der insert-Anweisung, doch werden diese Werte, ähnlich den Primärschlüsselwerten neuer Zeilen, durch Standardwerte der Tabellen angereichert, darum müssen wir uns nicht kümmern. In einigen Datenmodellen wird diese Spalte weggelassen mit der Begründung, dass ein Eintrag lediglich bis zu einem jüngeren Eintrag gültig ist und man dies über eine select-Anweisung ermitteln könne. Ich bevorzuge den hier gewählten Weg, weil eine einfache Filterung über das Kriterium sysdate between ejo_valid_from and ejo_valid_until den aktuell gültigen Datensatz ermitteln kann.

Hätten wir keine neue Tabelle HR_EMPS erstellt, wäre der gesamte Umzug nicht in einer Anweisung möglich gewesen, denn die Tabelle HR_EMPLOYEES enthielte in diesen Fällen immer noch die nicht mehr benötigten Spalten sowie die Zuordnungen und Constraints. In diesem Fall würden anschließend alle nicht mehr benötigten Spalten auf NULL aktualisiert und könnten anschließend entfernt werden. Mit dem Entfernen der Spalten werden auch alle anhängenden Constraints entfernt.

Um bei einem realistischen Szenario zu bleiben, ist anzumerken, dass dieser Umzug nur einen kleinen Teil des gesamten Change Request darstellt, denn die bislang genutzten Anweisungen, die Mitarbeiter aktualisieren oder neu erfassen, funktionieren nicht mehr. Diese Anweisungen und die damit zusammenhängende Anwendungslogik muss überarbeitet und an die neuen Anforderungen angepasst werden, zudem muss bei einer select-Anweisung berücksichtigt werden, dass zu einem Mitarbeiter mehrere Zeilen ermittelt werden können, wenn nicht zeitgleich über einen entsprechenden Filter die zeitlich gewünschte Zeile ausgewählt wird und so weiter.

So ist das in SQL: Manch einfach erscheinende Abfrage ist überraschend komplex zu lösen, weil sie zum Beispiel eine Unterabfrage erfordert, andere Dinge, die sich kompliziert anhören, sind mit einer einzigen Anweisung schnell gemacht. Voraussetzung

dafür, dass Ihnen solche Ergebnisse gelingen, ist, dass Sie möglichst viele Werkzeuge in Ihrem SQL-Werkzeugkoffer haben. Und die Multi-Table-Insert-Abfrage gehört in diesen Werkzeugkoffer definitiv hinein.

Abschließend möchte ich Ihnen noch aus einem Projekt berichten, in dem diese `insert`-Option sehr hilfreich war. Wir wollten Testdaten für automatisierte Tests erzeugen und benötigten hierzu komplexe Datensätze, die in mehrere Tabellen aufgegliedert waren. Es lagen `1:n`-Beziehungen vor, allerdings reichte es für die Tests aus, das nur jeweils eine Zeile in den entsprechenden Tabellen hinterlegt wurde (es ging um das Testen eines Löschkonzepts für Bewegungsdaten). Andere Tabellen konnten belegt worden sein oder auch nicht. Anstatt nun viele `insert`-Anweisungen mit sich wiederholenden und aufeinander bezogenen Schlüsselwerten zu erstellen, haben wir eine Zusammenfassung aller unterschiedlichen Attribute in einer `select`-Anweisung vorbereitet (ab Version 23ai mittels `values`-Tabellenfunktion, versteht sich) und diese Datenmenge auf die unterschiedlichen Tabellen in einem Rutsch verteilt. Das ist nicht vorrangig wegen der schnelleren Abarbeitung von Interesse, sondern hat seinen Vorteil darin, dass Schlüsselwerte nur einmal angegeben und mehrfach referenziert werden können. Zudem werden, aufgrund der einmal eingerichteten Reihenfolge in der `insert`-Anweisung, die unterschiedlichen Tabellen in der korrekten Abfolge mit Daten versorgt. Das macht die Erfassung solcher Daten deutlich einfacher. Mir fiel dieses Beispiel wegen meiner letzten Bemerkung bezüglich des Werkzeugkoffers wieder ein, denn der Kollege, der diese Testdaten erstellen sollte, war schon einigermaßen verzweifelt ob der Aufgabe, all diese Einzelskripte schreiben und korrekt mit Daten versorgen zu müssen. Dass ich mich an dieses Werkzeug erinnert habe, hat dem Kollegen sichtlich geholfen!

Kapitel 17
Hierarchische Abfragen

Hierarchische Abfragen stellen ein gutes Beispiel für die Spezialisierung von SQL dar. Ursprünglich als Erweiterung zu SQL erstellt, haben sie – in anderer Form – Eingang in den ISO-Standard gefunden. Hierarchische Abfragen lösen ein häufig auftretendes Problem, das überraschend hinterhältig ist.

Hierarchien sind in Datenbanken schon seit geraumer Zeit selbstverständlich vorhanden. Sie lassen sich auf den ersten Blick einfach und elegant speichern, indem einfach zwei Spalten angegeben werden, deren eine die `ID` der Zeile enthält und deren andere die übergeordnete `ID`, oftmals als `PARENT_ID` bezeichnet. Sie haben das noch von der Tabelle `HR_EMPLOYEES` durch die beiden Spalten `EMP_ID` und `EMP_EMP_ID` in Erinnerung. Doch schon bei dieser einfachen Speicherung stoßen wir auf überraschende Schwierigkeiten, wenn wir die Daten als Hierarchie auslesen möchten. Lassen Sie uns zunächst die einfachste Form der hierarchischen Speicherung mit zwei Tabellenspalten betrachten, die aufeinander verweisen. Zwar kommen in der Realität oft komplexere hierarchische Speichermodelle vor, doch können wir das Grundproblem bereits an dieser einfachen Modellierung zeigen.

17.1 Das Problem

Zunächst ist alles ganz einfach. Wenn wir in der Tabelle `HR_EMPLOYEES` wissen möchten, wie der Vorgesetzte des Mitarbeiters `Pataballa` heißt, benötigen wir einen Self-Join auf die Tabelle `HR_EMPLOYEES`. Bereits bei der Besprechung der Joins haben Sie gesehen, dass der einzige Trick dieser Abfrage darin besteht, zwei Aliasse für die Tabelle `HR_EMPLOYEES` zu vereinbaren und zwischen diesen beiden Aliassen eine Join-Bedingung zu formulieren. Dadurch, so haben wir uns erklärt, stehen uns »zwei Finger« zur Verfügung, die sich unabhängig voneinander in der Tabelle `HR_EMPLOYEES` bewegen können. Schlagartig schwerer wird die Abfrage aber nun, wenn wir umgekehrt zeigen möchten, wie die Untergebenen eines Mitarbeiters inklusive der Untergebenen dieses Untergebenen etc. heißen, kurz, wenn wir versuchen, das Organigramm des Unternehmens darzustellen. Warum? Wir beginnen unsere Suche beim Mitarbeiter `King`. Dieser hat keinen Chef mehr und ist daher der Ausgangspunkt unserer Untersuchung. In der Tabelle `HR_EMPLOYEES` sind mehrere Manager direkt `King` unterstellt. Das haben

wir mit unserem zweiten Finger einfach klären können. Wenn wir aber mit dem zweiten Finger beim ersten Manager stehen und uns fragen, wie denn dessen Untergebene heißen, bekommen wir Probleme: Wir benötigen einen dritten Finger, denn verließen wir den Manager, wüssten wir nicht mehr, wo wir hergekommen sind. Wir benötigen also einen dritten und auch noch einen weiteren Finger, wenn der Untergebene unseres Managers weitere Untergebene hat. Wie viele Finger benötigen wir also? Wenn wir das Ganze etwas durchdenken, wird klar: Wir benötigen so viele Finger, wie unser Unternehmen Gliederungsebenen hat. Und wie viele sind das? Hier liegt das erste Problem: Wir wissen es nicht.

Das zweite Problem ist etwas hinterhältiger. Nehmen wir an, unser Unternehmen habe maximal vier Gliederungsebenen, wir benötigen also vier Aliasse auf die gleiche Tabelle, wie in der Abfrage aus Listing 17.1. Alle Ebenen werden durch Joins miteinander verbunden. Das Ergebnis der Abfrage zeigt allerdings nur zwei unterschiedliche Untergebene von `King`.

```
SQL> select e1.emp_last_name e1, e2.emp_last_name e2,
  2         e3.emp_last_name e3, e4.emp_last_name e4
  3    from hr_employees e1
  4    join hr_employees e2 on e1.emp_id = e2.emp_emp_id
  5    join hr_employees e3 on e2.emp_id = e3.emp_emp_id
  6    join hr_employees e4 on e3.emp_id = e4.emp_emp_id
  7   where e1.emp_id = 100
  8   order by e1, e2, e3, e4;

E1        E2           E3           E4
--------- ------------ ------------ ---------
King      De Haan      Hunold       Austin
King      De Haan      Hunold       Ernst
King      De Haan      Hunold       Lorentz
King      De Haan      Hunold       Pataballa
King      Kochhar      Greenberg    Chen
King      Kochhar      Greenberg    Faviet
King      Kochhar      Greenberg    Popp
King      Kochhar      Greenberg    Sciarra
King      Kochhar      Greenberg    Urman
King      Kochhar      Higgins      Gietz
```

Listing 17.1 Ein erster Versuch der Darstellung eines Organigramms

Was ist passiert? Der Grund für die fehlenden Zeilen liegt darin, dass nicht alle Teilbäume des Organigramms vier Ebenen haben, einige haben lediglich zwei oder drei Ebenen. Was passiert mit Teilbäumen, die nicht bis zur vierten Ebene organisiert sind? Da Ebene 2 vergeblich einen Eintrag in Ebene 3 sucht, wird die Ebene 2 nur dann

gezeigt, wenn Ebene 3 durch einen Outer Join mit Ebene 2 verbunden wurde. Sehen wir uns die Abfrage mit einem Outer Join an:

```
SQL> select e1.emp_last_name e1, e2.emp_last_name e2,
  2         e3.emp_last_name e3, e4.emp_last_name e4
  3    from hr_employees e1
  4    join hr_employees e2 on e1.emp_id = e2.emp_emp_id
  5    left join hr_employees e3 on e2.emp_id = e3.emp_emp_id
  6    left join hr_employees e4 on e3.emp_id = e4.emp_emp_id
  7   where e1.emp_id = 100
  8   order by e1, e2, e3, e4;

E1        E2           E3        E4
--------- ------------ --------- ---------
King      Cambrault    Bates
King      Cambrault    Bloom
King      Cambrault    Fox
King      Cambrault    Kumar
King      Cambrault    Ozer
King      Cambrault    Smith
King      De Haan      Hunold    Austin
King      De Haan      Hunold    Ernst
King      De Haan      Hunold    Lorentz
King      De Haan      Hunold    Pataballa
…
89 Zeilen ausgewählt
```

Listing 17.2 Hierarchische Abfrage mit Outer Joins

Das funktioniert, die Abfrage zeigt die insgesamt 89 verschiedenen Zweige unseres Organigramms. Sie sehen allerdings, dass der Aufwand ins Unermessliche stiege, wenn wir es nicht nur mit vier, sondern zum Beispiel mit 100 Ebenen zu tun hätten.

Dieses Beispiel zeigt, dass wir SQL mit diesem Abfragetyp über die Sprachgrenze hinaus belasten. Auf der einen Seite ist das verwunderlich, denn die Frage ist doch naheliegend. Auf der anderen Seite ist die Tatsache, dass sich Abfragen finden lassen, die mit SQL selbst nicht oder nicht gut zu beantworten sind, auch kein Wunder; denn jede Programmiersprache hat einen Fokus, innerhalb dessen Probleme sehr einfach und gut gelöst werden können, und Bereiche, wo dies eben nicht mehr geht. Da allerdings die Arbeit mit hierarchischen Strukturen so ein normales Problem ist, hat Oracle bereits vor vielen Jahren die Sprache um ein spezielles Konstrukt erweitert, das die Arbeit mit Hierarchien deutlich einfacher und schneller macht: die `connect by`-Klausel. Erst mit Version 11g der Datenbank kam ein alternativer Ansatz hinzu, den wir uns danach ansehen werden.

17.2 Lösung mit der CONNECT BY-Klausel

Eine elegante und schnelle Lösung des Problems liefert die `connect by`-Klausel, in deren Zusammenhang weitere Klauseln, Funktionen und Pseudospalten definiert wurden, die der einfachen Verarbeitung hierarchischer Daten dienen.

Grundform

Der Kern der Abfrage besteht aus zwei Klauseln, die einerseits festlegen, auf welche Weise die Hierarchie gebildet werden soll (also die Spalten definieren, in denen die Hierarchie gespeichert wird), und andererseits, wo in den hierarchischen Baum eingestiegen werden soll. Damit wir in der Abfrage aus Listing 17.3 auch etwas sehen, habe ich zudem die (nur in diesem Zusammenhang definierte) Pseudospalte `LEVEL` verwendet, die uns anzeigt, auf welcher Ebene der Hierarchie wir uns befinden.

```
SQL>  select level,
  2         lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3    from hr_employees
  4   start with emp_emp_id is null
  5  connect by prior emp_id = emp_emp_id;
LEVEL NAME
----- --------------
    1 King
    2 ..Kochhar
    3 ....Greenberg
    4 ......Faviet
    4 ......Chen
    4 ......Sciarra
    4 ......Urman
    4 ......Popp
    3 ....Whalen
    3 ....Mavris
    3 ....Baer
    3 ....Higgins
    4 ......Gietz
    2 ..De Haan
    3 ....Hunold
…
```

Listing 17.3 Hierarchische Abfrage mit der Klausel CONNECT BY

Vielleicht erkläre ich kurz den etwas konfus wirkenden Ausdruck in der `select`-Klausel. Dieser Ausdruck hat nichts mit der hierarchischen Abfrage selbst zu tun, sondern erzeugt lediglich, abhängig von der Schachtelungstiefe, eine unterschiedliche Anzahl

an Punkten, die vor dem Namen eingefügt werden und so dafür sorgen, dass wir in der Ausgabe der Anweisung die hierarchischen Beziehungen sehen. Wie habe ich das gemacht? Die Funktion `lpad` füllt Zeichenketten nach links auf. Ich fülle einen einzelnen Punkt auf `(level - 1) * 2` Stellen auf. Bei Ebene 1 also auf 0 Stellen, womit die Punkte verschwinden, danach pro Ebene auf 2 Punkte. An diese Punkteliste klebe ich einfach den Namen. Sie können das Ergebnis leicht prüfen, die Spalte `LEVEL` zeigt Ihnen den jeweils gültigen Wert für die Schachtelungstiefe.

Wichtiger ist aber die Klausel `connect by`, die der ganzen Abfrage ihren Namen gibt. Sie legt fest, welche Spalten zur Definition der Hierarchie verwendet werden sollen. Dabei zeigt das Schlüsselwort `prior` auf den hierarchisch höherstehenden Knoten. In unserem Beispiel schreiben wir `prior emp_id = emp_emp_id`. Der Mechanismus funktioniert so: Wir suchen zwei Zeilen in der Tabelle, für die die Spalte `EMP_ID` der einen Zeile gleich der Spalte `EMP_EMP_ID` der anderen Spalte ist. Nun haben wir zwei Zeilen, die diese Bedingung erfüllen. Welche der beiden Zeilen ist hierarchisch höher? Da wir den Vorzug der Zeile geben, die den Wert in der Spalte `EMP_ID` hat, wird diese Zeile hierarchisch höher eingestuft. Das Schlüsselwort `prior` kann auch mehrfach in einer Abfrage verwendet werden und hat eine ähnliche Funktion wie die analytische Funktion `lag`, denn sie ermöglicht den Zugriff auf eine vorhergehende Zeile (in diesem Kontext die hierarchisch höherstehende Zeile), ähnlich wie das auch die Funktion `lag` tut.

In der Abfrage wird die Klausel `start with` verwendet. Diese Klausel legt fest, wo wir in den hierarchischen Baum einsteigen möchten, und erwartet eine boolesche Prüfung, die für eine oder mehrere Zeilen zu `wahr` evaluiert werden muss, wenn wir überhaupt eine Ausgabe sehen wollen. Ich habe die Zeile gewählt, deren Spalte `EMP_EMP_ID` einen `null`-Wert enthält, weil ich das gesamte Organigramm zeigen wollte. Wir hätten auch eine Ebene tiefer einsteigen können, wie man in Listing 17.4 sieht. Die einzige Änderung dieser Abfrage ist der neue boolesche Vergleich in der Klausel `start with`. Das Ergebnis zeigt nicht mehr den Mitarbeiter `King`, denn der wird durch die Hierarchie nicht mehr gesehen, sondern die direkten Untergebenen von `King` auf Ebene 1 sowie ihre jeweilige Untergebenen.

Seine Einfachheit macht diesen Abfragetyp so bestechend: Wir haben keinerlei Self-Joins benötigt, die gesamte Hierarchie wird durch einen einfachen Full Table Scan auf die Tabelle `HR_EMPLOYEES` sowie eine spezialisierte Optimierung sehr performant erledigt, wie Sie im Ausführungsplan sehen.

```
SQL>  select level,
  2         lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3    from hr_employees
  4   start with emp_emp_id = 100
  5 connect by prior emp_id = emp_emp_id;
```

```
LEVEL NAME
----- --------------
    1 Kochhar
    2 ..Greenberg
    3 ....Faviet
    3 ....Chen
    3 ....Sciarra
    3 ....Urman
    3 ....Popp
    2 ..Whalen
    2 ..Mavris
    2 ..Baer
    2 ..Higgins
    3 ....Gietz
    1 De Haan
    2 ..Hunold
    3 ....Ernst
…

Ausführungsplan
----------------------------------------------------------------
| Id  | Operation                          | Name | Rows  |
----------------------------------------------------------------
|   0 | SELECT STATEMENT                   |      |    14 |
|*  1 |  CONNECT BY NO FILTERING WITH START-WITH|      |       |
|   2 |   TABLE ACCESS FULL                | EMP  |    14 |
----------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("MGR"=PRIOR "EMPNO")
       filter("JOB"='MANAGER')
```

Listing 17.4 Hierarchische Abfrage mit mehreren Einsprungspunkten

Als Erweiterung dieses Konzepts zeige ich Ihnen in Listing 17.5 noch zwei weitere Abfragen, um Ihnen die Verwendung des Schlüsselworts `prior` näherzubringen. Ich hatte angedeutet, dass es eine ähnliche Funktion wie die analytische Funktion `lag` hat und uns Zugriff auf die vorhergehende Zeile gewährt. Dies machen wir uns in der ersten Abfrage zunutze, um herauszufinden, wie der jeweilige Vorgesetzte heißt.

In der zweiten Abfrage benutzen wir die Klausel zweimal in der Bedingung, um nur die Vorgesetzten zu finden, die als Abteilungsleiter arbeiten. Diese zeichnen sich dadurch aus, dass sie entweder keinen Vorgesetzten besitzen oder dass dieser in einer anderen Abteilung arbeitet.

```
SQL>  select level, emp_last_name, prior emp_last_name manager
  2     from hr_employees
  3    start with emp_emp_id is null
  4  connect by prior emp_id = emp_emp_id;

LEVEL EMP_LAST_NAME MANAGER
----- ------------- ---------
    1 King
    2 Kochhar       King
    3 Greenberg     Kochhar
    4 Faviet        Greenberg
    4 Chen          Greenberg
    4 Sciarra       Greenberg
    4 Urman         Greenberg
…

SQL>  select emp_last_name, emp_dep_id
  2     from hr_employees
  3    start with emp_emp_id is null
  4  connect by prior emp_id = emp_emp_id
  5      and prior emp_dep_id != emp_dep_id;

EMP_LAST_NAME EMP_DEP_ID
------------- ----------
King                  90
Raphaely              30
Weiss                 50
Fripp                 50
Kaufling              50
Vollman               50
Mourgos               50
Russell               80
Partners              80
Errazuriz             80
Cambrault             80
Zlotkey               80
Hartstein             20
13 Zeilen ausgewählt.
```

Listing 17.5 Verwendung der Klausel PRIOR

In diesem Beispiel ist der boolesche Ausdruck, der festlegt, welche Zeile als Vorgänger zur aktuellen Zeile anzusehen ist, aus zwei Teilprüfungen aufgebaut, daher werden

nur die Mitarbeiter ausgegeben, die hierarchisch pro Abteilung am höchsten stehen. Insbesondere sind Kochhar und de Haan als Manager nicht mehr dabei, denn ihr Chef King arbeitet in der gleichen Abteilung wie sie. In dieser Abfrage ist es zudem unumgänglich, die connect by-Klausel durch diese Erweiterung zu ändern, in der where-Klausel hätte diese Bedingung nicht funktioniert. Beachten Sie auch, dass Kimberley Grant nicht ausgegeben wird, was daran liegt, dass sie keiner Abteilung angehört.

17.2.1 Die Pseudospalte LEVEL

Ein Wort zur Pseudospalte LEVEL. Diese wird, wie alle anderen Pseudospalten auch, zusammen mit der Hierarchie berechnet, und zwar sehr früh im Zyklus der Abfrage, denn sie wird bereits in der from-Klausel gefüllt. Das bedeutet, dass diese Pseudospalte eben keine Zeilenfunktion ist, die in der select-Klausel ausgewertet wird, sondern bereits vorher gefüllt ist. Daher können Filterungen über diese Spalte in der where-Klausel vorgenommen werden, ebenso wie Partitionierungen von Gruppenfunktionen. Betrachten Sie diese Spalte einfach wie jede andere in Ihrer Tabelle. In Listing 17.6 möchte ich Ihnen einige kleine Auswertungen zeigen, die mit der Pseudospalte arbeiten. Ich stelle die Abfragen einfach mit einem kurzen Kommentar hintereinander; Sie können sich die Abfragen sicher leicht erklären, da keine neuen Techniken eingesetzt werden.

```
SQL> -- Maximale Anzahl Gliederungsebenen
SQL>  select max(level) max_level
  2     from hr_employees
  3    start with emp_emp_id is null
  4  connect by prior emp_id = emp_emp_id;

MAX_LEVEL
---------
        4

SQL> -- Anzahl Mitarbeiter und Durchschnittsgehalt pro Ebene
SQL>  select level, avg(emp_salary) avg_sal, count(*) anzahl
  2     from hr_employees
  3    start with emp_emp_id is null
  4  connect by prior emp_id = emp_emp_id
  5    group by level;

LEVEL    AVG_SAL ANZAHL
----- ---------- ------
    1      24000      1
    2      11100     14
```

```
    3  5418,4878     82
    4       6770     10

SQL> -- Zeige nur Mitarbeiter in Level >= 3
SQL>  select level,
  2         lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3    from hr_employees
  4   where level >= 3
  5   start with emp_emp_id is null
  6  connect by prior emp_id = emp_emp_id;

LEVEL NAME
----- -------------
    3 ....Greenberg
    4 ......Faviet
    4 ......Chen
    4 ......Sciarra
    4 ......Urman
    4 ......Popp
    3 ....Whalen
    3 ....Mavris
…
```

Listing 17.6 Einige Abfragen mit der Pseudospalte LEVEL

17.2.2 Sortierung mit ORDER SIBLINGS BY

Eine einfache Sortierung einer hierarchischen Abfrage wie in Listing 17.7 wäre zwar technisch möglich, aber nicht sinnvoll, weil dadurch die optische Zusammengehörigkeit der Knoten wieder aufgelöst würde. Eigentlich sollten die Knoten nach Ebene sortiert werden und nicht über die Hierarchiegrenzen hinweg. Das geht mit normalen Mitteln nicht, sondern nur durch eine Erweiterung der Klausel `order by` um das Schlüsselwort `siblings` (Geschwister), das sich auf Knoten bezieht, die einen gemeinsamen Vorgesetzten haben.

Die Verwendung zeigt die zweite Abfrage. Sie erkennen, dass auf Ebene `3 Bates` vor `Bloom` und `Fox` sortiert wurde, auch auf den anderen Ebenen stimmt die Sortierung. Natürlich gilt für diese Erweiterung, dass sie lediglich im Umfeld von `connect by`-Abfragen verwendet werden kann, ebenso wie die Pseudospalte `LEVEL`.

```
SQL>  select level,
  2         lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3    from hr_employees
  4   start with emp_emp_id is null
```

```
  5  connect by prior emp_id = emp_emp_id
  6    order by emp_last_name;

LEVEL NAME
----- -------------
    3 ....Abel
    3 ....Ande
    3 ....Atkinson
    4 ......Austin
    3 ....Baer
    3 ....Baida
    3 ....Banda
    3 ....Bates
    3 ....Bell
…

SQL>  select level,
  2         lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3     from hr_employees
  4    start with emp_emp_id is null
  5  connect by prior emp_id = emp_emp_id
  6    order siblings by emp_last_name;

LEVEL NAME
----- -----------
    1 King
    2 ..Cambrault
    3 ....Bates
    3 ....Bloom
    3 ....Fox
    3 ....Kumar
    3 ....Ozer
    3 ....Smith
    2 ..De Haan
    3 ....Hunold
    4 ......Austin
    4 ......Ernst
    4 ......Lorentz
    4 ......Pataballa
…
```

Listing 17.7 Hierarchische Abfrage mit falscher Sortierung

17.3 Erweiterungen der CONNECT BY-Klausel

Im Laufe der Zeit sind einige praktische Erweiterungen zu diesem Abfragetyp in Oracle integriert worden. Diese Erweiterungen ermöglichen es auf einfache Weise, eine Analyse der Hierarchie vorzunehmen, sich die Pfade zu einem Blatt des Baums anzeigen zu lassen und vieles mehr.

17.3.1 Weitere Pseudospalten

Im Zusammenhang mit der hierarchischen Abfrage definiert Oracle zwei weitere Pseudospalten neben `LEVEL`: `CONNECT_BY_ISLEAF` und `CONNECT_BY_ISCYCLE`. Von beiden Spalten halte ich die erste für die, welche häufiger genutzt werden dürfte: Sie zeigt an, ob ein Knoten noch Kindelemente hat oder nicht. In der Terminologie von Bäumen wird unterschieden zwischen Knoten und Blättern. Hat ein Knoten keine weiteren Kindelemente, wird er als *Blatt* bezeichnet, englisch also als `leaf`. Demzufolge ist auch klar, woher der Name der Funktion kommt. Die Pseudospalte `CONNECT_BY_ISCYCLE` zeigt an, ob eine Zeile einen Zirkelbezug erzeugt oder nicht. Stellen Sie sich dazu vor, Clerk `Feeney` sei in unserem Datenmodell der Vorgesetzte von Präsident `King`. Dann wäre ein Kreis geschlossen, die Hierarchie ginge unendlich weiter. In diesem Fall würde bei `Feeney` angezeigt, dass hier ein Zirkelbezug beginnt. Beide Pseudospalten liefern entweder eine `0` oder eine `1`, die `falsch` oder `wahr` entsprechen und ebenfalls genutzt werden können wie die Pseudospalte `LEVEL`. In der Folge zeige ich die Anwendung mit einigen kleinen Alternativen.

Um die Pseudospalte `CONNECT_BY_ISCYCLE` in Aktion zu zeigen, musste ich allerdings die Daten so verändern, dass `Feeney` der Manager von `King` wurde. Zudem muss bei einer hierarchischen Abfrage, die Zirkelschlüsse beinhalten kann, die Option `nocycle` gesetzt werden, wie in den Abfragen aus Listing 17.8 zu sehen. Diese Option verhindert, dass Oracle in eine Endlosschleife gerät und einen Fehler wirft.

```
SQL> update hr_employees
  2     set emp_emp_id = 197
  3   where emp_id = 100;
1 Zeile wurde aktualisiert.

SQL>  select level,
  2          lpad('.', (level - 1) * 2, '.') || emp_last_name name
  3     from hr_employees
  4    start with emp_emp_id = 100
  5  connect by prior emp_id = emp_emp_id
  6    order siblings by emp_last_name;
```

```
LEVEL NAME
----- --------------------
    1 Cambrault
    2 ..Bates
    2 ..Bloom
    2 ..Fox
    2 ..Kumar
…
ERROR:
ORA-01436: CONNECT-BY-Schleife in Benutzerdaten
Hilfe: https://docs.oracle.com/error-help/db/ora-01436/

SQL>  select level, emp_last_name name, connect_by_iscycle
  2     from hr_employees
  3    start with emp_id = 100
  4  connect by nocycle prior emp_id = emp_emp_id
  5    order siblings by emp_last_name;
LEVEL NAME       CONNECT_BY_ISCYCLE
----- ---------- ------------------
    1 King                       0
    2 Cambrault                  0
    3 Bates                      0
    3 Bloom                      0
…
    3 Feeney                     1
…
107 Zeilen ausgewählt

SQL> rollback;
Transaktion mit ROLLBACK rückgängig gemacht.
```

Listing 17.8 Verwendung der Pseudospalte CONNECT_BY_ISCYCLE

Die Pseudospalte `CONNECT_BY_ISLEAF` kann jederzeit verwendet werden und zum Beispiel dazu genutzt werden, um zu analysieren, ob ein Mitarbeiter Untergebene hat oder, im Umfeld einer Anwendung, in der die Hierarchie ein Steuerelement füllt, um zu steuern, ob ein Ordner- oder Dateisymbol angezeigt werden soll. Listing 17.9 zeigt einfache Beispiele für die Verwendung.

```
SQL>  select level, emp_last_name, connect_by_isleaf
  2     from hr_employees
  3    start with emp_emp_id is null
  4  connect by prior emp_id = emp_emp_id
  5    order siblings by emp_last_name;
```

```
     LEVEL EMP_LAST_NAME CONNECT_BY_ISLEAF
---------- ------------- -----------------
         1 King                          0
         2 Cambrault                     0
         3 Bates                         1
         3 Bloom                         1
         3 Fox                           1
         3 Kumar                         1
         3 Ozer                          1
         3 Smith                         1
         2 De Haan                       0
         3 Hunold                        0
…
SQL> -- Filterung über die Pseudospalte CONNECT_BY_IS_LEAF
SQL>  select level, emp_last_name, connect_by_isleaf
  2     from hr_employees
  3    where connect_by_isleaf = 0
  4    start with emp_emp_id is null
  5  connect by prior emp_id = emp_emp_id
  6    order siblings by emp_last_name;

LEVEL EMP_LAST_NAME CONNECT_BY_ISLEAF
----- ------------- -----------------
    1 King                          0
    2 Cambrault                     0
    3 Bates                         1
    3 Bloom                         1
    3 Fox                           1
    3 Kumar                         1
    3 Ozer                          1
    3 Smith                         1
    2 De Haan                       0
    3 Hunold                        0
…
18 Zeilen ausgewählt
```

Listing 17.9 Verwendung der Pseudospalte CONNECT_BY_ISLEAF

17.3.2 Operator CONNECT_BY_ROOT

Eine ähnliche Funktion wie der Operator `prior` hat der Operator `connect_by_root`. Im Gegensatz zu `prior` ermöglicht er den Zugriff auf das jeweilige Wurzelelement der Zeile und nicht auf den unmittelbaren Vorgänger. Listing 17.10 zeigt, wie auf den

Namen des jeweils höchsten Vorgesetzten zugegriffen werden kann. Um das Ergebnis interessanter zu gestalten, habe ich als Startpunkt die direkten Untergebenen von `King` gewählt, ansonsten wäre das Ergebnis stets der Präsident gewesen.

```
SQL>  select level, emp_last_name, connect_by_root(emp_last_name) manager
  2     from hr_employees
  3    start with emp_emp_id = 100
  4  connect by prior emp_id = emp_emp_id
  5    order siblings by emp_last_name

LEVEL EMP_LAST_NAME MANAGER
----- ------------- ------------
    1 Cambrault     Cambrault
    2 Bates         Cambrault
    2 Bloom         Cambrault
    2 Fox           Cambrault
    2 Kumar         Cambrault
    2 Ozer          Cambrault
    2 Smith         Cambrault
    1 De Haan       De Haan
    2 Hunold        De Haan
    3 Austin        De Haan
    3 Ernst         De Haan
    3 Lorentz       De Haan
    3 Pataballa     De Haan
    1 Errazuriz     Errazuriz
    2 Ande          Errazuriz
...
```

Listing 17.10 Verwendung des Operators CONNECT_BY_ROOT

17.3.3 Die Funktion SYS_CONNECT_BY_PATH

Schließlich existiert noch eine Zeilenfunktion im Zusammenhang mit hierarchischen Abfragen, die es ermöglicht, den Pfad zu einem Knoten auszugeben: die Funktion `sys_connect_by_path`. Warum diese Funktion das Präfix `sys_` benötigt, entzieht sich meiner Kenntnis ... Diesmal müssen zwei Parameter übergeben werden, nämlich zum einen die Spalte, die als Pfad ausgegeben werden soll, und zum anderen das Trennzeichen, das zwischen den Pfadangaben verwendet werden soll.

```
SQL>  select level, emp_last_name,
  2         sys_connect_by_path(emp_last_name, '/') pfad
  3    from hr_employees
```

```
  4    start with emp_emp_id is null
  5  connect by prior emp_id = emp_emp_id
  6    order siblings by emp_last_name;

LEVEL EMP_LAST_NAME PFAD
----- ------------ --------------------------------
    1 King         /King
    2 Cambrault    /King/Cambrault
    3 Bates        /King/Cambrault/Bates
    3 Bloom        /King/Cambrault/Bloom
    3 Fox          /King/Cambrault/Fox
    3 Kumar        /King/Cambrault/Kumar
    3 Ozer         /King/Cambrault/Ozer
    3 Smith        /King/Cambrault/Smith
    2 De Haan      /King/De Haan
    3 Hunold       /King/De Haan/Hunold
    4 Austin       /King/De Haan/Hunold/Austin
    4 Ernst        /King/De Haan/Hunold/Ernst
    4 Lorentz      /King/De Haan/Hunold/Lorentz
    4 Pataballa    /King/De Haan/Hunold/Pataballa
    2 Errazuriz    /King/Errazuriz
    3 Ande         /King/Errazuriz/Ande
...
```

Listing 17.11 Verwendung der Funktion SYS_CONNECT_BY_PATH

Diese Funktion kann vielfältig genutzt werden. Ich möchte in Listing 17.12 gerne ein weiteres Beispiel zeigen, das uns mitteilt, welche übergeordneten Mitarbeiter von einer Änderung an einem untergeordneten Mitarbeiter in Kenntnis gesetzt werden müssen. Ich benutze hier eine Unterabfrage, in die ich die hierarchische Abfrage verlege.

```
SQL> with managers as(
  2         select emp_last_name manager_name,
  3                sys_connect_by_path(emp_last_name, '|') path
  4           from hr_employees
  5          where emp_last_name = 'Pataballa'
  6          start with emp_emp_id is null
  7        connect by prior emp_id = emp_emp_id)
  8  select distinct emp_last_name
  9    from hr_employees
 10    join managers
 11      on instr(path, emp_last_name) > 0
 12   where emp_last_name != manager_name;
```

```
EMP_LAST_NAME
----------
De Haan
Hunold
King
```

Listing 17.12 Beispielabfrage mit SYS_CONNECT_BY_PATH

Was hier genau passiert, möchte ich Ihnen gern als Tüftelaufgabe überlassen. Zusatzfrage: Warum musste ich `distinct` verwenden? Versuchen Sie sich doch auch an einer alternativen Lösung, die, ausgehend vom Blatt, den Weg zum Knoten ermittelt.

17.3.4 Ein etwas komplexeres Anwendungsbeispiel

Ich möchte Ihnen gern ein etwas komplexeres Anwendungsbeispiel einer hierarchischen Abfrage vorstellen. Dazu benötige ich etwas Vorarbeit: Wir werden zwei Tabellen mit Daten füllen und diese dann auswerten. Doch zunächst das Szenario: In einem Projekt werden verschiedene Aufgaben definiert. Die einzelnen Aufgaben sind hierarchisch aufgebaut, allerdings nicht in einem einfachen Baum, sondern als komplexeres Gewebe von Abhängigkeiten. All diese Aufgaben benötigen eine gewisse Zeit, um ausgeführt zu werden. Einige Aufgaben können erst begonnen werden, wenn alle hierarchisch vorhergehenden Aufgaben abgeschlossen wurden. Uns interessiert der sogenannte *kritische Pfad*, also die Frage: Welche Aufgaben haben die Dauer des Gesamtprojekts bestimmt? Es ist nicht ganz einfach, diese Frage zu beantworten, und viele Anwender von SQL reagieren auf diese Problemstellung, indem sie eine Prozedur programmieren. Aufgrund eines solchen Programms bin ich auch auf dieses Beispiel gekommen. Ausgangspunkt für unser Beispiel ist ein Blog von *leniel*, einem brasilianischen Programmierer, der eine C#-Implementierung dieses Problems vorstellt. Ich bin auf dieses Blog gestoßen, weil ich wiederum in einem Projekt genau dieses Verfahren im bestehenden Code gefunden und mich gefragt habe, ob so etwas nicht auch direkt in SQL geht.

Beginnen wir damit, uns das Problem anzusehen. Abbildung 17.1 habe ich dem Blog von *leniel* entnommen. Der obere Teil der Abbildung stellt verschiedene Aufgaben innerhalb eines Projekts dar. Die Pfeile zeigen die Abhängigkeiten. Jedes Projekt hat (innerhalb des Kreises angegeben) eine Ausführungsdauer. Oberhalb einer Aufgabe ist vermerkt, wann diese Aufgabe frühestens hätte gestartet und beendet werden können, unten stehen die entsprechenden Angaben für den spätestmöglichen Zeitpunkt. Geben Sie sich gern etwas Zeit, die Angaben zu verstehen. Als Beispiel können Sie Projekt `C` verwenden: Es dauert 2 Minuten, diese Aufgabe zu erfüllen. Da aber zunächst Aufgabe `A` erledigt werden muss und diese Aufgabe 3 Minuten benötigt, kann diese Aufgabe frühestens nach 3 Minuten gestartet werden und wäre demnach nach frühestens 5 Minuten erledigt. Auf dieser Ebene der Aufgaben ist Aufgabe `E` die-

jenige, die am längsten zur Ausführung benötigt, nämlich 5 Minuten. Daher könnte Aufgabe C auch noch so gestartet werden, dass sie nach insgesamt 8 Minuten (Aufgabe A plus Aufgabe E benötigen so lange) beendet würde. Da Aufgabe C 2 Minuten benötigt, könnte sie also nach 6 Minuten spätestens gestartet worden sein, ohne die Gesamtausführungsdauer des Projekts zu verändern.

Der untere Teil der Abbildung zeigt den kritischen Pfad, also die Aufgabenfolge, die die Gesamtausführungszeit bestimmt hat. Ich hatte für die erste und zweite Ebene ja bereits erklärt, dass Aufgabe A und E auf dem kritischen Pfad liegen, denn diese beiden Aufgaben bestimmen die Gesamtausführungszeit. Der Algorithmus, den *leniel* in C# vorstellt, berechnet den kritischen Pfad für das gesamte Projekt, und das ist sicher eine komplexere Aufgabe, denn es muss ja die Abhängigkeitskette genau analysiert und die pro Ebene längste Aufgabe ermittelt werden. Aufgabenketten können aber auch enden, ohne für das gesamte Projektergebnis relevant zu sein. Diesen Fall hätten wir zum Beispiel, wenn Aufgabe Z (in der obersten Zeile der Abbildung) nicht noch eine Abhängigkeit zu Aufgabe V hätte. Dieser Teilbaum der Aufgaben endet dann halt, bevor der Hauptbaum abgearbeitet ist.

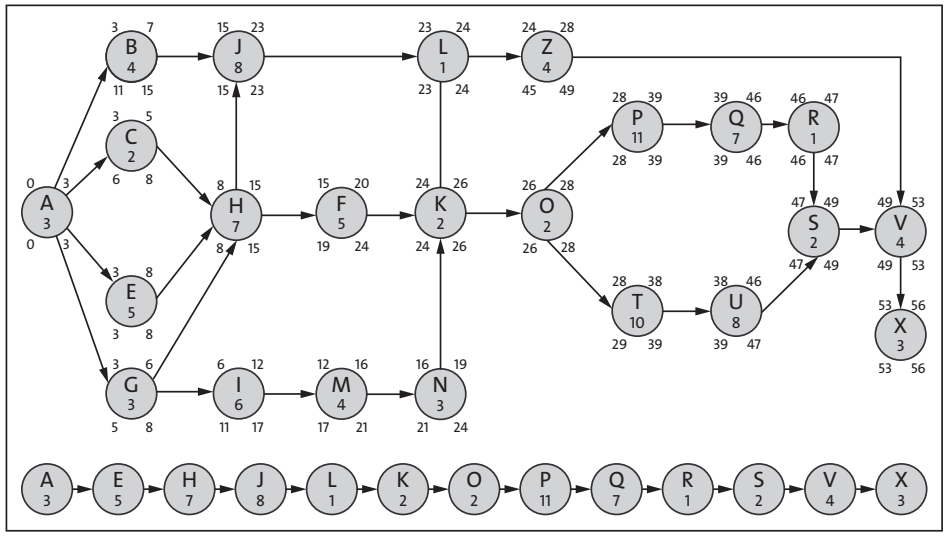

Abbildung 17.1 Abbildung verschiedener Aufgaben eines Projekts und des kritischen Pfades

Die Idee der Programmierung klingt sehr gut: Es wird zunächst einmal in einem »Vorwärtslauf« ermittelt, wo der Hauptbaum des Projekts liegt, also welche Folge von Aufgaben die längste Ausführungszeit benötigt. Hat man diese Endaufgabe identifiziert, wird über einen »Rückwärtslauf« der kritische Pfad so ermittelt, dass, ausgehend von der letzten Aufgabe, die Vorgängeraufgabe gesucht wird, die zeitlich als letzte beendet wurde. Das Ergebnis dieser Analyse ist der kritische Pfad (Abbildung 17.1. unten), also die Folge von Aufgaben, die die Gesamtdauer des Projekts bestimmt hat.

Vorarbeiten: Erstellung des Datenmodells

Ich möchte hier die SQL-Anweisungen zur Erstellung des Datenmodells nicht einzeln abdrucken, Sie können sie im Skript zum Buch nachlesen. Allerdings: Die Herausforderung liegt darin, dass eine Aufgabe Abhängigkeiten zu mehreren vorhergehenden Aufgaben haben kann. Daher scheidet ein einfaches Datenmodell aus, das lediglich zwei Spalten für die Vater-Kind-Beziehung vorsieht. Stattdessen verwende ich eine einfache Erweiterung, die die Hierarchie in eine eigene Tabelle auslagert, wie dies in Abbildung 17.2 zu sehen ist. Inhaltlich haben wir durch die zweite Tabelle eine `m:n`-Beziehung gestaltet, die allerdings etwas ungewöhnlich aussieht, da die beiden Tabellen, die normalerweise die Attribute der Beziehung enthalten, zu einer Tabelle `PROCESS` verschmolzen sind, da die Beziehung zwischen Prozessen modelliert werden soll. Sie kennen das Verfahren aus der Tabelle `EMP_MANAGERS`.

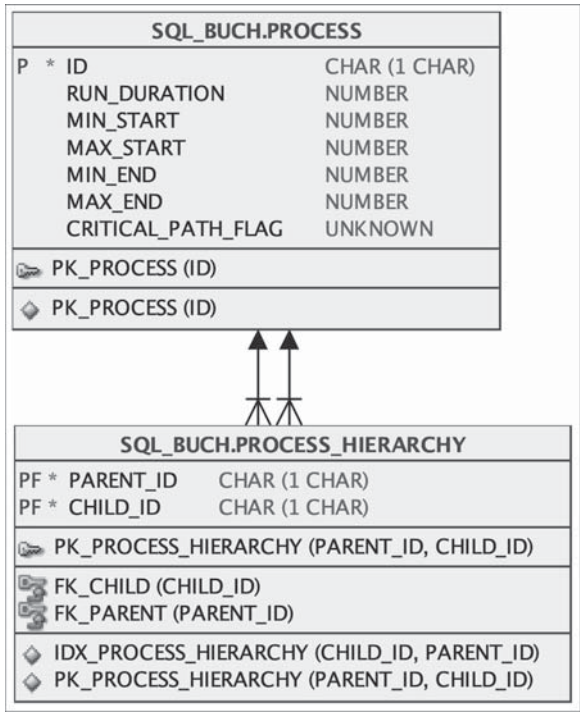

Abbildung 17.2 Datenmodell für den kritischen Pfad

Durch diese Erweiterung ist es möglich, eine Aufgabe mehrfach in der Tabelle `PROCESS_HIERARCHY` aufzuführen, jeweils mit einem anderen Kind- oder Vaterelement. So kann also eine Aufgabe mehrere Vorgängeraufgaben haben, ebenso wie eine Aufgabe mehrere Nachfolgeaufgaben anstoßen kann. Die nähere Beschreibung zu einer Aufgabe ist in der Tabelle `PROCESS` gespeichert. Diese beiden Tabellen enthalten alle Punkte mit den gleichen Ausführungszeiten und Abhängigkeiten.

Um unser Problem, die Ermittlung des kritischen Pfades, zu lösen, gehen wir in mehreren Teilschritten vor.

Ermittlung des kritischen Pfades

Wir beginnen damit, mittels einer hierarchischen Abfrage die möglichen Pfade der Aufgaben des Projekts darzustellen. Dieser Teil ist eine normale hierarchische Abfrage, wie Sie sie bereits mehrfach gesehen haben. Für diese Auswertung reicht mir der Zugriff auf die Tabelle PROCESS_HIERARCHY, denn es geht ja nur um die möglichen Pfade durch die Hierarchie in Listing 17.13. Bemerkenswert ist vielleicht die Erweiterung der Pfadangabe um das aktuelle Element. Das ist nötig, damit der Pfad »komplett« ist, was die spätere Berechnung erleichtert.

```
SQL>  select parent_id, child_id,
  2          sys_connect_by_path(parent_id, '|')
  3            || '|' || child_id || '|' path
  4     from process_hierarchy ph
  5  connect by prior ph.child_id = ph.parent_id
  6    start with ph.parent_id = 'A';

P C PATH
- - --------------------------------
A B |A|B|
B J |A|B|J|
J L |A|B|J|L|
L K |A|B|J|L|K|
K O |A|B|J|L|K|O|
O P |A|B|J|L|K|O|P|
P Q |A|B|J|L|K|O|P|Q|
Q R |A|B|J|L|K|O|P|Q|R|
…
V X |A|C|H|F|K|O|P|Q|R|S|V|X|
R S |A|G|I|M|N|K|O|P|Q|R|S|
S V |A|G|I|M|N|K|O|P|Q|R|S|V|
V X |A|G|I|M|N|K|O|P|Q|R|S|V|X|
O T |A|G|I|M|N|K|O|T|
T U |A|G|I|M|N|K|O|T|U|
U S |A|G|I|M|N|K|O|T|U|S|
S V |A|G|I|M|N|K|O|T|U|S|V|
V X |A|G|I|M|N|K|O|T|U|S|V|X|
137 Zeilen ausgewählt.
```

Listing 17.13 Die Darstellung aller möglichen hierarchischen Pfade

Nun folgt ein kleiner Trick: Der kritische Pfad ist definitionsgemäß der Pfad, dessen Summe der einzelnen Ausführungszeiten maximal ist. Daher müssen wir zwei Dinge tun: Zum einen müssen wir die Ausführungszeiten aller Aufgaben eines Pfades summieren und zum anderen den Pfad finden, dessen Ausführungszeit maximal ist. Dieser zweite Schritt entspricht einer Top-N-Analyse und ist daher relativ einfach. Schwieriger ist: Wie können wir die Ausführungszeiten aller Teilaufgaben aufsummieren, die an einem Pfad beteiligt waren? Hier greift der »Trick«: Wir summieren die Ausführungszeiten all der Aufgaben, die in der Spalte `PATH` enthalten sind. Um dies zu erreichen, erstellen wir einen Join auf die Tabelle `PROCESS` über eine `instr`-Funktion. Die Join-Bedingung ist auch die Begründung dafür, dass jeder Pfad auf einem Pipe-Zeichen endet. Hätten wir sie nicht, könnten falsche positive Matches auftauchen.

```
SQL> with data as (
  2         select parent_id, child_id,
  3                sys_connect_by_path(parent_id, '|') ||
  4                  '|' || child_id || '|' path
  5           from process_hierarchy ph
  6        connect by prior ph.child_id = ph.parent_id
  7          start with ph.parent_id = 'A')
  8  select path, sum(run_duration) sum_duration
  9    from data
 10    join process
 11      on instr(path, '|' || id || '|') > 0
 12   group by path
 13   order by sum(run_duration) desc
 14   fetch first 1 row only;

PATH                           SUM_DURATION
------------------------------ ------------
|A|E|H|J|L|K|O|P|Q|R|S|V|X|              56
```

Listing 17.14 Die endgültige Abfrage: Ermittlung des kritischen Pfades

Das Ergebnis entspricht dem des Blogs. Die Umsetzung in 14 Zeilen SQL halte ich persönlich für absolut machbar, daher würde ich diesen Weg bevorzugen, würde ich nicht die entscheidenden Limitierungen dieses Verfahrens kennen: Die Funktion `sys_connect_by_path` kann maximal 4.000 Byte zurückliefern, längere Pfade sind ausgeschlossen. Sicher können Sie diese Grenze eine Zeit lang aufschieben, wenn Sie möglichst kurze Schlüssel für Ihre Aufgaben verwenden, aber Sie sehen das grundsätzliche Problem. Auch die Berechnung der minimalen oder maximalen Start- bzw. Endzeit ist mit dieser Anweisung noch nicht gelöst. Doch wenn diese Limitierungen kein Hindernis für Sie darstellen, ist dieser Weg sicher elegant. Ich werde diese Aufga-

benstellung bei der Besprechung der ISO-kompatiblen Schreibweise hierarchischer Abfragen noch einmal aufgreifen.

17.3.5 Ein weiteres Beispiel

Meine Frau konfrontierte mich einmal mit einer interessanten Herausforderung, bei der man wohl nicht als Erstes an ein hierarchisches Problem denken würde. Es sollten Daten aus zwei Datenbanken verglichen werden, und zwar Leistungen, die über einen mehrtägigen Zeitraum erbracht wurden und voneinander unabhängig in zwei Systemen dokumentiert werden mussten. Das Problem: Das eine System speicherte die Informationen einer mehrtägigen Leistung in einer Zeile ab, ein Beginn- und Enddatum wurde vergeben. Das zweite System jedoch speicherte die Leistung in mehreren Zeilen ab, und zwar für jeden Tag, an dem die Leistung erbracht wurde. Die Leistungen selbst hatten aber keine ID, an der sie als zusammengehörig erkannt werden konnten, sondern nur eine Kundennummer mit der Information, für wen die Leistung erbracht wurde. Da allerdings für einen Kunden mehrere mehrtägige Leistungen erbracht werden konnten, konnte man mehrtägige Leistungen nur an der Tatsache erkennen, dass das Ende des vorangegangenen Tages 1 Sekunde vor dem Beginn des nächsten Intervalls lag, wie in den Beispieldaten aus Listing 17.15. Die `RANGE_ID` entspricht der Kundennummer, die Intervalle verschiedener Kunden können überlappen, für einen Kunden können mehrere Leistungen erbracht werden.

```
SQL> select *
  2    from range_test
  3   order by range_id, range_from

RANGE_ID RANGE_FROM          RANGE_TO
-------- ------------------- -------------------
     123 02.02.2024 15:30:00 02.02.2024 23:59:59
     123 03.02.2024 00:00:00 03.02.2024 23:59:59
     123 04.02.2024 00:00:00 04.02.2024 17:30:00
     123 05.02.2024 09:25:00 05.02.2024 23:59:59
     123 06.02.2024 00:00:00 06.02.2024 23:59:59
     123 07.02.2024 00:00:00 07.02.2024 17:22:15
     234 06.02.2024 19:45:00 06.02.2024 23:59:59
     234 07.02.2024 00:00:00 07.02.2024 23:59:59
     234 08.02.2024 00:00:00 08.02.2024 07:50:33
```

Listing 17.15 Die Beispieldaten

Wie können wir dafür sorgen, dass die Daten in eine Form überführt werden, die für eine mehrtägige Leistung nur eine Zeile anzeigt und den ersten Beginn sowie das letzte Ende anzeigt? Versuchen Sie es gern einmal mit Gruppenfunktionen, aber wie

gruppieren Sie hier? Welche Intervalle hängen miteinander zusammen, welche nicht? `connect by` to the rescue!

Listing 17.16 zeigt die Ermittlung der zusammengehörigen Intervalle. Der Einstieg in unsere Baumstruktur wird darüber ermittelt, dass das Startdatum `> 0` Uhr ist, während die Intervalle zusammengehören, wenn sowohl die `RANGE_ID` überreinstimmt als auch die Differenz von End- und Startdatum nur eine Sekunde beträgt. Um die Grenzen der Leistungen zu ermitteln, nutzen wir die Funktion `connect_by_isleaf`, denn wenn ein Blatt vorliegt, ist dies das letzte Teilintervall. Da wir mittels der Funktion `sys_connect_by_root` stets Zugriff auf den Beginn des Intervalls haben, reicht es aus, die Zeile auszugeben, für die `connect_by_isleaf = 1` ist, um die Gesamtdauer zu ermitteln.

```
SQL>  select range_id,
  2          connect_by_root(range_from) total_from,
  3          range_to total_to
  4     from range_test r
  5    where connect_by_isleaf = 1
  6    start with range_from - trunc(range_from) > 0
  7  connect by prior range_id = range_id
  8      and prior range_to + interval '1' second = range_from;

RANGE_ID TOTAL_FROM          TOTAL_TO
-------- ------------------- -------------------
     123 02.02.2024 15:30:00 04.02.2024 17:30:00
     123 05.02.2024 09:25:00 07.02.2024 17:22:15
     234 06.02.2024 19:45:00 08.02.2024 07:50:33
```

Listing 17.16 Die hierarchische Abfrage zum »Flachklopfen« der Intervalle

Das Schwierige an der Abfrage ist wahrscheinlich nur, auf die hierarchische Abfrage als Lösungsstrategie zu kommen ... Ich werde dieses Thema in Kapitel 21, »Row Pattern Matching«, noch einmal aufgreifen und Ihnen eine alternative und robustere Lösung vorstellen.

17.4 Hierarchische Abfragen nach ISO-Standard

Der Nachteil der Abfrage `connect by`: Diese Abfrageform ist Oracle-proprietär und kann so in anderen Datenbanken nicht verwendet werden. Alternativ unterstützt Oracle auch den offiziellen ISO-SQL-Standard für hierarchische Abfragen. Die Idee des Standards ist, hierarchische Abfragen über einen rekursiven Aufruf in der `with`-Klausel zu lösen. Na, herzlichen Dank, werden Sie sagen, was soll das denn sein?

Merke

Unter *Rekursion* verstehen wir, dass sich eine Funktion selbst aufruft, um ein Problem zu lösen. Dieses nicht sehr intuitive Verfahren lässt sich vielleicht an der Fakultätsfunktion erklären. `3!` (sprich 3 Fakultät) ist definiert als `3 * 2 * 1`. Hätten wir also eine Funktion `fak(n)`, könnte diese Funktion das Ergebnis berechnen, indem sie den übergebenen Wert mit dem Ergebnis der Fakultätsfunktion mit um `1` kleiner werdendem `n` aufruft, bis `n = 1` ist. Als Beispiel:

```
fak(3) = 3 * fak(2) = 3 * 2 * fak(1) = 3 * 2 * 1 = 6
```

Nicht nur Sie fragen sich, warum man so etwas tun sollte, doch wird dieses Verfahren häufig eingesetzt, wenn wir einfach vorab nicht wissen, wie oft eine Funktion sich selbst aufrufen muss. Das könnte dadurch gegeben sein, dass die Bedingung, die dazu führt, dass der rekursive Aufruf nicht mehr weitergeführt wird, erst bei der Analyse innerhalb der Funktion selbst ans Tageslicht kommt. Und genau diese Situation haben wir bei Hierarchien: Wir bearbeiten die Kindelemente rekursiv so lange weiter, bis eben keine weiteren Kinder mehr existieren. Da wir aber vorab nicht wissen, wie viele Ebenen existieren, bietet sich das Verfahren der Rekursion an.

17.4.1 Grundform

Die Umsetzung einer rekursiven Abfrage in SQL wird als harmonisierte Unterabfrage einer Abfrage `union all` durchgeführt, die in der `with`-Klausel der Anweisung notiert wird. Schön, wenn Sie diesen Satz direkt verstanden haben, noch schöner, wenn Sie sich sagen: Klar, hätte ich auch so gemacht … Für alle anderen ist aber wohl eine Erläuterung angebracht: Sie wissen, dass wir mit der `with`-Klausel die Definition einer Inner View aus der eigentlichen Abfrage auslagern können. Listing 17.17 zeigt diese Möglichkeit, den rekursiven Aufruf unserer hierarchischen Abfrage zu realisieren.

```
SQL> with hierarchie (lvl, emp_last_name, emp_id) as (
  2         select 1 lvl, emp_last_name, emp_id
  3           from hr_employees
  4          where emp_emp_id is null
  5         union all (
  6         select h.lvl + 1, e.emp_last_name, e.emp_id
  7           from hierarchie h
  8           join hr_employees e
  8             on h.emp_id = e.emp_emp_id))
  9  select lvl, emp_last_name
 10    from hierarchie;
```

```
LVL EMP_LAST_NAME
--- -------------
  1 King
  2 Hartstein
  2 Kochhar
  2 De Haan
  2 Raphaely
  2 Weiss
  2 Fripp
  2 Kaufling
…
```

Listing 17.17 ISO-SQL-kompatible hierarchische Abfrage

Schon diese einfache Abfrage ist ein ziemlicher Brocken. Sehen wir uns die Bestandteile an. Zunächst ist leicht erkennbar, dass die Musik in der `with`-Klausel spielt. Dort definieren wir eine Inner View mit dem Namen `HIERARCHIE`. Diese Inner View besteht aus zwei SQL-Anweisungen, die über eine Klausel `union all` miteinander verbunden sind.

Ungewöhnlich ist auch, dass eine Liste mit Spaltenaliassen definiert werden muss, über die die Spalten der Inner View später angesprochen werden sollen. Diese Liste wird der eigentlichen SQL-Anweisung in Klammern vorangestellt. So etwas können Sie auch in »normalen« `with`-Klauseln oder auch bei der Definition einer View machen, es ist aber nicht sehr weit verbreitet und daher vielleicht etwas seltsam zu sehen. Im Zusammenhang mit der hierarchischen Abfrage ist diese Spaltenliste Pflicht.

Der Startpunkt unserer hierarchischen Abfrage wird durch den ersten Teil der Abfrage definiert, die lediglich die Zeile definiert, bei der die Hierarchie beginnen soll. In unserem Fall ist das die Zeile mit allen Mitarbeitern, die keine Vorgesetzten haben, wie wir das bereits aus der `connect by`-Klausel kennen. Der zweite Teil der `union all`-Abfrage ist zwar für sich genommen auch nicht schwer, nur verwirrend, denn in dieser Abfrage wird ein Join auf die Inner View verwendet, die wir gerade definieren, nämlich `HIERARCHIE`. Das ist doch schon sehr merkwürdig.

Sie können sich das vielleicht so vorstellen: Wir starten mit der Zeile `King`. Diese Zeile wird durch die erste Abfrage geliefert, und zwar nur diese Zeile, weil der `union all`-Teil der Abfrage `NULL` geliefert hat, denn beim ersten Aufruf gibt es noch keinen Kontext, auf den wir uns beziehen können, `h.emp_id` ist daher `NULL`. Nun gehen wir im zweiten Durchlauf auf die gerade ermittelte Zeile `King` (die sich in `HIERARCHIE` befindet) und suchen hierzu alle Untergebenen, für die deren Spalte `EMP_EMP_ID` gleich der bereits ausgewählten Spalte `EMP_ID` ist. Für jede Zeile, die sich dort ergibt, ruft sich die Abfrage rekursiv auf; wir gehen wiederum auf die gleiche Inner View `HIERARCHIE`, diesmal aber mit den in der zweiten Teilabfrage ermittelten Zeilen. Solange neue

Kindelemente gefunden werden, werden diese wiederum dafür sorgen, dass eine weitere Suche nach deren Kindern durchgeführt wird. Erst wenn die zweite Teilabfrage keine Daten mehr liefert, wird die Rekursion beendet.

Dass dies so funktioniert, erkennen Sie im Übrigen auch an einer Spalte, die wir händisch berechnen lassen: der Spalte `LVL`. Wir beginnen in der ersten Teilabfrage mit `LVL = 1`. Danach zählen wir in jedem Aufruf der zweiten Teilabfrage den Zähler um `1` hoch und erhalten damit die Schachtelungstiefe der rekursiven Aufrufe. In der Ausgabe enthält diese Spalte die gleichen Daten wie die Pseudospalte `LEVEL` aus der Oracle-Schreibweise, deren Namen ich allerdings nicht verwenden darf, da es sich um einen von Oracle geschützten Bezeichner handelt.

Noch ist allerdings nicht alles gleich wie beim Pendant von Oracle. Insbesondere stört, dass die Abfrage zunächst alle Mitarbeiter auf Ebene `1`, dann auf Ebene `2` usw. auswertet und anzeigt. Es mag ja sein, dass die Ebenen korrekt zugeordnet sind, aber so erkennen wir keine Struktur. Was wir benötigen, ist ein Verfahren, das dem Baum an jedem Ast zunächst einmal bis zu den Blättern folgt und erst anschließend den nächsten Ast bearbeitet.

Da wir keine Einstellungen vorgenommen haben, um dieses Verhalten zu erzeugen, ist dies das Standardverfahren, das als `breadth first` bezeichnet wird. Listing 17.18 ändert das Standardverhalten zu `depth first`, wodurch zu einem Eintrag zunächst alle Kindelemente, deren Kindelemente und so weiter ermittelt werden. Beachten Sie, dass die Angabe der Spalte, mittels derer die Klausel definiert wird, eindeutig sein muss, um keine falschen Beziehungen zu erstellen. Daher verwende ich hier die Spalte `EMP_ID`.

Mittels der Erweiterung `set` kann die durch die Klausel ermittelte Reihenfolge in einer zu benennenden Spalte (`SOPRT_SEQ`) gespeichert und später als Sortierkriterium verwendet werden. Dies entspricht inhaltlich der Klausel `siblings` in der `connect by`-Abfrage und erzeugt ein Ergebnis, wie wir es bereits aus der `connect-by`-Abfrage kennen.

```
SQL> with hierarchie (lvl, emp_last_name, emp_id) as (
  2         select 1 lvl, emp_last_name, emp_id
  3           from hr_employees
  4          where emp_emp_id is null
  5         union all (
  6         select h.lvl + 1, e.emp_last_name, e.emp_id
  7           from hierarchie h
  8           join hr_employees e
  9             on h.emp_id = e.emp_emp_id))
 10  search depth first by emp_id set sort_seq
 11  select lvl,
 12         lpad('.', (lvl - 1) * 2, '.') || emp_last_name emp_last_name
 13    from hierarchie
```

```
 14   order by sort_seq;

 LVL EMP_LAST_NAME
---- --------------------
   1 King
   2 ..Cambrault
   3 ....Bates
   3 ....Bloom
   3 ....Fox
   3 ....Kumar
   3 ....Ozer
   3 ....Smith
   2 ..De Haan
   3 ....Hunold
   4 ......Austin
   4 ......Ernst
...
```

Listing 17.18 Auswertung wie bei CONNECT BY

Was soll man sagen, die Oracle-proprietäre Schreibweise sieht im Vergleich extrem elegant aus … Nun könnte die neue Schreibweise dennoch empfehlenswert sein, wenn sie denn massive Vorteile böte. Einerseits, und das ist sicher ein Argument, ist diese Schreibweise standardkonform. Ob dieses Argument allerdings sticht, muss sich zeigen. Nicht alle SQL-Anweisungen müssen direkt gegen zehn verschiedene Datenbanken funktionieren, zumal nicht sichergestellt ist, dass diese zehn auch alle diese Form der Standardnotation beherrschen (Oracle kann das auch erst ab Version 11.2). Andererseits könnte es sein, dass wir mit dieser Schreibweise Limitierungen der alten Schreibweise umgehen könnten. Doch ist, bei Licht besehen, eigentlich das Gegenteil der Fall: Wir müssen erst einmal nachweisen, dass die Funktionalität der Schreibweise `connect by` auch in dieser Schreibweise möglich ist.

17.4.2 Erweiterungen

Sie haben gesehen, dass wir die Sortierung bereits mit der Grundform »erschlagen« haben. Daher fehlen uns noch die Pseudospalten `connect_by_isleaf` und `connect_by_iscycle`, der Operator `connect_by_root` sowie die Funktion `sys_connect_by_path`. Listing 17.19 zeigt die Implementierung der beiden letzten Teilaufgaben.

Den Pfad können wir im Rahmen der Rekursion einfach aufbauen, das geht sehr elegant und schafft zudem Raum für weitere Optionen, die wir mit dem Pendant der Abfrage `connect by` nicht hätten. Verbuchen wir das als Plus, ebenso wie die einfache Referenz auf eine Spalte `HIERARCHIE`, um uns Informationen aus der Wurzel des Baums zu besorgen. Diese beiden Punkte gehen an die neue Schreibweise.

```
SQL> with hierarchie (lvl, emp_last_name, emp_id, pfad, root) as (
  2         select 1 lvl, emp_last_name, emp_id,
  3                '/' || emp_last_name pfad, emp_last_name root
  4           from hr_employees
  5          where emp_emp_id is null
  6         union all
  7         select h.lvl + 1, e.emp_last_name, e.emp_id,
  8                h.pfad || '/' || e.emp_last_name, h.root
  9           from hierarchie h
 10           join hr_employees e
 11             on h.emp_id = e.emp_emp_id)
 12  search depth first by emp_last_name set sort_seq
 13  select lvl, emp_last_name, pfad, root
 14    from hierarchie
 15   order by sort_seq;

LVL EMP_LAST_NAME PFAD                    ROOT
--- ------------- --------------------- ----
  1 King          /King                 King
  2 Cambrault     /King/Cambrault       King
  3 Bates         /King/Cambrault/Bates King
  3 Bloom         /King/Cambrault/Bloom King
  3 Fox           /King/Cambrault/Fox   King
  3 Kumar         /King/Cambrault/Kumar King
  3 Ozer          /King/Cambrault/Ozer  King
  3 Smith         /King/Cambrault/Smith King
  2 De Haan       /King/De Haan         King
...
```

Listing 17.19 Simulation der Funktionen SYS_CONNECT_BY_PATH und CONNECT_BY_ROOT

Auch die Funktion `connect_by_iscycle` lässt sich darstellen, wie Listing 17.20 zeigt, allerdings benötigen wir hierfür zweierlei: Zum einen muss ich den Mitarbeiter `King` wieder einem anderen Mitarbeiter unterstellen, zum anderen verwenden wir, analog zur Klausel `nocycle`, eine `cycle`-Klausel mit weiteren Einstellungsmöglichkeiten, die sich weitgehend selbst erklären. Erstaunlicherweise werden aber die falschen Daten ausgegeben, nämlich der Knoten, der den Zirkelbezug verursacht, anstatt des Vorgesetzten von `King`. Dieses Verhalten habe ich nicht korrigieren können.

```
SQL> update hr_employees
  2     set emp_emp_id = 106 -- Pataballa
  3   where emp_emp_id is null;
1 Zeile wurde aktualisiert.
```

```
SQL> with hierarchie (lvl, emp_last_name, emp_id) as (
  2       select 1 lvl, emp_last_name, emp_id
  3         from hr_employees
  4        where emp_id = 100
  5        union all
  6       select h.lvl + 1, e.emp_last_name, e.emp_id
  7         from hierarchie h
  8         join hr_employees e
  9           on h.emp_id = e.emp_emp_id)
 10  search depth first by emp_id set sort_seq
 11   cycle emp_id set is_cycle to 1 default 0
 12  select lvl, emp_last_name, is_cycle
 13    from hierarchie
 14   order by sort_seq;

LVL EMP_LAST_NAME IS_CYLCE
--- ------------- --------
  1 King                 0
  2 Kochhar              0
  3 Greenberg            0
  4 Faviet               0
  4 Chen                 0
…
  4 Pataballa            0
  5 King                 1
…

SQL> rollback;
```

Listing 17.20 Darstellung der Funktion CONNECT_BY_ISCYCLE

Was allerdings nur mit Klimmzügen geht, ist die wichtige Option `connect_by_isleaf`. Das Problem ist verständlich: Ob ein Knoten ein Blatt ist, erfahren wir erst im nächsten Rekursionslauf, also eigentlich zu spät. Wir wollen ja wissen, ob wir Kinder haben oder nicht, aufgrund der Natur der Rekursion wird dies aber lediglich dadurch ausgedrückt, dass es keine weitere Rekursion gibt. In der aktuellen Implementierung liegt keine Klausel vor, mit der wir diese Information erfragen können, es bleibt nur die nachfolgende Analyse. Dies könnte, wie in einem Blog von *Lucas Jellma* vorgeschlagen wird, dadurch simuliert werden, dass bei einer speziellen Sortierung – bei der Abfrageart `depth first` und gleichzeitiger Sortierung – nur die Elemente als Knoten bezeichnet, deren nachfolgendes Element eine größere Schachtelungstiefe hat. Viel Wenn und Aber für meinen Geschmack. Sehen wir uns diesen Fall einmal an. Es ist zumindest eine schöne Verwendung einer analytischen Funktion, denn wir müssen auf die Folgezeile einer Abfrage sehen:

```
SQL> with hierarchie (lvl, emp_last_name, emp_id) as (
  2       select 1 lvl, emp_last_name, emp_id
  3         from hr_employees
  4        where emp_emp_id is null
  5        union all
  6       select h.lvl + 1, e.emp_last_name, e.emp_id
  7         from hierarchie h
  8         join hr_employees e
  9           on h.emp_id = e.emp_emp_id)
 10  search depth first by emp_last_name set sort_seq
 11  select lvl, emp_last_name, sort_seq,
 12         case
 13         when lvl - lead(lvl) over (order by sort_seq) < 0
 14         then 0 else 1 end is_leaf
 15    from hierarchie
 16   order by sort_seq;

LVL EMP_LAST_NAME SORT_SEQ IS_LEAF
--- ------------- -------- -------
  1 King                 1       0
  2 Cambrault            2       0
  3 Bates                3       1
  3 Bloom                4       1
  3 Fox                  5       1
  3 Kumar                6       1
  3 Ozer                 7       1
  3 Smith                8       1
  2 De Haan              9       0
  3 Hunold              10       0
  4 Austin              11       1
  4 Ernst               12       1
  4 Lorentz             13       1
  4 Pataballa           14       1
…
```

Listing 17.21 Simulation der Pseudospalte CONNECT_BY_ISLEAF

Es erfordert ein wenig Mühe, diesem Weg zu folgen, aber eigentlich ist das Ergebnis klar: Wenn die Folgezeile der aktuellen Zeile eine größere Schachtelungstiefe hat, muss es sich bei der Folgezeile um ein Kind handeln, daher ist die aktuelle Zeile ein Knoten. Für mich bleibt ein fader Nachgeschmack in zweierlei Hinsicht: Zum einen ist diese Schreibweise komplizierter als die Pseudospalte, zum anderen ist die Funktion von der korrekten Abfragetechnik und Sortierung abhängig. Das ist mir zu

wackelig. Eine stabilere Implementierung dieser Funktion erreichen wir, wenn wir aus Sicht der übergeordneten Zeile nachsehen, ob es Kinder gibt. Das können wir zum Beispiel mit einer skalaren Unterabfrage erreichen, wie in der folgenden Abfrage in Listing 17.22.

```
SQL> with hierarchie (lvl, emp_last_name, emp_id, is_leaf) as (
  2       select 1 lvl, emp_last_name, emp_id,
  3              (select decode(count(*), 0, 1, 0)
  4                 from hr_employees
  5                where emp_emp_id = m.emp_id) is_leaf
  6         from hr_employees m
  7        where emp_emp_id is null
  8        union all
  9       select h.lvl + 1, e.emp_last_name, e.emp_id,
 10              (select decode(count(*), 0, 1, 0)
 11                 from hr_employees
 12                where emp_emp_id = e.emp_id) is_leaf
 13         from hierarchie h
 14         join hr_employees e
 15           on h.emp_id = e.emp_emp_id)
 16  search depth first by emp_last_name set sort_seq
 17  select lvl, emp_last_name, is_leaf
 18    from hierarchie;

LVL EMP_LAST_NAME IS_LEAF
--- ------------- -------
  1 King                0
  2 Cambrault           0
  3 Bates               1
  3 Bloom               1
  3 Fox                 1
  3 Kumar               1
  3 Ozer                1
  3 Smith               1
  2 De Haan             0
  3 Hunold              0
  4 Austin              1
  4 Ernst               1
  4 Lorentz             1
  4 Pataballa           1
…
```

Listing 17.22 Simulation der Pseudospalte CONNECT_BY_ISLEAF mit skalarer Unterabfrage

Dieses Beispiel einer Abfrage zeige ich Ihnen als Anwendung einer skalaren Unterabfrage, weniger als Empfehlung. Ich möchte solche Konstruktionen nicht jedes Mal schreiben müssen, nur um eine hierarchische Abfrage zu erzielen. Ich bleibe da konservativ und sage, dass mir die eingebaute Funktionalität besser gefällt, für mich ist sie intuitiver. Vielleicht bessert sich die Situation mit weiteren Releases der Oracle-Datenbank, allerdings müssen Sie berücksichtigen, dass eine Erweiterung dieser Funktion zunächst im ISO-SQL-Standard vorgenommen werden sollte, denn nur aus Gründen der Kompatibilität mit dem Standard gibt es diese Notation bei Oracle ja überhaupt. Eine einseitige Erweiterung des Standards durch Oracle ergibt also keinen Sinn.

Fachlich ist zur Abfrage in Listing 17.22 zu sagen, dass die skalare Unterabfrage als harmonisierte Unterabfrage auftritt und für jede Zeile die Anzahl der Zeilen zählt, in denen die aktuelle Mitarbeiternummer als Managernummer auftritt. Wird kein Kind gefunden, liefert die `decode`-Anweisung, die ich hier als Ersatz einer `case`-Anweisung wegen ihrer Kürze nutze, eine `1` zurück, ansonsten eine `0`. Sie könnten berechtigterweise einwenden, dass mich nicht die genaue Anzahl der abhängigen Zeilen interessiert, sondern lediglich die Tatsache, dass ein Kind existiert und daher eine `exists`-Abfrage besser geeignet wäre, dieses Problem zu lösen, und ich stimme zu. Allerdings würde die Abfrage dann noch einmal wesentlich komplexer, das wollte ich mir und Ihnen eigentlich nicht zumuten. Wenn Sie das einmal als Übung machen wollen, gern.

Ein weiteres Argument für die `connect by`-Klausel ergibt sich, wenn wir auf die Ausführungspläne schauen, denn die seit Urzeiten bekannte `connect by`-Abfrage wird besser optimiert als die rekursive Unterabfrage. Der Vergleich stammt aus Version 23ai, ist also immer noch aktuell. Es kommt hinzu, dass die `connect by`-Abfrage 5 Zeilen umfasst, während die rekursive `with`-Abfrage auf immerhin 17 Zeilen kommt.

```
-- rekursive WITH-Klausel
Ausführungsplan
----------------------------------------------------------------------------
| Id  | Operation                          | Name             | Cost (%CPU)|
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                  | 18924   (9)|
|   1 |  VIEW                              |                  | 18924   (9)|
|   2 |   UNION ALL (RECURSIVE WITH) DEPTH F|                 |            |
|   3 |    SORT AGGREGATE                  |                  |            |
|*  4 |     INDEX RANGE SCAN               | EMP_EMP_ID_FK_IX |     1   (0)|
|*  5 |    TABLE ACCESS FULL               | HR_EMPLOYEES     |     3   (0)|
|   6 |    SORT AGGREGATE                  |                  |            |
|*  7 |     INDEX RANGE SCAN               | EMP_EMP_ID_FK_IX |     1   (0)|
|*  8 |    HASH JOIN                       |                  |  3335   (2)|
```

```
|   9 |     BUFFER SORT (REUSE)            |                |           |
|  10 |      TABLE ACCESS FULL             | HR_EMPLOYEES   |     3   (0)|
|  11 |     RECURSIVE WITH PUMP            |                |           |
-------------------------------------------------------------------------

-- CONNECT BY-Abfrage
Ausführungsplan
-------------------------------------------------------------------------
| Id  | Operation                          | Name         | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |              |     4  (25)|
|*  1 |  CONNECT BY NO FILTERING WITH START-WITH|           |           |
|   2 |   TABLE ACCESS FULL                | HR_EMPLOYEES |     3   (0)|
-------------------------------------------------------------------------
```

Listing 17.23 Vergleich der Ausführungspläne

17.4.3 Anwendungsbeispiel reloaded: Kritischer Pfad

Um den negativen Beigeschmack dieser Abfragestrategie etwas zu lindern, möchte ich noch einmal auf die Berechnung des kritischen Pfades zurückkommen. Wir hatten in der ersten Lösung zwei Nachteile aus Listing 17.14 zu akzeptieren:

- Es ist eine Summierung mittels eines `instr`-Vergleichs gegen eine Knotenliste erforderlich.
- Die Maximallänge der Knotenliste beträgt 4.000 Byte.

Können wir eventuell beide Probleme mit der rekursiven `with`-Klausel lösen? Schließlich hatten wir die Möglichkeit, einfache Summen und Zeichenkettenlisten zu berechnen, als Vorteil vermerkt (zum Beispiel auch bei der Zusammenstellung der Pfadliste). Wie also sähe eine Lösung dieses Problems aus? Listing 17.24 zeigt die Abfrage. Es bleibt bei einer etwas länglichen Formulierung, die ich durch einen kleinen Trick etwas übersichtlicher gestalten wollte. Zunächst lege ich mir die Werte in einem Join zwischen den Tabellen `PROCESS` und `PROCESS_HIERARCHY` zurecht, denn ich kann in der nachfolgenden `with`-Abfrage die Summe der jeweiligen Ablaufströme direkt berechnen und benötige hierfür die Laufzeit. Als Startpunkt wähle ich Aufgabe `A`, und weil diese in der `PROCESS_HIERARCHY` nur als Elternelement auftaucht, verwende ich einen Outer Join aus der Blickrichtung `PROCESS`, damit alle Aufgaben mit Dauern verfügbar sind. In der nachfolgenden Teilabfrage hinter dem `union all` wird daraus sowohl der Pfad als auch die Gesamtdauer berechnet. Ist dies gemacht, geht es ganz schnell: Es wird nach der Länge sortiert und nur der Pfad mit der längsten Summe ausgewählt.

```
SQL> with process_values as (
  2         select id child_id, parent_id, run_duration
  3           from process
  4           left join process_hierarchy
  5             on id = child_id),
  6       pathes (child_id, path, sum_duration) as (
  7         select child_id,
  8                child_id path,
  9                run_duration sum_duration
 10           from process_values
 11          where parent_id is null
 12         union all
 13         select v.child_id,
 14                p.path || '|' || v.child_id path,
 15                p.sum_duration + v.run_duration sum_duration
 16           from pathes p
 17           join process_values v
 18             on v.parent_id = p.child_id)
 19  select path, sum_duration
 20    from pathes
 21   order by sum_duration desc
 22   fetch first 1 row only;

PATH                         SUM_DURATION
---------------------------- ------------
A|E|H|J|L|K|O|P|Q|R|S|V|X              56
```

Listing 17.24 Berechnung des kritischen Pfades mit rekursiver WITH-Klausel

Diese Form der Abfrage ist nicht durch die Länge des Pfades limitiert und mag dem ein oder anderen intuitiver erscheinen als dieser seltsame Join über eine `instr`-Funktion. Ich stimme zu und komme zu dem Schluss, dass es wie üblich ist: Die Aufgabe bestimmt die Werkzeuge, die eingesetzt werden sollten.