Kapitel 4
Datenbankobjekte und SQL

Nachdem nun das technische Umfeld bereitet und auch die Ebene zwischen der physikalischen Speicherung und dem Anwender besprochen worden sind, können wir uns den Datenbankobjekten zuwenden, die für den Entwickler die eigentlichen Berührungspunkte zur Datenbank darstellen. Zudem möchte ich ein Plädoyer für die Sprache SQL halten.

Als *Datenbankobjekte* werden alle Objekte verstanden, die einem Benutzer gehören können. Hinter diesem Sammelbegriff verbergen sich Tabellen, Indizes, temporäre Tabellen und materialisierte Sichten, aber auch Datenbank-Links, Sequenzen und viele andere Objekte. Wir werden uns diese Datenbankobjekte lediglich im Überblick ansehen, damit Sie die grundsätzliche Arbeitsweise verstehen; sollte ein umfassenderes Verständnis nötig werden, werde ich das an der entsprechenden Stelle nachholen.

4.1 Tabellen

Tabellen sind die Grundbestandteile einer Datenbank. In ihnen werden die Daten gespeichert, die für eine Anwendung benötigt werden. Tabellen liegen bei Oracle in mehreren Ausprägungen vor, von denen einige für die Administration wichtiger als für die Entwickler sind. Der häufigste Tabellentyp ist die *Heap Organized Table*, eine »normale« Datenbanktabelle. Zudem bietet Oracle noch die *Index Organzied Table*, die *Global Temporary Table* sowie die *partitionierte Tabelle* an.

4.1.1 Heap Organized Table

Normale Tabellen einer Datenbank sind *heap organized*. Das bedeutet, dass die Datenbank keine Sortierung der Daten irgendeiner Art garantiert. Alle Daten werden dorthin gespeichert, wo gerade Platz ist. Hat Oracle für Tabellen wirklich keine bessere Lösung? Oracle hat, aber die Heap Organized Table ist auch nicht so schlecht wie der erste Eindruck, den sie hinterlässt. Vergleichen wir dazu doch einmal eine Tabelle mit einem Lagerraum voller leerer Regale. Jeder Regalplatz hat eine fortlaufende und eindeutige Stellnummer, zudem hängt am ersten Regal eine rote Fahne, ansonsten ist der Lagerraum groß und leer. Nun kommt eine Palette Farbe, die Sie in das Lager

räumen müssen. Große Eimer, kleine Eimer, blaue Farbe, gelbe Farbe, Acryllack und Kunstharzlack, alles durcheinander. Wie wollen Sie hier Ordnung hineinbringen? Eines ist sicher: Egal, für welches Sortierkriterium Sie sich entscheiden, es wird nicht das richtige sein. Sortieren Sie nach Farbe, fragt jemand nach allen Kunstharzlacken. Sortieren Sie danach, fragt jemand nach allen 2,5-kg-Gebinden. Also warum nicht von vornherein auf eine Sortierung verzichten und alles so ins Lager einräumen, wie es kommt? Denn, und das kommt ja noch hinzu: Haben Sie alles schön nach Farbe sortiert, kommt eine neue Lieferung Gelb. Leider steht Gelb in der Mitte, eingerahmt von Rot und Blau. Räumen Sie jetzt das ganze Lager um, um Platz für die neuen gelben Eimer zu schaffen? Täten Sie das, täten Sie bald nichts anderes mehr. Sie räumen nun also alle Farbeimer in das Lager ein, nehmen sich allerdings vorher die rote Fahne mit und stecken sie an das letzte Regal, das nun Farbeimer enthält.

Stellen wir uns, um im Bild zu bleiben, nun vor, dass über die Zeit von den ursprünglich 1.000 Farbeimern 950 verwendet wurden. Sie haben nun keine einfache Möglichkeit zu erkennen, in welchem Regal nun noch ein Farbeimer steht und in welchem nicht. Nun könnten Sie den Platzverbrauch des Lagers dadurch optimieren, dass Sie alle verbliebenen Farbeimer an den Beginn des Lagers räumen. Sie werden in Abschnitt 4.2, »Index«, sehen, dass es auch harte technische Gründe gibt, so etwas nicht zu tun, wir können uns aber im Moment auch mit der Begründung davon abhalten, dies zu tun, dass diese Arbeit ja bei jeder Entnahme eines Farbeimers für das gesamte Lager durchgeführt werden müsste. So etwas bringt ebenso wenig wie das permanente Sortieren. Wenn nun aber neue Farbeimer in das Lager geräumt werden sollen, werden zunächst die freien Lagerplätze wiederverwendet, bevor neue, noch nicht benutzte Regalflächen belegt werden. Daher ist spätestens von nun an nicht mehr vorhersagbar, in welcher Reihenfolge die Farbeimer im Regal platziert wurden.

Wenn Sie also einen Farbeimer suchen, bleibt Ihnen nichts anderes übrig, als das gesamte Lager zu durchsuchen, denn auch ein Eimer, den Sie zuletzt eingeräumt haben, kann sehr weit vorn einen Platz gefunden haben. Allerdings brauchen Sie nur bis zu dem Regal zu suchen, an dem sich die rote Fahne befindet, denn diese ist am letzten *jemals* belegten Lagerplatz befestigt. Das Lager wurde ja auf Zuwachs gebaut. Doch noch sind nicht alle Regalmeter belegt worden. Als das Lager bislang maximal gefüllt war, wurden, sagen wir, 135 der 250 verfügbaren Regale benötigt. Am Ende von Regal 135 steckt jetzt die rote Fahne, die anzeigt, dass hinter ihr sicher keine Daten mehr zu finden sein werden. Diese Markierung benutzt die Datenbank, um ihre Suche nach Zeilen abzubrechen. Was sie bis hierhin nicht findet, gibt es in dieser Tabelle nicht. Um ein bisschen technischer zu werden: Die rote Fahne heißt bei Oracle *High Watermark* (HWM – auch ein schönes Bild: ein Maximalpegelmesser), und den Prozess der Suche bis zur HWM nennt Oracle einen *Full Table Scan*. Die Idee mit den Regalen ist so falsch auch nicht: Ein Regal besteht aus einzelnen, sagen wir, einen Meter breiten Teilregalen. Dies entspricht einem *Block*, der nun mehrere Zeilen

einer Tabelle aufnehmen kann. Viele Blöcke zusammen bilden ein sogenanntes *Extent* – also eine Einheit, die festlegt, in welchen Größenschritten eine Tabelle wächst. Die eindeutige Lagerplatznummer, die wir später noch verwenden werden, wird bei Oracle als *rowid* bezeichnet. In dieser Lagerplatznummer verbirgt sich nicht nur die konkrete Nummer des Lagerplatzes, sondern auch die Nummer des Blocks innerhalb der Datei, die Nummer der Datei innerhalb des Tablespaces und die interne ID der Tabelle.

Vielleicht noch diese Information: Das oben beschriebene Verhalten bezüglich der Entnahme von Farbeimern stimmt, wenn eine `delete`-Anweisung benutzt wird. Diese Anweisung ist der Standard bei einer produktiv laufenden Anwendung. Administratoren können darüber hinaus auch die Anweisung `truncate` verwenden. Diese Anweisung löscht den Speicherplatz der Tabelle auf der Festplatte und damit auch ausnahmslos alle Zeilen einer Tabelle. Durch eine `truncate`-Anweisung wird die Tabelle wieder auf ihre Startgröße gebracht und die HWM auf den ersten Block der Tabelle gesetzt. Zudem gibt es auch Möglichkeiten, eine Tabelle online reorganisieren zu lassen, doch sind dies administrative Aufgaben, die nicht in den Bereich der Entwicklung gehören.

4.1.2 Index Organized Table

Als Ergänzung zur Heap Organized Table gibt es bei Oracle bereits seit vielen Jahren auch die *Index Organized Table* (IOT). Ich weise hier auf diese lange Zeitdauer hin, weil dieser Tabellentyp in Datenmodellen extrem selten verwendet wird. Das ist eigentlich schade, in meinen Datenmodellen sehe ich bei 30–40 % der Tabellen eigentlich eine gute Verwendungsmöglichkeit für diesen Tabellentyp. Was unterscheidet die IOT von einer normalen Heap Organized Table? Sie garantiert eine Sortierung der Werte nach einer Schlüsselspalte. Stellen wir uns einmal ein normales Datenmodell vor. Insbesondere interessiert uns eine einfache `m:n`-Beziehung. Wir sehen drei Tabellen, die jeweils mit Primärschlüsseln gesichert (und zusätzlich noch indiziert) sind. Wie Sie in Abschnitt 5.3.1, »Datenintegrität«, noch sehen werden, haben Primärschlüssel immer einen Index zur Folge, der sicherstellt, dass die Daten in einer sortierten Weise gespeichert werden (zu Indizes siehe auch Abschnitt 4.2, »Index«). Eine IOT stellt nun die Kombination aus einem Index und einer Tabelle dar. Daten werden in einer IOT, wie gesagt, nach einem Sortierkriterium (und zwar immer nach einem Primärschlüssel) sortiert gespeichert und ersparen dadurch die externe Indizierung. Der große Vorteil dieser Tabellentypen besteht darin, dass die Datenbank nicht zwischen Index und Tabelle hin- und herspringen muss, sondern die Nutzdaten direkt sortiert vorfindet.

IOT können oftmals sehr sinnvoll eingesetzt werden. Hier sind zunächst die Rahmenbedingungen, die für den Einsatz einer IOT sprechen:

- Daten sind mit einem Primärschlüssel gesichert.
- Die Zeilenlänge ist nicht zu lang (das hängt von der Blockgröße ab, etwa 40 % der Blockgröße ist das Maximum für eine Zeile).
- Die Zugriffe auf diese Tabelle erfolgen fast immer über den Primärschlüssel und nicht (oft) über andere Suchkriterien.
- Die Daten sind nicht *zu* volatil, und es werden keine oder nur sehr wenige weitere Spalten indiziert.

  Der Grund dafür ist relativ komplex, es soll uns reichen, dass aufgrund der sortierten Speicherung die Zeilen öfter »umgeräumt« werden müssen und daher die Lagerplatznummer einer Zeile nicht mehr so konstant ist wie bei einer normalen Tabelle. Indizes auf IOT-Spalten sind dadurch nicht mehr so effizient.

Wenn eine IOT verwendet werden kann, sollte man das ernsthaft erwägen, denn IOTs sind deutlich performanter (Faustregel: ca. 20 % weniger CPU-Last) als die Kombination aus Tabelle und Index. Außerdem sind sie deutlich kleiner auf der Platte, denn es wird kein separater Index gespeichert. Richtig ist aber auch: Sie sind beim Schreiben langsamer als eine normale Heap Organized Table *ohne* Index oder Primärschlüssel.

4.1.3 Temporäre Tabellen

Etwas exotischer sind temporäre Tabellen. Diese Tabellen werden »normal« angelegt, enthalten aber normalerweise keine Daten, sondern sind lediglich als Struktur bekannt. Im Rahmen einer Datenbank-Session kann ein Benutzer Daten in temporäre Tabellen ablegen und sie dort manipulieren, löschen etc. wie in einer normalen Tabelle. Je nach Einstellung der Tabelle verliert sie jedoch bei der nächsten `commit`-Anweisung wieder alle Daten (das ist die Standardeinstellung) oder nicht (mithilfe der Klausel `on commit preserve rows`). Spätestens wenn die Session beendet wird, sind aber alle Daten aus dieser Tabelle gelöscht. Interessant ist, dass die Daten einer temporären Tabelle *privat* für die Session des jeweiligen Benutzers sind, der die Daten in die Tabelle eingefügt hat. Diese Tabellen sind sogar vor dem Administrator sicher: Selbst der `SYS`-Benutzer einer Datenbank hat keine Möglichkeit, die Daten zu lesen, die innerhalb einer Session eines anderen Benutzers in eine temporäre Tabelle geschrieben wurden. Bevor ich Ihnen einige Einsatzbereiche vorstelle, hier die Syntax zur Erzeugung:

```
create global temporary table my_temp(
  id number,
  value varchar2(40))
<on commit preserve rows>
```

Die Tabelle wird genauso genutzt wie jede andere Tabelle auch, also mit `insert`-Anweisungen gefüllt etc., und kann für eine Reihe von Zwecken genutzt werden. Hier sind einige Anwendungen, denen ich begegnet bin:

- Nebenrechnungen einer PL/SQL- oder SQL-Funktion können hier gespeichert und dann weiterverarbeitet werden.
- Session-relevante Daten können in einer solchen Tabelle vorgehalten werden, etwa Session-Variablen und Ähnliches. Das Aufräumen erledigt Oracle.
- Sicherheitsrelevante Informationen sind hier vor dem DBA sicher. So könnten z. B. entschlüsselte Informationen von Tabellen hier entschlüsselt zwischengelagert werden.
- Daten können für die Dauer einer Transaktion geparkt werden. Einen Fall kann ich zwar erwähnen, aber noch nicht erklären (ich komme später darauf zurück): In einer Transaktion können alte Daten vor der Änderung geparkt werden, um nach dem Einfügen oder Aktualisieren mit diesen Daten weitere Aktionen auszuführen.
- In Data Warehouses können temporäre Tabellen auch dazu dienen, Zwischenergebnisse aus vielen Teiltabellen zu speichern. Auf diese Weise kann der Optimizer unterstützt werden, der ansonsten aufgrund der vielen Tabellen manchmal nicht den optimalen Ausführungsplan findet.

4.1.4 Partitionierte Tabellen

Partitionierte Tabellen sind Tabellen, deren Zeilen intern auf mehrere physikalische Teiltabellen verteilt werden. Diese Teiltabellen können wiederum in jeweils unterschiedlichen Tablespaces gespeichert werden. Das hat zur Folge, dass eine Tabelle kontrolliert in mehreren Datendateien gespeichert werden kann. Für den Anwender ändert sich zunächst einmal nichts: Die Tabelle wird nach wie vor als eine logische Einheit über SQL angesprochen. Die Partitionierung erfolgt für den Anwender transparent.

Die Partitionierung kann nach verschiedenen Kriterien erfolgen. Hier stehen im Grunde drei Verfahren zur Auswahl:

- **Range**
  Diese Methode definiert Wertebereiche, die über die Zuordnung entscheiden. Der Klassiker sind Datumsbereiche (z. B. aktuelles Quartal, letztes Jahr etc.). Version 11g erweitert dieses Verfahren noch um den Typ *Interval*, der es erlaubt, automatisch neue Partitionen anzulegen, falls die eingefügten Werte nicht in die verfügbaren Partitionen eingefügt werden können. Ein Beispiel könnte sein, dass eine neue Partition für jedes Geschäftsjahr automatisch angelegt wird.
- **List**
  Bei diesem Verfahren wird ein Wert gegen eine Liste von Werten geprüft (z. B. Ländernamen) und entsprechend in eine Partition gelegt. Beispiele dafür sind etwa Länderlisten, die zu einer Verkaufsregion gehören (Deutschland, Österreich und Schweiz gehen in die Partition DACH etc.).

- **Hash**
  Das Hash-Partition-Verfahren setzt einen Hash-Algorithmus ein, der in diesem speziellen Fall nur sehr wenige Hash-Werte liefert. Optimal funktioniert er mit 2, 4, 8, 16 ... Partitionen, also mit der Zweierpotenzreihe. Ein Spaltenwert wird durch den Hash-Algorithmus geschleust und anhand des Hash-Wertes auf die Partition verteilt.

Es gibt aus meiner Sicht drei Gründe für das Partitionieren von Tabellen. Diese sind (in der Reihenfolge ihrer Bedeutung):

- Es erleichtert die Administration, weil es dem Administrator erlaubt, Teile einer Tabelle seltener in das Backup zu nehmen als andere Teile.

  Da die Tabelle auf mehrere Datendateien verteilt ist, kann also auch die Backup-Strategie für Altdaten anders ausfallen als für aktuelle Daten; ebenso können für Altdaten, die vielleicht nicht mehr oft gebraucht werden, preiswertere (langsamere) Speichermedien verwendet werden. Indirekt steigt durch die niedrigere Last beim Backup auch die Verfügbarkeit der Datenbank.
- Es erleichtert die Administration, weil es dem Administrator erlaubt, Teile der Daten schneller und unkomplizierter zu löschen oder zu bewegen.

  Wenn Daten aufgrund der gesetzlichen Vorgaben nicht mehr gespeichert werden müssen, tendieren die meisten Unternehmen dazu, diese Daten aus haftungsrechtlichen Gründen auch zu löschen. Oft müssen sie dies aufgrund der Gesetzeslage auch tun. Eine Partition mit diesen Daten zu löschen geht erheblich viel schneller als eine `delete`-Anweisung auf eine Tabelle mit mehreren Milliarden Zeilen. Die Archivierung von Altdaten, das Verschieben solcher Daten auf eine langsamere Festplatte etc. sind weitere Gründe für die erleichterte Administration.
- Es *kann* die Performance von SQL-Anweisungen erhöhen, wenn die Partitionierungsmethode *ganz gezielt* für diese *eine* Art von Anfrage optimiert wurde.

  Hier bewegen wir uns eigentlich ausnahmslos im Bereich von Data Warehouses mit drastisch vielen Daten. Durch eine sinnvolle Partitionierung (als Beispiel: monatsweise im aktuellen Jahr) können mehrere Prozessoren parallel an den Monatsberichten arbeiten, um die Daten anschließend in einen Jahresbericht zu überführen. Bei Anfragen, die ansonsten einen Full Table Scan auf die Tabelle ausführen, kann die Datenmenge eingeschränkt werden, wenn das `where`-Kriterium anzeigt, dass die gesuchten Daten *ausnahmslos* in einer Partition liegen (Oracle nennt dies *Partition Pruning*). Im Gegensatz dazu kann aber die Performance auch deutlich langsamer werden, wenn das `where`-Kriterium gerade *nicht* in einer Partition liegt, weil dann nicht nur ein, sondern entsprechend der Anzahl der Partitionen viele *Full Partition Scans* durchgeführt werden müssen. Andersherum: Wenn Sie eine Anfrage in einem transaktionsorientierten System beschleunigen möchten, denken Sie bitte als Allerletztes daran, dafür die Partitionierung zu verwenden.

Dies ist ein Unterschied zu vielen anderen Datenbanken, in denen solche Verfahren üblicher und auch nötiger sind. Oracle benötigt solche Verfahren im normalen Betrieb transaktionsorientierter Anwendungen nur im begründeten Ausnahmefall.

4.2 Index

Kommen wir doch noch einmal zu dem Problem der Platzverwaltung in einer Heap Organized Table zurück. Wie können wir hier Ordnung hineinbringen und dennoch die volle Flexibilität beliebiger Sortierkriterien erreichen? Eine wirklich gute Idee wäre, um in unserem Bild mit dem Lager zu bleiben, die Lagerplatznummer zu nutzen. Warum legen wir nicht eine Liste mit Einträgen für alle Farben an? Pro Farbe wird ein Blatt eingefügt, und auf dem Blatt steht die Lagerplatznummer der Eimer der entsprechenden Farbe. Die einzelnen Blätter werden in einem Ordner, sortiert nach Farbe, abgelegt. Dann können wir zudem noch einen Ordner nach Hersteller, einen nach Gebindegröße etc. erstellen. Auch wenn die Farbeimer wild durcheinanderstehen, können wir nun in den entsprechenden Ordnern sehr schnell nach bestimmten Farbeimern suchen, und über die Lagerplatznummer finden wir diese auch. Sie finden alles schneller, allerdings zulasten eines höheren Aufwands beim Einräumen, denn nun müssen Sie ja alle Änderungen am Lager penibel in den Listen vermerken. Diese Listen müssen auch in sich gepflegt werden, denn wenn z. B. eine Lieferung gelbe Farbe kommt, der Platz auf dem Blatt für gelbe Farbe aber nicht mehr ausreicht, müssen Sie ein neues Blatt hinter dem letzten Blatt für Gelb einfügen etc. Dazu werden Indizes verwendet, die wir uns nun ein wenig genauer ansehen werden.

Ein Index beschleunigt den Suchvorgang in Datensätzen, indem er ein Attribut der Tabelle sortiert speichert. Anstatt also die gesamte Tabelle seriell zu durchsuchen, sucht die Datenbank gezielt im Index, liest dort die `rowid` der indizierten Zeile und greift mit dieser Information auf die Tabelle zu. Um die Daten sortiert zu speichern, legt die Datenbank parallel zur Tabelle also ein neues Datenbankobjekt, eben den Index, an, der diese Informationen speichert. Wird die Tabelle verworfen, sorgt Oracle auch dafür, dass alle auf ihr beruhenden Indizes ebenfalls gelöscht werden. In der Diskussion der Heap Organized Table hatten wir gesehen, dass jede Zeile eine eindeutige `rowid` besitzt. Diese können Sie als Pseudospalte in einer `select`-Anweisung abfragen:

```
SQL> select rowid, ename, job
  2    from emp;
ROWID              ENAME      JOB
------------------ ---------- ---------
AAAQa3AAEAAAAAkAAA SMITH      CLERK
AAAQa3AAEAAAAAkAAB ALLEN      SALESMAN
```

```
AAAQa3AAEAAAAAkAAC WARD       SALESMAN
AAAQa3AAEAAAAAkAAD JONES      MANAGER
AAAQa3AAEAAAAAkAAE MARTIN     SALESMAN
AAAQa3AAEAAAAAkAAF BLAKE      MANAGER
AAAQa3AAEAAAAAkAAG CLARK      MANAGER
AAAQa3AAEAAAAAkAAH SCOTT      ANALYST
AAAQa3AAEAAAAAkAAI KING       PRESIDENT
...
14 Zeilen ausgewählt.
```

Listing 4.1 Darstellung der »rowid« über die Pseudospalte »rowid«

[!]

Merke

Die `rowid` ist vom Datentyp `rowid` und wird in Base64-Codierung dargestellt. Bei dieser Codierung werden die Zeichen `A–Z`, `a–z`, `0–9`, `+` und `/` genutzt. Die `rowid` ist 10 Byte lang. Inhaltlich setzt sie sich aus folgenden Einzelinformationen zusammen:

- **Datenbankobjektnummer**
  Jedes Segment (das ist der Oberbegriff über Strukturen wie z. B. eine Tabelle) hat eine eigene Nummer. Normalerweise steht hier also die interne Nummer der Tabelle, zu der die Zeile gehört.
- **Datendateinummer**
  die interne Nummer der Datendatei (relativ zum Tablespace, zu dem die Datei gehört), die den Datenbankblock enthält
- **Datenbankblocknummer**
  Dies ist der Block, der auch im Data Block Buffer der SGA gespeichert wird. Dieser Block enthält unsere Zeile.
- **Zeile innerhalb des Datenbankblocks**
  Dabei handelt es sich um einen Zeiger auf die Zeile innerhalb des Datenbankblocks.

Sie können sich diese Informationen mit einer SQL-Abfrage auch ausgeben lassen. Wir verwenden dazu ein von Oracle mitgeliefertes Package `dbms_rowid`, das uns Zugriff auf diese Informationen ermöglicht (die Abfrage muss als Administrator ausgeführt werden, etwa als Benutzer `SYSTEM`):

```
SQL> select e.rowid,
  2         f.file_name,
  3         dbms_rowid.rowid_block_number(e.rowid) block_number,
  4         dbms_rowid.rowid_row_number(e.rowid) pos_in_block
  5    from scott.emp e
  6    join dba_data_files f
  7      on dbms_rowid.rowid_to_absolute_fno(
```



```
  8           e.rowid, 'SCOTT', 'EMP') = f.file_id
ROWID              FILE_NAME          BLOCK_NUMBER POS_IN_BLOCK
------------------ ------------------ ------------ ------------
AAFEu6AAEAAAFXkAAA C:\...\USERS.DBF          21988            0
AAFEu6AAEAAAFXkAAB C:\...\USERS.DBF          21988            1
AAFEu6AAEAAAFXkAAC C:\...\USERS.DBF          21988            2
AAFEu6AAEAAAFXkAAD C:\...\USERS.DBF          21988            3
AAFEu6AAEAAAFXkAAE C:\...\USERS.DBF          21988            4
AAFEu6AAEAAAFXkAAF C:\.. \USERS.DBF          21988            5
AAFEu6AAEAAAFXkAAG C:\...\USERS.DBF          21988            6
AAFEu6AAEAAAFXkAAH C:\...\USERS.DBF          21988            7
AAFEu6AAEAAAFXkAAI C:\...\USERS.DBF          21988            8
AAFEu6AAEAAAFXkAAJ C:\...\USERS.DBF          21988            9
AAFEu6AAEAAAFXkAAK C:\...\USERS.DBF          21988           10
AAFEu6AAEAAAFXkAAL C:\...\USERS.DBF          21988           11
AAFEu6AAEAAAFXkAAM C:\...\USERS.DBF          21988           12
AAFEu6AAEAAAFXkAAN C:\...\USERS.DBF          21988           13
```

Listing 4.2 Darstellung der Bestandteile der »rowid«

Für die Datenbank bietet die `rowid` die schnellste Möglichkeit, eine Zeile zu finden, da sie so etwas Ähnliches wie einen Hardware-Pointer auf die physikalische Speicherstelle der Zeile darstellt. Bis auf eher exotische Ausnahmen (bei der Speicherung von Daten in sogenannten *Clustern*) ist eine `rowid` einer Zeile einer Tabelle datenbankweit eindeutig.

Ein Index speichert also zu jedem indizierten Fachbegriff die `rowid` und sorgt neben der verbesserten Ordnung noch für etwas anderes: Er koordiniert die lesenden und schreibenden Zugriffe und verhindert so, dass fehlerhafte Einträge in den Index geschrieben werden. Da die Datenbank eine Zeile einer Tabelle über einen Index in wenigen Suchschritten findet, egal, ob die Tabelle 100 oder 100 Millionen Zeilen enthält, ist die Suche über einen Index immer weitgehend konstant schnell. Etwas genauer: Die Geschwindigkeit der Suche hängt von der Tiefe des Indexbaums ab. Allerdings sind bei Oracle die Indexbäume meistens zwei, drei Ebenen tief, sodass die unterschiedliche Suchdauer weitgehend ignoriert werden kann.

Der normale Index ist der *B*Baum-Index*. Dieser Indextyp wird angelegt, wenn »einfach nur« eine `create index`-Anweisung abgesetzt wird. Doch Oracle unterscheidet zwischen verschiedenen Varianten, die allerdings technisch nicht sehr verschieden sind: dem B*Baum-, dem *Reverse-Key*- und dem *funktionsbasierten Index*. All diese Indizes existieren in der Variante *Unique* oder *Non Unique*. Darüber hinaus gibt es noch den etwas exotischeren *Bitmap*-Index, der in aller Regel nur für Data Warehouses Verwendung findet. Zwar ist dieser Index in diesem Zusammenhang wirklich cool, aber für uns ist er etwas außerhalb des Fokus, daher werde ich diesen Indextyp nicht genauer

besprechen. Zudem ist es möglich, eigene Indextypen zu programmieren, und Oracle hat dies für verschiedene Problemdomänen auch getan, z. B. für XML mit dem `XMLIndex`. Diese speziellen Indextypen werden konsequenterweise als *Domain Indexes* bezeichnet. Sie werden sie kennenlernen, wenn wir bei diesen speziellen Bereichen der Datenbank angelangt sind.

4.2.1 Anmerkung zur Benutzung von Indizes

Bevor wir uns diese Indextypen genauer ansehen, noch einige Überlegungen zum Einsatz dieser Indizes. Ich werde auf eine Bedeutung dieser Indizes im Zusammenhang mit der Prüfung der Datenbank-Constraints zu sprechen kommen, doch interessiert mich hier zunächst einmal der Einsatz von Indizes zur Erhöhung der Lese-Performance. Es scheint mir einer der großen Mythen über Datenbanken zu sein, dass Indizes eine Abfrage immer schnell machen. Das kann sein, muss aber überhaupt nicht so sein. Andersherum: Wäre es so, warum sollte Oracle nicht einfach jede Spalte zwangsweise indizieren? Dann hätte man doch per Definition eine schnelle Datenbank. Doch leider funktioniert es so nicht. Zunächst einmal *reduzieren* Indizes nämlich die Performance der Datenbank, zumindest beim Schreiben. Da der Index gepflegt und diese Pflege mit jedem `insert`, `update` oder `delete` durchgeführt werden muss, verlangsamt der Index den Schreibprozess in der Datenbank. Sollten Sie also in eine Tabelle öfter schreiben als lesen, ist ein Index zunächst nicht ratsam.

Dann muss ein Index die Suche auch wirklich beschleunigen *können*. Stellen wir uns den Index dazu wie den Index in einem Fachbuch vor. Nun denken wir uns, dass wir im Index das Wort »und« indiziert hätten. Sie suchen nun jedes Vorkommen des Wortes »und« im Buch. Sieh mal an, sagen Sie sich, auf Seite 1 steht das Wort. Also blättern Sie nach vorn und suchen das erste Vorkommen auf Seite 1. Dann zurück zum Index. Oha, Seite 2. Und so fort. Natürlich ist in einem solchen Fall das Lesen des gesamten Buches viel schneller. Bei Oracle kommt hinzu, dass die Datenbank immer einen ganzen Rutsch Zeilen der Tabelle auf einmal liest, einfach, weil sie annimmt, dass die nächsten Zeilen sicher auch noch gebraucht werden. Sollte der indizierte Eintrag also nicht selten genug vorkommen, wird Oracle die Benutzung dieses Index schlicht ablehnen. Er bedeutete mehr Aufwand, als er Nutzen brächte. Wir bezeichnen ein Suchkriterium in diesem Zusammenhang als unterschiedlich *selektiv*. Ein Kriterium, das nur für ein Tausendstel der Zeilenmenge einer Tabelle zutreffend ist, ist also deutlich selektiver als ein Kriterium, das für jede zweite Zeile gilt. Je höher die Selektivität eines Kriteriums ist, desto sinnvoller ist die Verwendung eines Index. Als Faustregel gilt, dass maximal etwa 5–10 % der Zeilen durch einen indizierten Begriff zurückgeliefert werden dürfen, ansonsten rechnet sich der Gebrauch nicht. Das ist aber natürlich eine Zahl, die von vielen Faktoren, wie der Länge der Zeile und damit der Anzahl der Datenblöcke, die gelesen werden müssen, abhängig ist.

Als nächstes Kriterium sollten die Indizes, die Sie auf eine Tabelle gelegt haben, auch *benutzt* werden. Das klingt seltsam, ist es aber nicht. Unter realistischen Datenmengen getestet, wird Oracle Ihnen Informationen darüber geben, ob ein Index aus Sicht der Datenbank Sinn macht oder nicht. Der Optimizer der Datenbank überschlägt die Kosten, die die Benutzung des Index für die Abfrage nach sich zieht, und entscheidet sich für die preiswerteste Alternative. Ist diese Alternative ein Full Table Scan, wird der Index ignoriert. Im Regelfall hat Oracle bei dieser Entscheidung auch recht. Nun kann es aber sein, dass ein hochselektiver Index dennoch nicht genutzt wird. Das kann z. B. dann der Fall sein, wenn Sie die Spalte `last_name` indiziert haben, in Ihrer Suche aber konsequent nach `upper(last_name)` suchen. In diesem Fall kann der Index nicht benutzt werden, weil Sie einfach nach etwas suchen, was nicht im Index steht. Verwenden Sie in diesem Fall einen Index über `upper(last_name)` (funktionsbasierter Index, siehe unten). Das ist nur ein Beispiel für viele Gründe, die der Benutzung eines Index im Weg stehen.

Eine letzte wichtige Regel: Indizieren Sie eine Spalte nur dann, wenn sie noch nicht indiziert ist. Ein Index kann mehrere Spalten indizieren, wobei er zunächst die erste, dann die zweite Spalte und so fort indiziert. Normalerweise werden Indizes auf mehrere Spalten also angelegt, wenn das erste Indizierungskriterium nicht ausreichend selektiv ist, in Kombination mit einem zweiten Kriterium aber schon. Die Reihenfolge der Indizierung richtet sich im Normalfall nach der Selektivität der indizierten Spalten: Die selektivste Spalte kommt als erste an die Reihe. Ist nun aber eine Spalte bereits durch einen anderen Index indiziert, macht eine erneute Indizierung keinen Sinn. Allerdings gibt es auch Ausnahmen von dieser Regel: Ist eine Spalte in einem anderen Index zwar enthalten, nicht aber als erste Spalte, kann eine erneute Indizierung durchaus sinnvoll sein. Der Grund: Wenn in einer Suchabfrage nur nach der zu indizierenden Spalte gefiltert wird, diese aber in einem Index erst als zweite Spalte auftaucht, ist die Benutzung dieses Index viel weniger effizient, als wäre die Spalte an der ersten Position indiziert. Daher wird der Optimizer die Verwendung dieses Index im Regelfall ablehnen.

Der Grund, warum eine Spalte nur einmal (als erster Eintrag in einem Index) indiziert werden sollte, ergibt sich aus dem vorher Gesagten: Indizes belasten die Schreib-Performance und verbrauchen nicht unerheblichen Plattenplatz. Zehn Indizes auf die gleiche Spalte belasten die Schreib-Performance zehnmal, optimieren die Abfrage aber nicht weiter. Zudem wird die Optimierung der `select`-Anweisung aufwendiger, weil die Optimierung immer mehr verfügbare Indizes ins Kalkül ziehen muss.

4.2.2 B*-Baum-Index

Dies ist die technische Bezeichnung aller Indizes, die wir im Folgenden besprechen werden. Daher gelten die allgemeinen Anmerkungen für alle Indizes. Die weiteren

Typen unterscheiden sich lediglich darin, welche Werte indiziert werden, nicht in der technischen Umsetzung.

B*Baum-Indizes funktionieren grob wie die alten Ratespiele, in denen mit möglichst wenigen Versuchen eine Zahl zwischen 1 und 1.000 geraten werden sollte. Man fängt in der Mitte an und teilt immer weiter, bis die Zahl geraten ist. Allerdings werden bei Indizes nicht alle Stellen eines Begriffs einzeln indiziert, sondern pro Entscheidungsschritt werden Bereiche unterschieden. Insofern verhält sich ein Index eher wie ein Amt: »Einwohner mit den Namen A–D bitte Zimmer 23« etc.

Auf diese Weise werden Indizes mit relativ wenigen Suchschritten fündig. Ein B*Baum-Index bei Oracle ist meist nur wenige Ebenen tief. Das bedeutet, dass der Index bei vielen indizierten Begriffen eine erhebliche Breite einnimmt. Damit diese Indexstruktur effizient verwaltet werden kann, werden die zusammengehörenden Daten möglichst in einen Block auf der Festplatte gespeichert. Im Gegensatz zur Heap Organized Table muss also ein erheblicher Aufwand betrieben werden, um die Daten an der »richtigen« Stelle zu speichern. Der B*Baum zeichnet sich dadurch aus, dass sich die Konten der Baumstruktur (in Anbetracht der Breite des Index wäre hier wohl eher von einer Strauchstruktur zu sprechen ...) selbst balancieren. Damit ist gemeint, dass Einträge in den Knoten so auf die Nachbarknoten verteilt werden, dass alle Knoten in etwa gleich viele Einträge beinhalten. Durch diesen Kniff werden die Verwaltung und die Suchgeschwindigkeit optimiert. Zudem wird jeder einsortierte Begriff mit seinem Vorgänger und seinem Nachfolger verknüpft, sodass eine doppelt verknüpfte Liste entsteht. Diese Verknüpfung macht einen Index hocheffizient, wenn es darum geht, Bereichsüberprüfungen durchzuführen. Eine solche Bereichsüberprüfung (Oracle nennt dies einen *Index Range Scan*) wird z. B. bei einer so einfachen Abfrage wie dieser hier durchgeführt, nachdem die Spalte `LAST_NAME` indiziert wurde:

```
SQL> set autotrace on;
SQL> select last_name, first_name, hire_date
  2    from employees
  3   where last_name like 'K%'
LAST_NAME                 FIRST_NAME           HIRE_DATE
------------------------- -------------------- ----------
Kaufling                  Payam                01.05.1995
Khoo                      Alexander            18.05.1995
King                      Janette              30.01.1996
King                      Steven               17.06.1987
Kochhar                   Neena                21.09.1989
Kumar                     Sundita              21.04.2000
6 Zeilen ausgewählt.
Ausführungsplan
```



```
----------------------------------------------------------
Plan hash value: 2077747057
----------------------------------------------------------
| Id  | Operation
----------------------------------------------------------
|   0 | SELECT STATEMENT
|   1 |  TABLE ACCESS BY INDEX ROWID
|*  2 |   INDEX RANGE SCAN
----------------------------------------------------------
Predicate Information (identified by operation id):
----------------------------------------------------------
   2 - access("LAST_NAME" LIKE 'K%')
       filter("LAST_NAME" LIKE 'K%')
```

Listing 4.3 Benutzung eines Index

Für diese (gekürzte) Ausgabe haben wir den *Ausführungsplan*, d. h. die interne Strategie zur Ausführung dieser Anweisung, sichtbar gemacht, indem wir die Anweisung `set autotrace on` vorweg gesendet haben. Zurück zum Index: Warum hat diese Abfrage einen Index Range Scan zur Folge? Der Index wird den ersten Eintrag lokalisieren, für den der Nachname mit `M` beginnt. Anschließend kann der Index über die doppelt verknüpfte Liste einfach so lange seine Nachfolger lesen, bis deren Nachname mit dem nächstgrößeren Buchstaben beginnt. Diesen Bereich von Namen scannt der Index durch, daher der Name. Ähnliche Suchmuster können bei Zahlen und Datumsangaben durchgeführt werden.

Wie schon bei der Einführung zu Indizes besprochen, speichern diese Strukturen neben dem zu indizierenden Begriff auch die `rowid` der zu diesem Begriff gehörenden Zeile in einer Tabelle. Wenn der Index den gleichen Eintrag mehrfach gestattet, werden mehrere `rowids` gespeichert. Das ist die Standardeinstellung. Soll jeder Begriff lediglich genau einmal indiziert werden dürfen, wird dies durch das Schlüsselwort `unique` bei der Erstellung des Index vermerkt:

```
create unique index idx_emp_last_name_u
    on employees(last_name);
```

Listing 4.4 Erstellung eines Unique Index

Technisch ist ein `Unique Index` bis auf diese Unterscheidung identisch mit einem `Non Unique Index`.

4.2.3 Reverse-Key-Index

Gerade bei aufeinanderfolgenden Nummern besteht die Gefahr, dass ein Index sich sozusagen einseitig belastet: weil aufeinanderfolgende Zahlen sich lediglich in den

letzten Stellen unterscheiden, tendiert der Index dazu, viele Einträge in *einen* Teil des Indexbaums einzufügen und andere Teile schwach zu belasten. Da der Index sich selbst balanciert, hat dies eine häufige Umstrukturierung des Index zur Folge. Zudem ist eine weitere Folge, dass sich, im Mehrbenutzerbetrieb und stärker noch in geclusterten Datenbanken, ein Run mehrerer Sessions auf wenige Indexblöcke einstellen wird, weil alle in die gleichen Blöcke schreiben möchten. Eine Optimierung besteht darin, den Index die zu identifizierende Zahl von hinten nach vorn lesen zu lassen. Aufeinanderfolgende Ziffern unterscheiden sich nun in der ersten Stelle, was dazu führt, dass der Index aufeinanderfolgende Zahlen über den gesamten Index verteilt. Die Anweisung für einen solchen Index lautet:

```
create (unique) index idx_emp_id
       on employee(employee_id) reverse key;
```

Der Nachteil dieser Methode besteht darin, dass nun keine Index Range Scans auf diese Einträge mehr möglich sind, was aber wohl bei technischen Primärschlüsseln zu verschmerzen sein dürfte.

4.2.4 Funktionsbasierter Index

Der funktionsbasierte Index stellt insofern eine Besonderheit dar, als nicht ein Spaltenwert indiziert wird, sondern das Ergebnis einer Berechnung. Diese Berechnung kann im Grunde beliebig komplex sein, allerdings müssen die Berechnungen deterministisch sein, was bedeutet, dass die Funktion zu jeder Zeit für die gleichen Eingangsgrößen gleiche Ausgangswerte zurückliefert. Daher ist eine Logik, die sich z. B. auf eine Zufallszahl, das Systemdatum oder angemeldete Datenbankbenutzer bezieht, nicht erlaubt. Achten Sie auch darauf, nicht mit kulturabhängigen Daten zu rechnen, wie es z. B. der `n`-te Tag der Woche ist, der etwa in Amerika, wo die Woche am Sonntag beginnt (der damit die Ordnungszahl 1 erhält), anders definiert ist als hierzulande.

Sehen wir uns ein einfaches Beispiel an: Eine Tabelle speichert Bestellungen. Alle Bestellungen haben eine Bestellmenge und eine Liefermenge. Nun sollen die Bestellungen gefiltert werden, deren Bestellmenge ungleich der Liefermenge ist, was eine nicht abgeschlossene Bestellung anzeigt (ich weiß, das Beispiel ist relativ stark vereinfacht, zeigt aber das Prinzip). Wenn die Tabelle über mehrere Millionen Einträge verfügt, müssen ebenso viele Berechnungen angestellt werden, nur um einen sehr kleinen Prozentanteil der Zeilen zu filtern. Um diese Abfrage zu beschleunigen, wird ein Index über das Ergebnis der Differenz erstellt:

```
create index idx_order_open
       on orders(ordered_items - delivered_items)
```


Nun muss die Abfrage nach den offenen Bestellungen den gleichen Funktionsaufruf beinhalten wie die Definition des Index:

```
select *
  from orders
 where ordered_items - delivered_items <> 0;
```

Anstatt nun Millionen Rechenoperationen auszuführen, wird lediglich ein Index Scan durchgeführt, der uns die `rowid` der Zeilen liefert, die einen Lieferrückstand (oder zu viele gelieferte Produkte) haben. Wann und wie wird ein solcher Index gepflegt? Die Antwort ist: Wie jeder andere Index auch, nämlich durch eine DML-Anweisung, also während der Datenmanipulation mittels `insert`, `update` oder `delete`. Sobald die Datenmanipulation abgeschlossen wird, werden die beteiligten Indizes aktualisiert.

Eines stört noch an dem gerade erzeugten Index: Er indiziert sehr viele `0`-Werte. Doch eigentlich wollen wir diese Werte nicht indizieren. (Sie erinnern sich daran, dass Indizes nur genutzt werden, wenn die gesuchten Werte stark selektiv sind? Der `0`-Wert in unserem Beispiel ist es sicher nicht.) Sie verbrauchen also nur unnötig Speicherplatz. Doch wie können diese Werte aus dem Index entfernt werden? Die Lösung macht sich die Tatsache zunutze, dass Indizes grundsätzlich unfähig sind, `null`-Werte zu indizieren. Da diese Werte undefiniert sind, können sie auch nicht in eine (sortierte) Indexstruktur eingepasst werden, ein Index ignoriert den Wert `null`. Lassen Sie uns also die Funktion so umschreiben, dass der Normalwert `= null` gesetzt wird:

```
create index idx_order_open
       on orders(case when ordered_items = delivered_items
                      then null
                      else ordered_items - delivered_items end)
```

Achten Sie nun aber darauf, auch Ihre Abfrage mit dieser `case`-Anweisung zu schreiben, weil Oracle ansonsten nicht erkennen kann, dass der Index benutzt werden könnte:

```
select *
  from orders
 where case when ordered_items = delivered_items
            then null
            else ordered_items - delivered_items end
       is not null
```

Listing 4.5 Beispiel zum Einsatz eines funktionsbasierten Index

Funktionsbasierte Indizes können ebenfalls Unique sein wie alle B*Baum-Indizes. Sie können außerdem auch PL/SQL-Funktionen aufrufen (auch Ihre eigenen!) und von

daher grundsätzlich beliebig komplexe Berechnungen zur Datenmanipulationszeit durchführen und die Ergebnisse indiziert speichern. Diese Indizes können die Abfrage-Performance drastisch erhöhen, haben aber natürlich auch einen Kostenanteil während des Schreibens. An diesem Beispiel sieht man zudem sehr gut, dass die Möglichkeiten für das Performance-Tuning für einen Administrator begrenzt sind: Er müsste nicht nur erkennen, dass diese Optimierung an einer bestimmten Stelle im Code Sinn ergäbe, sondern auch noch den SQL-Code so ändern, dass der Index auch tatsächlich benutzt werden kann. Daher ist es die Aufgabe des Entwicklers, sich mit dieser Materie so weit zu beschäftigen, dass er die Performance-Steigerung hier erkennt, *bevor* die Anwendung in Produktion geht, und nicht erst mit dem ersten Bugfix ...

4.3 Views und Materialized Views

Views und Materialized Views sind extrem wichtige Hilfsmittel der Datenbank, die immer wieder während der Programmierung gebraucht werden. Sie werden zur Steuerung der Datensicherheit, der Kapselung von komplexen Abfragen, zur Beschleunigung von Anwendungen und vielen weiteren Zielen verwendet.

4.3.1 Views

Betrachten wir zunächst einfache *Views*. Views sind einfach nur gespeicherte `select`-Anweisungen im Data Dictionary, die einen Namen erhalten haben. Wird eine View abgefragt, wird stattdessen die der View zugrunde liegende `select`-Anweisung ausgeführt. Dieses Vorgehen hat eine Reihe von Vorteilen:

- Es kapselt Komplexität, weil der Anwender die Definition der View nicht kennen muss, sondern lediglich das Ergebnis einer `select`-Anweisung konsumiert, die ein anderer Entwickler erstellt hat.
- Es kapselt das Datenmodell vor der Anwendung und macht daher die Änderung des Datenmodells leichter.
- Es sichert den Zugriff auf Daten, weil eine View z. B. nur eine Auswahl der Spalten und Zeilen einer Tabelle umfasst und dem Anwender den Zugriff auf die Tabellen, die in der View angesprochen werden, verwehren kann.

Views erledigen außerdem noch eine ganze Reihe weitere schöne Dinge für uns. Sie werden ganz einfach erzeugt:

```
SQL> create or replace view emp_vw
  2  as
  3  select ename, job, dname, loc, grade
  4    from emp e
  5    join dept d on e.deptno = d.deptno
  6    join salgrade s on e.sal between s.losal and s.hisal
  7  ;
View created.

SQL> select *
  2    from emp_vw;
ENAME      JOB       DNAME          LOC            GRADE
---------- --------- -------------- ------------- ----------
KING       PRESIDENT ACCOUNTING     NEW YORK              5
FORD       ANALYST   RESEARCH       DALLAS                4
SCOTT      ANALYST   RESEARCH       DALLAS                4
JONES      MANAGER   RESEARCH       DALLAS                4
BLAKE      MANAGER   SALES          CHICAGO               4
CLARK      MANAGER   ACCOUNTING     NEW YORK              4
ALLEN      SALESMAN  SALES          CHICAGO               3
TURNER     SALESMAN  SALES          CHICAGO               3
MILLER     CLERK     ACCOUNTING     NEW YORK              2
WARD       SALESMAN  SALES          CHICAGO               2
MARTIN     SALESMAN  SALES          CHICAGO               2
ADAMS      CLERK     RESEARCH       DALLAS                1
JAMES      CLERK     SALES          CHICAGO               1
SMITH      CLERK     RESEARCH       DALLAS                1
14 rows selected.
```

Listing 4.6 Erzeugung und Verwendung einer View

Es reicht eine `create or replace`-Anweisung, um die Definition der View unter dem Namen, der folgt, im Data Dictionary zu hinterlegen. Bis auf den positiven Effekt, dass die `select`-Anweisung der View der Datenbank bekannt und daher bereits geparst ist, gibt es keinen Unterschied zur direkten Abfrage der `select`-Anweisung, die der View zugrunde liegt. Insbesondere benötigt eine View lediglich den Speicherplatz der `select`-Anweisung, nicht aber Platz für die Daten, die die Abfrage repräsentiert, weil diese nicht berechnet werden, bevor die View in einer `select`-Anweisung verwendet wird.

4.3.2 Materialized Views

Der Begriff *Materialized View* (MV) klingt zunächst etwas esoterisch, doch stellen diese Views in vielen Bereichen sehr leistungsfähige Konzepte dar, die die Programmierung einer Lösung stark vereinfachen können. In diesem Fall kann man auch oft von echten »Performance-Boostern« sprechen, denn im richtigen Umfeld eingesetzt,

können sie wie intelligente Indizes wirken. Doch zunächst: So eine Materialized View, was ist das eigentlich?

Eine Materialized View ist eine Sicht, deren Abfrageergebnis zu *einem definierten Zeitpunkt* ermittelt und dann auf die Festplatte gespeichert worden ist. Der Vorteil: Sollen die Daten dieser Sicht abgefragt werden, muss die aufwendige `select`-Anweisung nicht mehr ausgeführt werden, sondern es kann das gespeicherte Ergebnis zurückgeliefert werden – allerdings mit dem Nachteil, dass diese Daten nicht unbedingt aktuell sind. Sollten nach dem Aktualisieren der MV die Daten geändert worden sein, bekommt dies die MV nicht notwendigerweise mit. Doch oft ist die letzte Millisekunde gar nicht entscheidend: Sollen z. B. im Bereich des Berichtswesens die Daten des gestrigen Tages dargestellt werden, könnte man sich gut vorstellen, dass die aufwendige Abfrage der Daten nachts erledigt wird. Wenn sich die Daten von gestern heute nicht mehr ändern, ist die Abfrage der letzten Nacht für uns aktuell genug. Ebenso finden sich Szenarien im Umfeld von Daten, die nicht sehr häufig geändert werden, wie z. B. Stammdaten. Hier könnten MVs eine denormalisierte Sicht auf normalisierte Stammdatentabellen anbieten, die es der Anwendung erspart, jedes Mal das gesamte Gestrüpp normalisierter Stammdatentabellen abzufragen, um z. B. eine Adresse zu erhalten.

Eine zentrale Frage bei MVs bezieht sich darauf, wie und wann die Daten aktualisiert werden. Oracle bietet für MVs grundsätzlich folgende Möglichkeiten an:

- **Wie?**
  Die MV kann inkrementell oder komplett aktualisiert werden. Ob eine inkrementelle Aktualisierung möglich ist, hängt von der Komplexität der Abfrage ab. Je nach Datenbankversion ist die Fähigkeit dazu gestiegen, doch gibt es Abfragen, die nur komplett aktualisiert werden können. Inkrementelle Aktualisierungen sind nur möglich, wenn Oracle die Änderungen an den Basistabellen der MV protokollieren kann. Dazu werden *Materialized View Logs* eingesetzt.
- **Wann?**
  Zunächst einmal kann die MV auf Anweisung (*on demand*) aktualisiert werden. Zusätzlich können Sie aber auch ein Startdatum und ein Intervall benennen, zu dem die Aktualisierung, ähnlich einem Cron- oder AT-Job, mithilfe eines Datenbankjobs durchgeführt wird. Als letzte Option bietet es sich an, die Aktualisierung anzustoßen, wenn eine Datenänderung auf die an der MV beteiligten Tabellen durch `commit` bestätigt wird.

Zwar sind die Optionen vielfältig, doch werde ich Ihnen in einem Beispiel den prinzipiellen Vorgang beim Anlegen einer MV zeigen. Wir nehmen für unser Beispiel an, dass eine View auf eine Tabelle nur die Daten des gestrigen Tages darstellen soll. Die MV soll sich jedes Mal gegen Mitternacht selbstständig aktualisieren (*gegen* Mitternacht: Die Aktualisierung wird über einen Job in der Datenbank ausgeführt, der mit

einem Verzug von eventuell wenigen Sekunden gestartet werden kann, je nach Last auf der Datenbank). Eine solche MV würde wie folgt definiert:

```
SQL> create materialized view orders_yesterday
  2  refresh complete on demand -- Zeitgesteuertes Refresh
  3  start with sysdate -- MV wird sofort erstellt
  4  next trunc(sysdate) + interval '1' day -- und jeden Tag neu
  5  as
  6  select *
  7    from orders
  8   where trunc(order_date) = trunc(sysdate) - 1;
Materialized View wurde erstellt.
Abgelaufen: 00:00:01.85
```

Listing 4.7 Erstellung einer materialisierten Sicht

Zur Erläuterung: Die Anweisung `refresh complete on demand` besagt, dass die MV komplett aktualisiert werden soll (eine inkrementelle Aktualisierung wäre in diesem Beispiel Blödsinn), und zwar auf Anweisung. Als Startdatum wird das aktuelle Systemdatum vereinbart (das ist Standard und hätte nicht angegeben werden müssen), als Aktualisierungsintervall wird der jeweils nächste Tag um Mitternacht berechnet. Die Funktion `trunc()` wirkt bei Datumsangaben so, dass die Uhrzeit abgerundet und damit das Datum auf `00:00` Uhr eingestellt wird. Wird zu diesem Datum `interval '1' day` (ein Zeitraum der Länge `1 Tag`) hinzugerechnet, wird die MV am nächsten Tag, `00:00` Uhr, aktualisiert.

Da eine MV ein Zwischending zwischen einer Tabelle, einem Index und einer View ist, kann sie nicht, wie z. B. eine View, über die Anweisung `create or replace` ersetzt werden, sondern muss, wie eine Tabelle, zunächst gelöscht werden, wenn sie geändert werden soll.

4.4 PL/SQL-Programm

Eine weitere wesentliche Gruppe von Datenbankobjekten stellen die Programme dar, die in der Sprache PL/SQL oder auch Java (nicht in der Oracle XE), wenn Sie mögen, sogar in C oder C# erzeugt werden. Ich werde diese Objekte jetzt noch nicht im Detail besprechen, wir haben dafür schließlich noch ein ganzes Buch Zeit, sondern Ihnen lediglich einen ersten allgemeinen Überblick geben.

PL/SQL-Programme treten in verschiedenen Formen auf: als Packages, Prozeduren, Funktionen oder Trigger. Diese verschiedenen Formen dienen verschiedenen Zwecken, die wir später noch einzeln diskutieren werden. Allen diesen Formen ist gemeinsam, dass der Programm-Code im Data Dictionary gespeichert wird, ähnlich wie die

Definition einer Tabelle. Es werden also keine Code-Dateien außerhalb der Datenbank geführt (auch wenn dies für den Ex- und Import möglich ist), sondern der Code liegt immer direkt in der Datenbank. Das Kompilieren eines PL/SQL-Programms ist somit immer auch gleichbedeutend mit der Speicherung des Kompilats in der Datenbank. Dies ermöglicht dem Compiler von PL/SQL, den Programmcode gegen das sonstige Data Dictionary abzugleichen. Wenn also z. B. in einem PL/SQL-Programm auf eine Tabelle Bezug genommen wird, kann der Compiler testen, ob diese Tabelle auch existiert. Einmal kompilierte Objekte unterliegen zudem der Abhängigkeitskontrolle: Wird ein Objekt geändert, von dem dieser Code abhängig ist, und hat diese Änderung einen Fehler im Code zur Folge, wird der Code unmittelbar nach der Änderung des zugrunde liegenden Objekts invalide. Zudem übernimmt die Datenbank damit die Kontrolle über den Zugriff auf den Code: PL/SQL-Programme dürfen nur vom Besitzer des Codes oder von autorisierten anderen Datenbankbenutzern ausgeführt werden, ähnlich wie auch der Zugriff auf die Tabellen eines Benutzers von der Datenbank kontrolliert wird.

4.5 Sonstige Datenbankobjekte

Die sonstigen Datenbankobjekte, die ein Schema ausmachen, können zum derzeitigen Zeitpunkt eher summarisch besprochen werden. Falls nötig, komme ich auf einzelne Objekte noch genauer zu sprechen. Uns soll es im Moment reichen, grob zu wissen, was diese Objekte sind und wozu sie verwendet werden können.

4.5.1 Sequenzen

Oracle bietet einen auf den ersten Blick umständlich erscheinenden Mechanismus zur Erzeugung eindeutiger Schlüsselwerte an, vergleichbar dem Autowert anderer Datenbanken: die *Sequenz*. Eine Sequenz ist ein Datenbankobjekt, das nichts anderes tut, als neue Zahlen zurückzuliefern, ähnlich wie das auch ein Autowert-Datentyp täte. Doch im Gegensatz zu einem Datentyp in der Tabelle hat die Sequenz eine Reihe von Vorteilen:

- Sie kann parametriert werden. Zum Beispiel können der Startwert, der Maximalwert, die Schrittweite und viele andere Parameter eingestellt werden. Dies erhöht die Flexibilität sehr.
- Sie kann für mehr als eine Tabelle eindeutige Werte zur Verfügung stellen. Der Datentyp *Autowert* ist nur für die jeweilige Tabellenspalte eindeutig. Mithilfe einer externen Struktur können aber mehrere Spalten einer Tabelle oder auch mehrere Tabellen untereinander eindeutige Zahlen erhalten.
- Die Sequenz ist optimiert und für den massiv parallelen Zugriff vorbereitet. Damit ist diese Struktur die schnellste Möglichkeit, eine neue Zahl für eine Tabelle zu erzeugen.

Ein Nachteil der Sequenz sei allerdings auch nicht verschwiegen: Es ist mit einer Sequenz (ebenso wenig wie mit einer Autowert-Spalte anderer Datenbanken) nicht möglich, eine geschlossene Folge von Zahlen zu erzeugen. Wird z. B. eine Zahl aus der Sequenz abgerufen und dann doch nicht festgeschrieben, ist diese Zahl für die Sequenz verbraucht und wird nicht wieder geliefert.

Ein Gefühl muss ich allerdings entschärfen, das oft im Zusammenhang mit diesen »verworfenen« Zahlen aufkommt: das Gefühl der Verschwendung. Bei vielen Entwicklern stellt sich das Gefühl ein, man könne sich diesen laschen Umgang mit Zahlen nicht leisten, weil ansonsten schnell das Ende des `number`-Datentyps erreicht sei. Lassen Sie sich beruhigen: Der Datentyp `number` hat eine Maximalgröße von 1×10^{-130} bis $9.99...9 \times 10^{125}$ bei 38 Nachkommastellen. Sollten Sie also, sagen wir, 1.000.000 Zahlen pro Sekunde erzeugen, reichte der `number`-Datentyp allein der positiven Zahlen etwa $3{,}17^{111}$ Jahre ...

Eine Sequenz wird auf folgende Weise erzeugt:

```
create sequence my_seq;
```

Anschließend kann die Sequenz in einer `insert`-Anweisung wie folgt benutzt werden:

```
insert into orders (order_id, order_date, ...)
values (my_seq.nextval, sysdate, ...);
```

Listing 4.8 Erzeugung und Verwendung von Sequenzen

Alternativ kann ein sogenannter *Trigger* eingerichtet werden, ein Stück Programmcode, der auf eine Tabelle eingerichtet und automatisch ausgeführt wird, wenn eine Schreiboperation auf diese Tabelle ausgeführt wird. In diesem Trigger ist es möglich, den Wert der Sequenz vor dem Einfügen eines neuen Datensatzes ermitteln zu lassen. Ab Version 12c können Sequenzen auch als `default`-Wert einer Tabellenspalte referenziert werden und reduzieren so den Bedarf an Triggern für diesen Zweck (was schneller ist, weil kein Umgebungswechsel zwischen SQL und PL/SQL stattfindet, um einen neuen Sequenzwert zu ermitteln). Zudem steht nun auch eine »Autowert«-Spalte zur Verfügung, die Oracle eine *Identity*-Spalte nennt, die im Kern eine Sequenz erzeugt (und auch über die gleichen Optionen verfügt), diese aber nur für eine Tabelle verfügbar macht.

Als weitere Neuerungen gibt es für temporäre Tabellen Sequenzen, deren Schlüsselwerte nur pro Session eindeutig sind und die daher leichtgewichtiger in der Verwaltung sind.

4.5.2 Synonym

Ein Synonym ist ein alternativer Name für ein anderes Datenbankobjekt. Mit Synonymen können z. B. Tabellen einen anwenderfreundlicheren Namen erhalten. Ein

Synonym kann entweder innerhalb eines Schemas oder für die gesamte Datenbank (dann sprechen wir von einem `public synonym`) gültig und sichtbar sein. Nehmen wir an, dem Benutzer `OE` wäre ein `select`-Recht auf die Tabelle `promotions` des Datenbankbenutzers `SH` eingeräumt worden. Nun könnte `OE` die Tabelle abfragen als:

```
select *
  from sh.promotions;
```

`OE` könnte nun ein Alias für diese Tabelle mit dem Namen `promotion` erstellen:

```
create synonym promotion on sh.promotions;
```

Dann könnte er auf diese Tabelle nun mit folgender Anweisung zugreifen:

```
select *
   from promotion;
```

Listing 4.9 Erstellung und Verwendung von Synonymen

Die Benutzerrechte werden allerdings nach wie vor auf den durch das Synonym repräsentierten Datenbankobjekten vergeben, nicht auf den Synonymen. Dieses Synonym nennen wir ein privates Synonym, weil es einem Benutzer gehört und demzufolge nur für diesen Benutzer verwendbar ist. Alternativ kann ein öffentliches Synonym erzeugt werden, das einen alternativen Namen für ein Datenbankobjekt datenbankweit verfügbar macht. Dieses Verfahren nutzt Oracle bei den System- und Data-Dictionary-Views, die aus jedem Benutzer mit gleichem Namen angesprochen werden können. Natürlich müssen in diesem Fall die Bezeichner datenbankweit eindeutig sein.

4.5.3 Database Link

Ein Database Link ist ein Datenbankobjekt, das die Verbindung einer Datenbank zu einer anderen Datenbank repräsentiert. Dabei sind nicht nur Oracle-Datenbanken gemeint, sondern durchaus auch RDBMS anderer Hersteller. Oracle verwendet zur Verbindung zu Datenbanken anderer Hersteller das Produkt *Oracle Heterogenous Services*, das separat lizenziert werden muss. Treiber für ODBC- oder OLE-DB-Verbindungen werden allerdings mitgeliefert. Für die Verbindung zu entfernten Datenbanken ist es lediglich erforderlich, dass die entfernte Datenbank vom Datenbankserver aus »gesehen« werden kann. Das kann z. B. durch einen Eintrag in den Netzwerkeinstellungen der Datenbank (*tnsnames.ora*) oder der anderen Systeme zur Verbindungsaufnahme erfolgen. Der Database Link kann sich im Namen des gerade angemeldeten Benutzers oder auch eines festen Datenbankbenutzers anmelden (die selbstverständlich jeweils auf der entfernten Datenbank bekannt sein müssen). Hier

sehen Sie ein Beispiel für einen einfachen Database Link auf die Datenbank, die durch einen `TNSNames`-Eintrag `production` zu erreichen ist:

```
create database link prod
connect to hr identified by hr_pass
using 'production';
```

Nach der Erstellung des Database Links kann nun eine Tabelle des Benutzers `HR` auf der Datenbank `production` vom lokalen System abgefragt werden, indem folgende SQL-Anweisung abgesetzt wird:

```
select *
  from employees@prod;
```

Listing 4.10 Erstellung und Benutzung eines Database Links

Dabei ist es nicht erforderlich, dass der Benutzer `HR` auf der entfernten Datenbank das `select`-Recht auf die Tabelle freigegeben hat. Wer der Eigentümer des Database Links ist, ist für die entfernte Datenbank völlig unerheblich, sie führt alle Anweisungen über den Database Link mit den Rechten des Benutzers `HR` aus, weil sich der externe Benutzer in dessen Namen und mit dessen Passwort angemeldet hat. Für die entfernte Datenbank ist es im Grunde unerheblich, ob sich eine andere Datenbank oder ein »natürlicher« Benutzer angemeldet hat, über eine Netzwerkverbindung ist eine Benutzeranfrage gekommen, der Benutzer hat sich mit einem Namen und einem Passwort authentifiziert. Das ist alles, was für die Datenbank zählt. Auf diese Weise lassen sich selbstverständlich nicht nur `select`-Anweisungen an entfernte Datenbanken senden, sondern auch DML-Anweisungen wie `insert`, `update` oder `delete`, allerdings keine DDL-Anweisungen wie `create table` oder ähnlich. Oracle kümmert sich, transparent für den Anwender, um die Nickeligkeiten sogenannter *verteilter Transaktionen*, indem es die Abstimmung zwischen den Datenbanken, wann ein `commit` tatsächlich durchgeführt wird, automatisch steuert.

4.5.4 Große Datenmengen: CLOB, NCLOB, BLOB und BFile

Zur Speicherung großer Datenmengen in der Datenbank setzt Oracle LOB-Datentypen (LOB = *Large Object*) ein. Diese Typen unterscheiden die intern in der Datenbank gespeicherten Datentypen (`CLOB`, `NCLOB`, `BLOB`) und den extern gespeicherten `BFile`-Datentyp. Alle internen LOB-Typen sind Verweistypen, die ab einer Grenzgröße (etwa 4.000 Byte) die eigentlichen LOB-Daten in ein spezielles LOB-Segment ausgliedern. In der Tabelle wird in diesem Fall lediglich ein Zeiger auf den Speicherbereich im LOB-Segment gespeichert, der bei Bedarf aufgelöst wird. Die Datentypen haben folgende Ausprägungen:

- `CLOB`
  Character Large Object. Speichert Textinformationen in der Zeichensatzcodierung der Datenbank.
- `NCLOB`
  National Character Large Object. Speichert Textinformationen in der NLS-Zeichensatzcodierung der Datenbank.
- `BLOB`
  Binary Large Object. Speichert einen Binärstrom.
- `BFILE`
  Speichert einen Pointer auf eine extern zur Datenbank gespeicherte Datei.

Die Maximalgröße eines internen LOB-Datentyps beträgt beeindruckende 2^{32} * `Blockgröße in Byte` (bei einer Blockgröße von 8 KB also 32 TB pro Zelle). `BFiles` können so groß sein, wie das Betriebssystem es erlaubt. LOBs haben die seit Jahrzehnten nicht mehr empfohlenen Datentypen `LONG` und `LONG RAW` ersetzt, die Sie in eigenen Projekten bitte nicht mehr benutzen. (`LONG` ist, nebenbei, etwas grundsätzlich anderes als der `LONG`-Typ in Java!)

Mittlerweile verhält sich ein LOB-Datentyp für SQL oder PL/SQL annähernd wie ein `varchar2`-Datentyp. Alle Operationen, die auf `varchar2` angewandt werden können, können auch auf LOB-Datentypen angewandt werden, also z. B. `replace`, `substr` etc. Insofern merkt der Benutzer von diesen Datentypen nichts. Dennoch sollten nun nicht alle Zeichenketten als CLOB in der Datenbank hinterlegt werden. Weil Oracle um die potenzielle Größe dieser Datenstrukturen weiß, sind Lesezugriffe und vor allem die Übermittlung der Daten über das Netzwerk oftmals nicht so effizient wie eine normale Zeichenkette. Da Zeichenketten seit Version 12c ebenfalls eine Maximalgröße von 32 KByte in der Datenbank haben können, ist ein weiteres Argument für CLOBs möglicherweise hinfällig. Wenn Sie aber große Textmengen speichern müssen, machen Sie das in jedem Fall und ohne Vorbehalte in diesen Datentypen. Interessant ist die Diskussion, ob denn nun große Binärdaten (Bilder, Videos, Excel-Dateien etc.) oder große Textmengen innerhalb oder außerhalb der Datenbank gespeichert werden sollten. Oracle empfiehlt in jedem Fall, solche Daten innerhalb der Datenbank zu speichern. Die Gründe liegen in einem einheitlichen Benutzerzugriffsmanagement, einer einheitlichen Backup-Strategie und einer sichereren Speicherung, weil die Daten nicht durch einfaches Ändern eines Ordnernamens für die Datenbank unsichtbar werden können, wie das bei der externen Speicherung solcher Daten im Dateisystem der Fall wäre.

In der Realität müssen solche Argumente durch Tests untermauert werden. Werden die Daten in der Datenbank gespeichert, ist man auch auf die Oracle-API angewiesen, um die Daten zu lesen. Gerade zu Beginn der Einführung dieser Datentypen haben

die Entwickler einiges Lehrgeld zahlen müssen, was die Performance oder sonstige Einschränkungen dieser Datentypen anging. Das ist mittlerweile aber Vergangenheit. Der Datentyp `BFile` wird daher eher dafür empfohlen, externe Daten in die Datenbank einzulesen. Als Datentyp für Tabellen ist er selten anzutreffen, aber immer noch besser als eine einfache Zeichenkette, die eine URL repräsentiert: Durch den Datentyp `BFile` steht immer auch die gesamte API zur Manipulation der externen Datei aus der Datenbank heraus zur Verfügung.

LOB-Datentypen können als Secure Files in der Datenbank gespeichert werden. Diese Option stellt einen leistungsfähigen Zugang zu großen Objekten dar und unterstützt Oracles Vision von einem Dateisystem innerhalb der Datenbank. Diese Option ist sicher hochinteressant für den schnellen Zugriff auf die Daten. Kapitel 15, »Arbeiten mit LOBs (Large Objects)«, beschäftigt sich näher mit dem Thema, daher können wir an dieser Stelle auf eine nähere Diskussion verzichten.

4.5.5 Benutzerdefinierte Typen, XML, JSON

Oracle ist schon seit Version 8i weit mehr als eine relationale Datenbank. Seit dieser Zeit (Ende der 90er-Jahre) bietet Oracle eine Reihe objektorientierter Erweiterungen an, um die Schnittstelle zwischen objektorientierter Programmierung und relationalen Datenbanken überwinden zu helfen. Diese Typen werden wir uns im weiteren Verlauf des Buches noch ansehen, hier reicht es uns, festzustellen, dass Sie bei Oracle eigene Datentypen mit mächtigen Funktionen definieren und verwenden können. Diese Datentypen gehören ebenfalls zu Ihrem Schema.

Ebenfalls auf Basis der objektorientierten Erweiterungen ist die XML-Unterstützung der Datenbank entwickelt worden. Zu diesen Erweiterungen gehören der Oracle-Datentyp `XmlType`, die SQL-Erweiterung XML/SQL, die unter anderem auch XQuery implementiert, die XMLDB sowie eine Fülle von Programmierwerkzeugen für den Umgang mit XML. Da wir uns auch XML noch gesondert ansehen werden, genügt uns hier dieser erste Ausblick. Als weitere Datentypen dieser Gruppe möchte ich JSON erwähnen, der in Version 12c neu hinzugekommen ist, sowie Mediadaten wie Bilder, Audio- oder Videodateien und Ähnliches sowie ein Format zur Speicherung von Geodaten oder medizinischen Bildformaten. All diese Typen sind nicht nur als Datentypen zur Speicherung in der Datenbank da, sondern bringen jeweils eine umfangreiche Funktionalität zur Manipulation dieser Datentypen mit.

4.5.6 Weitere Datenbankobjekte

Die weiteren Datenbankobjekte sind im Moment nicht so wesentlich, sie werden bei Bedarf im weiteren Verlauf des Buches besprochen.

4.6 Exkurs: Zeichensatzcodierung

Aus meiner Erfahrung aus vielen Kursen zu Oracle (sowohl für Programmierer als auch für Administratoren) weiß ich, dass das Thema Zeichensatzcodierung generell und deren Unterstützung in der Datenbank ein häufig unterschätztes Problem darstellt. Leider kennen viele nur zu gut die Auswirkungen dieser Problematik, ohne sie allerdings mit diesem Problem in Verbindung zu bringen. Daher werde ich an dieser Stelle einen kleinen Exkurs in die Grundlagen von Zeichensatzcodierungen einfügen und Ihnen zeigen, auf welche Weise Oracle dies unterstützt.

4.6.1 Zeichensatzcodierung im Überblick

Es gibt zwei grundlegende Techniken der Zeichensatzcodierung, zwischen denen wir unterscheiden müssen: die Single-Byte- und die Multi-Byte-Codierungen. Der Hintergrund dieser Diskussion ist, dass historisch für ein Zeichen eines Zeichensatzes ein Byte als Codierung verwendet wurde. Da ein Byte 256 verschiedene Zustände codiert, können in einem Zeichensatz dieser Codierung ebenso viele verschiedene Zeichen codiert werden. Die Grundlage aller Codierungen, die heute verwendet werden, ist, dass die wichtigsten lateinischen Buchstaben, arabischen Zahlen und Steuerzeichen der westlichen Schriftsysteme mit der Hälfte dieser Zeichenmenge auskommen. Da diese Grundzeichen bereits seit den Anfängen der Computerprogrammierung verwendet werden und früh standardisiert wurden, bilden diese Zeichen sozusagen das Rückgrat aller verwendeten Zeichensatzcodierungen. Sie wurden im *American Standard Code for Information Interchange* (ASCII) 1967 standardisiert und sind seitdem unverändert.

Komplizierter wird die Situation durch den Anspruch der verschiedenen Länder, auch deren jeweilige Schriftsonderzeichen darstellen zu können. Für den westeuropäischen Bereich sind dies vor allem die Umlaute, Akzente und Ligaturen (wie das *ß*), die ausschließlich in diesen Ländern verwendet werden. Doch ist selbstverständlich der Anspruch der Griechen, Russen, Araber etc. ebenso gerechtfertigt, deren jeweilige Sonderzeichen und differierenden Alphabete anzeigen zu können. Dabei reden wir noch nicht von den japanischen oder chinesischen Schriftzeichen, von den indischen Schriften, von Hebräisch etc. Leider ist die Einführung dieser Sonderzeichen nicht so unproblematisch verlaufen, denn während der Integration dieser Zeichen war die Computerindustrie der Hoffnung, mittels proprietärer Codierung der Sonderzeichen Marktanteile sichern zu können. So existierten (und existieren immer noch) Zeichensatzcodierungen für Windows (z. B. Win-1252), Macintosh (z. B. MacRoman) etc., die sich in der Codierung der Sonderzeichen unterscheiden. Daher können Inhalte in einer Codierung oft nur mit Schwierigkeiten in anderen Codierungen angezeigt werden, vor allem wenn die Datei die Information über ihre Codierung

nicht mehr trägt. Diese Diskussion ist zwar längst vorbei, doch behalten wir aus dieser Zeit immer noch die Vielzahl zueinander inkompatibler Zeichensatzcodierungen zurück: Oracle listet derzeit noch 222 unterstützte Zeichensatzcodierungen auf. Diese Vielfalt bereitet auch heute noch unvorhergesehene Probleme.

In dieser Situation wird durch die ISO die Anordnung der Sonderzeichen in Zeichensätzen international standardisiert, und zwar in der Norm *ISO-8859*. Diese Norm wird ergänzt durch regionsspezifische Sonderzeichenseiten, die als Teilnormen nummeriert und an den Standard angehängt werden. So ist z. B. die Norm ISO-8859-1 die Codierung für den westeuropäischen Bereich, ISO-8859-5 codiert kyrillische, ISO-8859-7 griechische Alphabete etc. Mittlerweile ist die Teilnorm für den westeuropäischen Bereich ISO-8859-1 (Latin-1) durch die ISO-8859-15 (Latin-9) ergänzt worden, da diese (unter anderem) das Eurozeichen enthält.

All diese Normen beziehen sich auf Single-Byte-Zeichensätze, da zur Speicherung eines Buchstabens nach wie vor ein Byte verwendet wird. Diese Normen werden heutzutage allerdings nicht mehr weiterentwickelt, denn seit vielen Jahren versucht man, die Probleme der Zeichensatzcodierung auf andere Weise zu lösen. Die Multi-Byte-Zeichensätze verfügen im Gegensatz zu den älteren Single-Byte-Zeichensätzen über mehrere Byte pro Buchstabe. Daher können diese Codierungen sehr viel mehr unterschiedliche Zeichen codieren und machen es nicht mehr erforderlich, mittels verschiedener Teilnormen einen Zeichenvorrat für eine Region definieren zu müssen. Gerade für Hersteller von Produkten, die in vielen Ländern verkauft werden, sind diese Zeichensätze insofern von großem Vorteil, als sie alle Sonderzeichen aller benötigten Sprachen in einer einheitlichen Codierung enthalten.

Diese Zeichensätze werden im Standard *Unicode* definiert. In den ersten Unicode-Versionen, die als UTF-16 zwei (oder vier) Byte pro Zeichen vereinbaren, beinhaltete der Standard auch das (im Übrigen als offizielle Sprache anerkannte) Klingonisch, das auch heute noch aktiv von vielen gesprochen wird. (Es gibt Übersetzungen von Shakespeare ins klingonische »Original« sowie ein Projekt für ein Wörterbuch Klingonisch – Altägyptisch, an dem Sie sich gern beteiligen dürfen, sollten Sie sonst nichts zu tun haben.) In der Zwischenzeit ist Unicode in Version 10.0 (veröffentlicht Juni 2017) verfügbar. Es codiert knapp 137.000 verschiedene Zeichen. Die Speicherung erfolgt je nach Standard unterschiedlich. Zum einen ist der Standard UTF-16 zu nennen, der die Zeichen in ein oder zwei je zwei Byte langen Einheiten speichert. Dann existiert UTF-32, das für jedes Zeichen 4 Byte verwendet. Interessant und weit verbreitet ist allerdings der Standard UTF-8, der die Zeichen in 1 bis 4 Byte langen Worten speichert. Dabei (und das ist eine nachvollziehbare Forderung des angloamerikanischen Sprachraums) werden die Zeichen aus dem ursprünglichen ASCII-Vorrat nur mit einem Byte gespeichert, Sonderzeichen wie die Umlaute oder auch Zeichen aus anderen Alphabeten werden mit mehreren Byte gespeichert.

4.6.2 Zeichensatzcodierung bei Oracle

Oracle unterstützt beinahe alle geläufigen Zeichensatzcodierungen, insbesondere natürlich auch ISO-8859 und UTF. Die folgende Abfrage listet nur die unterstützten Codierungen dieser beiden Normen auf:

```
SQL> select value
  2    from v$nls_valid_values
  3   where parameter = 'CHARACTERSET'
  4     and isdeprecated = 'FALSE'
  5     and (value like '%UTF%' or value like '%ISO%')
  7   order by value
VALUE
---------------
AL16UTF16
AL32UTF8
AR8ISO8859P6
AZ8ISO8859P9E
BLT8ISO8859P13
CEL8ISO8859P14
CL8ISOIR111
CL8ISO8859P5
EE8ISO8859P2
EL8ISO8859P7
IW8ISO8859P8
LA8ISO6937
NEE8ISO8859P4
NE8ISO8859P10
SE8ISO8859P3
UTFE
UTF8
WE8ISOICLUK
WE8ISO8859P1
WE8ISO8859P15
WE8ISO8859P9
21 Zeilen ausgewählt.
```

Listing 4.11 Abfrage der unterstützten Zeichensatzcodierung

Sie erkennen, dass in der Datenbank für die Codierungen interne Bezeichnungen verwendet werden, die Sie kennen müssen, um die Datenbank z. B. auf eine andere Zeichensatzcodierung umstellen zu können. Die geläufigsten Codierungen einer Datenbank in Mitteleuropa sind `WE8ISO8859P1`, besser noch `WE8ISO8859P15` wegen des €-Zeichens, und die Unicode-Codierungen `AL32UTF8`, seltener `AL16UTF16` und `UTF8`.

Wichtig ist, dass die Zeichensatzcodierung, in der Oracle die Daten speichert, grundsätzlich beim Aufsetzen der Datenbank angegeben werden muss und anschließend nicht mehr (ohne Weiteres) verändert werden kann. Das ergibt auch Sinn: Zum einen ändert eine umgestellte Codierung beinahe alle Daten in der Datenbank, denn ein Umlaut ist in einer Zeichensatzcodierung ja auf einem anderen Platz hinterlegt als auf einer anderen Codierung: Beim Wechsel einer Single-Byte-Codierung auf eine Multi-Byte-Codierung änderte sich zudem auch noch der Platzbedarf jeder einzelnen Zelle. Leider ist der Dialog zur Definition der Zeichensatzcodierung im *Datenbank-Konfigurationsassistenten* (DBCA) von Oracle etwas versteckt und wird gern mit seinem Standardwert übernommen. Dieser Standardwert ist abhängig vom Betriebssystem sowie der Datenbankversion und lautet bei Windows z. B. `Win1252`, ist mithin also eine proprietäre Microsoft-Codierung, die aber, und das ist dann sozusagen Glück im Unglück, kompatibel mit ISO-8859-1 ist. Ab Version 12.2 wird Unicode als Standardcodierung der Datenbank auf allen Betriebssystemen eingesetzt. Da die meisten Datenbanken die Daten eines Unternehmens speichern und in diesen Unternehmen eventuell viele unterschiedliche Zeichensatzcodierungen zum Einsatz kommen, ist diese Vorauswahl aber nicht sinnvoll und muss vom Administrator der Datenbank beim Aufsetzen der Datenbank mit Bedacht gewählt werden.

Ist die Zeichensatzcodierung einmal gewählt, sind die Im- und Exportprogramme von Oracle normalerweise in der Lage, die verschiedenen Codierungen korrekt aufeinander abzubilden. Daher können die verschiedenen Single-Byte-Zeichensätze normalerweise ohne Informationsverlust in die Datenbank eingespielt werden. Das muss aber für Ihre eigene Programmierung in PL/SQL nicht notwendigerweise gelten! Als Beispiel: Der Import von Daten über das Werkzeug `impdp` gelingt normalerweise problemlos, denn dieses Programm konvertiert den Zeichensatz. Erstellen Sie allerdings Daten über eine SQL-Skriptdatei, die in einer von der Datenbank abweichenden Codierung gespeichert ist, werden Sie Ihre Umlaute verlieren, denn zum einen ist in den Skriptdateien die Information ihrer Codierung nicht enthalten, zum anderen konvertiert SQL die Daten eben nicht. Achten Sie beim Gebrauch einer Skriptdatei zudem darauf, dass SQL*Plus auf die Zeichensatzcodierung der Skriptdatei eingestellt ist. Haben Sie z. B. eine Skriptdatei in UTF-8, die Sie mit SQL*Plus in die Datenbank einspielen möchten, müssen Sie vor Aufruf von SQL*Plus die Umgebungsvariable `NLS_LANG` auf UTF-8 einstellen wie im folgenden Beispiel:

```
set nls_lang=AMERICAN_AMERICA.AL32UTF8
```

Listing 4.12 Einstellung der NLS-Umgebungsvariablen »NLS_LANG« auf Unicode-Codierung

Zudem gibt es naturgemäß Probleme, wenn Sie Daten in einem Multi-Byte-Zeichensatz vorliegen haben und in eine Datenbank importieren, die nur über einen Single-Byte-Zeichensatz verfügt. Dann sind Informationsverluste je nach Daten nicht zu

vermeiden. Bei der Programmierung mit Zeichendaten müssen Sie darüber hinaus die gesamte Verarbeitungskette daraufhin durchsehen, ob eine konsistente Textverarbeitung gewährleistet ist. Wird irgendwo in der Verarbeitungskette eine Umformung vorgenommen, sind die Probleme normalerweise kaum noch zu lösen.

Um diese Probleme in den Griff zu bekommen, bietet Oracle Ihnen eine zweite Zeichensatzcodierung an. Diese kann seit Version 9 der Datenbank nur ein Multi-Byte-Zeichensatz sein, also z. B. UTF-8 oder UTF-16. Nun können sowohl alte Datenbestände in einer Single-Byte-Codierung als auch neuere Daten in einer Multi-Byte-Codierung gespeichert werden. Zur Unterscheidung dieser beiden Codierungen bietet Oracle alle zeichenorientierten Datentypen sowohl in einer Variante mit der Grundcodierung als auch in einer Variante mit der Zusatzcodierung der Datenbank an. Die Grundcodierung wird bei den bekannten Datentypen `varchar2` etc. verwendet, während den Zusatzcodierungen ein `n` für *National Language Support* (NLS) vorangestellt wird. Daher ist der Datentyp `nvarchar2` ebenso in der Zusatzcodierung codiert wie der Datentyp `nclob`. Anders gesagt: Wenn Sie z. B. einen Text in UTF-8 in eine Datenbank speichern möchten, die als Codierung ISO-8859-15 verwendet, ist es relativ sicher, dass Sie Daten verlieren werden, wenn Sie nicht eine Spalte vom Typ `nclob` zur Speicherung verwenden. Natürlich hat diese Flexibilität ebenfalls ihren Preis: Sie müssen bei der Gestaltung des Datenmodells bereits darauf achten, die Informationen, die in einer abweichenden Codierung gespeichert werden sollen, mit dem entsprechenden Datentyp zu definieren.

Einen weiteren Punkt müssen Sie im Auge behalten: Wenn Sie einen zeichenorientierten Datentyp verwenden, der in einer Multi-Byte-Codierung abgelegt wird (egal, ob `nvarchar2` oder `varchar2` in einer Multi-Byte-Datenbank verwendet wird), müssen Sie daran denken, dass die Spaltenlänge als Default immer in Byte kalkuliert wird (es sei denn, Sie hätten den Startparameter `nls_length_semantics` auf den Wert `char` eingestellt, was nicht die Standardeinstellung ist). Daher kann es sein, dass ein Nachname mit zwölf Buchstaben nicht in eine `(n)varchar2(12)`-Spalte passt. Achten Sie bei Multi-Byte-Codierung darauf, dass Sie bei der Deklaration der Spalte explizit angeben, dass zwölf Zeichen gespeichert werden können sollen, selbst dann, wenn die Datenbank `nls_length_semantics` auf `char` gestellt hat. Denn wenn der Startparameter geändert wird oder Sie Ihre Datenbankobjekte auf einer anderen Datenbank erzeugen wollen, die diesen Parameter nicht umgestellt hat, werden Sie Probleme mit der Speicherung Ihrer Daten bekommen. Unabhängig von diesen Einstellungen – auch das als Erinnerung – wird allerdings die maximale Länge einer `varchar2`-Spalte immer 4.000 *Byte* betragen (ab Version 12 optional 32.767 Byte), niemals 4.000 *Zeichen* in einer Multi-Byte-Codierung. Natürlich *könnten* 4.000 Zeichen in eine solche Spalte passen, wenn alle Zeichen aus ASCII kommen; das erste Zeichen außerhalb dieses Zeichenvorrats wird allerdings die Gesamtzahl der Zeichen reduzieren.

Beherzigen Sie in Ihren Projekten und Datenmodellen bitte die Best Practice, Zeichenkettenspalten immer mit dem Zusatz `char` zu deklarieren, selbst wenn Ihre Datenbank eine Single-Byte-Codierung einsetzt. Der Grund hierfür liegt darin, dass beim Zugriff auf diese Datenbank durch eine Unicode-Datenbank über einen Datenbank-Link extrem hinterhältige Fehlersituationen auftauchen können, weil Variablen, die aufgrund der Festlegung Ihrer Datenbank definiert werden, und die Werte Ihrer Datenbank nicht in diese Variablen passen.

An dieser Diskussion erkennen Sie, wie problematisch die Situation durch die Vielzahl der verwendeten Zeichensatzcodierungen ist. Es ist mit Sicherheit sinnvoll, die gesamte Verarbeitungsumgebung innerhalb Ihres Unternehmens auf eine Zeichensatzcodierung zu normieren. Haben Sie viel mit international codierten Daten zu tun (oder aber auch mit XML und/oder Java, die standardmäßig UTF-8 verwenden), empfiehlt sich ein Multi-Byte-Zeichensatz, bevorzugt UTF-8 (`AL32UTF8`, da diese das Gros der europäischen und amerikanischen Zeichen mit nur einem Byte Länge speichert). Da ab Version 12.2 ohnehin Unicode als Standardcodierung in der Datenbank verwendet wird, empfehle ich nun auch die Single-Byte-Codierungen nicht mehr.

Sollten Sie die Zeichensatzcodierung ändern müssen, stehen Ihnen in Version 11g noch zwei Werkzeuge zur Verfügung, mit denen Sie das erledigen (lassen) können: der *Character Set Scanner* und ein SQL-Skript mit dem Namen *csalter.sql*. Diese beiden Werkzeuge analysieren die vorhandenen Daten und prüfen, ob die Datenbank migriert werden kann. Sie sind allerdings ab Version 12c als `deprecated` eingestuft und wurden durch eine in den Datenbank-Migrationsassistenten integrierte Lösung ersetzt, deren Fokus die Migration hin zu Unicode ist. Auch daran erkennen Sie, dass die Zeit der Singe-Byte-Codierungen langsam, aber sicher abläuft. Gut so, ich bin durchaus dafür, diese Probleme ein für alle Mal zu lösen.

Alternativ ist es auch möglich, eine Datenbank mit der neuen Zeichensatzcodierung leer neu aufzusetzen und anschließend mit den Im- und Exportprogrammen die Daten in die neuen Strukturen fließen zu lassen. Möchten Sie eine bestehende Datenbank migrieren, die z. B. in `Win1252` codiert ist, dann kann es, aufgrund des eventuell größeren Platzbedarfs, problematisch sein, diese Daten auf eine Multi-Byte-Codierung zu portieren. Dies gilt insbesondere dann, wenn eine Schlüsselspalte den neuen Wert nicht mehr aufnehmen kann, da er in der neuen Codierung ein oder zwei Byte mehr Platz beansprucht, als die Spalte Platz bietet. Abhilfe schafft dann nur, die Spalten auf eine Char-Semantik umzustellen. Allerdings ist mir für diese Arbeit kein Standardverfahren bekannt. Sie werden in diesen Fällen um die Programmierung einer Hilfsprozedur in PL/SQL nicht herumkommen, insbesondere dann nicht, wenn Ihr Datenmodell aus sehr vielen Tabellen besteht. Zudem habe ich gesagt, dass es *möglich* ist, eine Zeichensatzkonvertierung durchzuführen. Ich habe weder gesagt, dass dies unproblematisch funktioniert, noch, dass es schnell geht ...

4.7 Mächtigkeit von SQL

Als weiteren Aspekt der Datenbank ist es unbedingt notwendig, sich in die Mächtigkeit der Sprache SQL einzuarbeiten. Wie bereits ganz zu Beginn dieses Buches festgestellt, ist PL/SQL »lediglich« eine Erweiterung von SQL. Nichts geht in der Datenbank ohne SQL. Sie lesen oder ändern keine Daten, erhalten keine Auskunft über die Datenbank und können keine Datenbankobjekte anlegen und die Datenbank administrieren. So weit, so bekannt. Etwas weniger bekannt ist die Mächtigkeit der Oracle-Implementierung der Sprache SQL. Der *Optimizer*, ein Programm, das für den Ausführungsplan, d. h. für die Strategie der Abarbeitung einer `select`-Anweisung verantwortlich ist, ist unglaublich gut darin, komplexe logische Probleme zu optimieren und eine extrem schnelle Bearbeitung sicherzustellen. Dies kann er aber nur, wenn Sie als Programmierer der Datenbank die Chance dazu geben. Jede Stunde, die Sie in die Verbesserung Ihrer SQL-Kenntnisse stecken, ist gut investierte Zeit, die es Ihnen ermöglichen wird, weniger Code zu schreiben und die Performance Ihrer Anwendung deutlich zu verbessern.

Die Sprache SQL ist ungeheuer mächtig, insbesondere in der Implementierung von Oracle. Als Beispiel für die Mächtigkeit zeige ich Ihnen gern die Bereiche *analytische Funktionen*, *hierarchische Abfragen* und *Error Logging*. In diesem Buch wird eine gewisse Grundkenntnis von SQL vorausgesetzt, doch wäre es vermessen, die Kenntnis solcher Spezialfunktionen von Ihnen zu erwarten. Sehen wir uns daher einmal ein kurzes Beispiel für diese Bereiche an, das natürlich lediglich das grobe Prinzip darstellt und bei Weitem nicht erschöpfend die Möglichkeiten dieser Erweiterungen zeigt. Ich habe diese Funktionsbereiche von SQL gewählt, weil ich glaube, dass diese Funktionen in Ihren Anwendungen häufig eingesetzt werden können. Oracle liefert eine Überfülle weiterer Optimierungen, die Ihnen bei der Lösung Ihrer Anwendungsprobleme helfen. Daher gehört ein gutes Oracle-SQL-Buch in jedem Fall in Ihre Bibliothek. Wie wäre es z. B. mit meinem Buch zum Thema? Zumindest finden Sie in diesem Buch auch die nachfolgenden sowie noch weitere, exotischere Funktionen, wie etwas das *Pattern Matching*, mit dem die Mustersuche in Tabellendaten möglich ist.

4.7.1 Analytische Funktionen

Häufig werden Daten für Berichte benötigt. Diese Berichte müssen nach Kriterien gruppiert und geordnet werden, laufende Salden müssen kalkuliert und ausgegeben werden. Diese Funktionalität ist in SQL mittels der Gruppenfunktionen wie `sum`, `avg`, `count`, `max` oder `min` implementiert und kann dort schnell und einfach genutzt werden. Problematisch wird es allerdings, wenn innerhalb eines einzelnen Berichts unterschiedliche Gruppierungsebenen verwendet werden sollen oder wenn auf vorangegangene Zeilen Bezug genommen werden soll, etwa bei der Kalkulation eines laufenden Durchschnitts über die Daten der letzten 30 Tage. Die Folge ist bei herkömmlichem SQL eine aufwendige Abfrage mit vielen Tabellen-Aliassen, auf die Bezug genommen werden muss. Diese ISO-SQL-konforme Abfrage skaliert, grob gesagt, exponentiell, die doppelte Zeilenzahl hat also etwa eine Vervierfachung der Antwortzeit zur Folge. Analytische Funktionen treten an, um solche Auswertungen linear zu skalieren und gleichzeitig drastisch zu beschleunigen.

Sehen wir uns ein Beispiel an: In einem Mitarbeiterbericht sollen folgende Angaben gemacht werden:

- Es soll daraus hervorgehen, wie hoch der Anteil des Gehalts eines Mitarbeiters am Gesamtgehalt der Abteilung ist.
- Zudem soll eine akkumulierte Darstellung der Gehälter, sortiert nach Gehalt, pro Abteilung und für das Gesamtunternehmen dargestellt werden.
- Schließlich soll die Differenz zwischen dem aktuellen Gehalt und dem Gehalt des nächstschlechter verdienenden Mitarbeiters der gleichen Abteilung gezeigt werden.

Mit herkömmlichem ISO-kompatiblen SQL müssen nun folgende Voraussetzungen erfüllt sein:

- Es müssen mehrere Unterabfragen für die unterschiedlichen Gruppierungen erstellt werden.
- Richtig interessant ist aber die Erstellung einer akkumulierten Darstellung der Einzelgehälter. Grundsätzlich muss mit einer harmonisierten Unterabfrage für jedes Gehalt die Summe der Gehälter berechnet werden, die (in der gleichen Abteilung) kleiner oder gleich dem aktuellen Gehalt sind.
- Zudem benötigen wir eine harmonisierte Unterabfrage, um den Mitarbeiter mit dem nächstniedrigen Gehalt der gleichen Abteilung zu finden, um die Differenz zum aktuellen Gehalt zu berechnen.

Eine solche Abfrage ist nicht mehr trivial und mag dazu führen, dass Ihnen als Entwickler die Implementierung in einer Programmiersprache leichter fällt.

Sehen wir uns allerdings das SQL an, das Oracle für diese Aufgabenstellung benötigt, so fällt zunächst (neben der etwas ungewohnten Syntax) die Kürze und Einfachheit der Anweisung auf:

```
SQL> select department_id dept, last_name, salary sal,
  1         sum(salary) over
  2            (partition by department_id
  3             order by salary, last_name) s_d_sal,
  4         sum(salary) over
  5            (order by department_id, salary, last_name) sum_sal,
  6         salary - lag(salary) over
  7                     (partition by department_id
  8                      order by salary) diff_sal,
```

```
 9         round(ratio_to_report(salary) over
10              (partition by department_id) * 100, 1) "%_DEPT"
11    from employees
12   order by department_id, salary, last_name;
DEPT LAST_NAME       SAL    S_D_SAL SUM_SAL DIFF_SAL %_DEPT
---- --------------- ------ ------- ------- -------- ------
10   Whalen            4400    4400    4400           100,0
20   Fay               6000    6000   10400            31,6
20   Hartstein        13000   19000   23400    7000    68,4
30   Colmenares        2500    2500   25900            10,0
30   Himuro            2600    5100   28500     100    10,4
30   Tobias            2800    7900   31300     200    11,2
30   Baida             2900   10800   34200     100    11,6
30   Khoo              3100   13900   37300     200    12,4
30   Raphaely         11000   24900   48300    7900    44,2
40   Mavris            6500    6500   54800           100,0
50   Olson             2100    2100   56900             1,3
50   Markle            2200    4300   59100     100     1,4
50   Philtanker        2200    6500   61300       0     1,4
50   Gee               2400    8900   63700     200     1,5
50   Landry            2400   11300   66100       0     1,5
50   Marlow            2500   13800   68600     100     1,6
50   Patel             2500   16300   71100       0     1,6
...
107 Zeilen ausgewählt.
Abgelaufen: 00:00:00.11
```

Listing 4.13 Beispiel für analytische Funktionen in einer »select«-Anweisung

In dieser Variante sind weder mehrere virtuelle Sichten auf die Tabelle noch harmonisierte Unterabfragen erforderlich. Wenn wir uns die Anweisung ansehen, fällt zunächst einmal die Erweiterung der Gruppenfunktionen wie `sum` oder `avg` etc. durch das Schlüsselwort `over` auf. Jede Gruppenfunktion (sogar die, die wir selbst in PL/SQL noch programmieren werden!) kann auf diese Weise in eine analytische Funktion überführt werden. In der Klammer hinter dem Schlüsselwort `over` habe ich die `partition by`-Klausel sowie die `order by`-Klausel verwendet. Diese beiden Klauseln nehmen die Gruppierung der Zeilen *für diese Spalte* vor. Da wir nun nicht mehr die ganze Tabelle gruppieren (das ginge zwar immer noch, würde aber durchgeführt, *bevor* die analytischen Funktionen rechnen), steht uns nun ein Mechanismus zur Verfügung, um *unterschiedliche* Gruppierungsregeln auf *ein* Tabellenalias anzuwenden. Zudem stellt Oracle einige Funktionen zur Verfügung, die nur als analytische Funktionen nutzbar sind: die Funktion `ratio_to_report` z. B., die den Anteil einer Spalte an einem Gesamtwert (z. B. gruppiert nach Abteilung) berechnet, oder die Funktionen `lag` und

`lead`, die uns erlauben, auf einen (bezogen auf ein Sortierkriterium) vorangegangenen oder nachfolgenden Datensatz zuzugreifen.

Analytische Funktionen erlauben es Ihnen, auf einfache Weise Rankings zu erzeugen, Vergleiche mit vorangegangenen Zeilen durchzuführen oder mit Zeilenmengen über ein Sortierkriterium zu arbeiten, und erleichtern die Erstellung komplexer Berichte innerhalb der Datenbank. Zudem skaliert diese Variante linear und ist um mehrere Faktoren schneller als ihr ISO-kompatibles Pendant. Möchten Sie sich intensiver über analytische Funktionen informieren, empfehle ich Ihnen den *Data Warehousing Guide* zu Ihrer Datenbankversion. Diese PDF-Datei enthält ein Kapitel über analytische Funktionen mit sehr vielen, zum Teil recht komplexen Beispielen. Auch diese Beispiele können Sie direkt gegen die Schemata `OE`, `SH` oder `HR` ausführen.

4.7.2 Hierarchische Abfragen

Eine Standardanforderung an SQL ist die Behandlung hierarchischer Abfragen. In vielen Datenmodellen findet sich der einfachste Fall einer solchen Hierarchie in Form von zwei Spalten einer Tabelle. Am Beispiel der `employees`-Tabelle aus dem Oracle-Beispielschema `HR` sehen wir eine solche Implementierung an den Spalten `employee_id` und `manager_id`. Die Idee: Jeder Mitarbeiter hat eine Mitarbeiternummer. Zudem hat jeder Mitarbeiter (außer dem Chef) einen Vorgesetzten. Welcher Vorgesetzte das ist, wird in der Spalte `manager_id` notiert. Sie stellt die Mitarbeiternummer des Vorgesetzten dar. Ausgehend von einem Mitarbeiter, können Sie so die Hierarchie des Unternehmens nachvollziehen, indem Sie jeweils die Vorgesetzten ermitteln. Umgekehrt wird, ausgehend von dem Mitarbeiter, der keinen Vorgesetzten hat (`manager_id is null`), das Organigramm des Unternehmens sichtbar. Ausgehend von der Mitarbeiternummer des Chefs, können dessen direkte Untergebene gefunden werden, weil sie die Mitarbeiternummer des Chefs als `manager_id` vermerkt haben. Sicher ist eine solche Datenmodellierung sehr einfach, es kann z. B. nicht nachvollzogen werden, wie die historische Zuordnung eines Mitarbeiters zu einem Manager war, auch kann ein Mitarbeiter immer nur einen Vorgesetzten zur gleichen Zeit haben etc., doch sieht man eine solche Datenmodellierung relativ häufig. Denken Sie z. B. an Pfade eines Dateisystems, die auf diese Weise modelliert werden könnten.

Bei der Abfrage einer solchen Hierarchie sollte SQL also alle benötigten Mittel an Bord haben. Leider ist dies ein Beispiel dafür, auf welch einfache Art eine Weltsicht (hier die relationale, aber ähnliche Beispiele können wir für alle Weltsichten konstruieren) an den Rand ihrer Ausdrucksfähigkeit gebracht werden kann. Warum? Stellen wir uns eine Abfrage vor, die das Organigramm eines Unternehmens darstellt. Das Problem besteht nun darin, dass bei SQL für jede Tabelle, die durchsucht wird, *ein* Zeilenzeiger verwendet wird. Dieser Zeiger steht auf der aktuellen Zeile, die gerade bearbeitet wird. Soll nun zu dieser Zeile ein anderer Mitarbeiter gefunden werden,

benötige ich einen zweiten Zeiger, der die Tabelle durchsucht. Einen solchen zweiten Zeiger gibt es aber noch nicht. Um ihn zu erzeugen, muss ich von der gleichen Tabelle eine zweite Sicht ableiten, indem ich in SQL ein zweites Alias auf die gleiche Tabelle deklariere wie im folgenden Beispiel:

```
SQL> select m.last_name || ' manages ' || e.last_name chefs
  1    from employees e, employees m
  2   where e.manager_id = m.employee_id;
CHEFS
-----
...
King manages Weiss
King manages Raphaely
King manages De Haan
King manages Kochhar
Kochhar manages Higgins
Kochhar manages Baer
Kochhar manages Mavris
Kochhar manages Whalen
Kochhar manages Greenberg
De Haan manages Hunold
Hunold manages Lorentz
Hunold manages Pataballa
...
106 Zeilen ausgewählt.
Abgelaufen: 00:00:00.12
```

Listing 4.14 Eine einfache hierarchische »select«-Anweisung

So ist das noch kein Problem. Doch nun wollen wir das Organigramm so darstellen, dass die Abhängigkeiten untereinander klar werden. Ich möchte also zu jedem Untergebenen von King zunächst deren Untergebene in der entsprechenden hierarchischen Beziehung, *bevor* der nächste Untergebene von King bearbeitet wird. Nun benötige ich *für jede Ebene* des Organigramms ein Tabellenalias. Nur – wie viele Ebenen sind denn das? Und welche Auswirkung hat eine solche Abfrage bei, sagen wir, 15 Ebenen auf die Übersichtlichkeit der Abfrage und die Performance, denn immerhin müssen nun 15 Tabellen über Joins miteinander verbunden werden? Es kommt hinzu, dass wir nun Outer Joins benötigen, denn da wir nicht wissen, wie viele Ebenen existieren, würde ein Datensatz nicht angezeigt, wenn für ihn nicht mindestens ein Datensatz bis zur tiefsten geplanten Ebene vorläge.

Das erste Problem können wir überhaupt nicht mit »normalem« SQL lösen. Die Anzahl der Ebenen kann nicht beliebig tief geschachtelt werden, weil wir eine definierte Anzahl Tabellenaliasse benötigen. Erst sein Version 11.2 implementiert Oracle hierfür einen ISO-SQL-kompatiblen Weg über eine rekursive `with`-Klausel (die ich furchtbar finde ...). Das zweite Problem ist ebenso schwierig: Die Übersichtlichkeit einer solchen Abfrage geht gegen null, die Antwortzeit gegen unendlich, wie Sie aus dem folgenden Beispiel erahnen können:


```
SQL> select e_1.last_name, e_2.last_name, e_3.last_name
  1  from employees e_1
  2         left join employees e_2
  3              on e_1.employee_id = e_2.manager_id
  4         left join employees e_3
  5              on e_2.employee_id = e_3.manager_id
  6 where e_1.manager_id is null;
```

Listing 4.15 Eine komplexere hierarchische Abfrage in ISO-SQL

Denken Sie sich für jede weitere Ebene eine weitere Tabelle hinzu ...

Oracle bietet für solche Abfragen eine hochoptimierte Erweiterung an: die hierarchische Abfrage über das Schlüsselwort `connect by`. Sehen wir uns eine solche Abfrage einmal an:

```
SQL>  select level, lpad('.', 2 * (level - 1)) || last_name emp,
  1          employee_id emp_id, manager_id man_id
  2     from employees
  3    start with manager_id is null
  4  connect by prior employee_id = manager_id
  5    order siblings by last_name;
LEVEL EMP            EMP_ID MAN_ID
----- -------------- ------ ------
    1 King              100
    2 ..Cambrault       148    100
    3 ....Bates         172    148
    3 ....Bloom         169    148
    3 ....Fox           170    148
    3 ....Kumar         173    148
    3 ....Ozer          168    148
    3 ....Smith         171    148
    2 ..De Haan         102    100
    3 ....Hunold        103    102
    4 ......Austin      105    103
    4 ......Ernst       104    103
    4 ......Lorentz     107    103
    4 ......Pataballa   106    103
```

```
    2 ..Errazuriz       147    100
    3 ....Ande          166    147
    3 ....Banda         167    147
    3 ....Greene        163    147
    3 ....Lee           165    147
    3 ....Marvins       164    147
...
107 Zeilen ausgewählt.
Abgelaufen: 00:00:00.34
```

Listing 4.16 Abfrage mit der »connect by«-Klausel

Die Anfrage nutzt neben dem Schlüsselwort `connect by` noch die Pseudospalte `level`, die, ähnlich wie die Spalte `rownum`, in SQL-Anweisungen genutzt werden kann. Die Pseudospalte `level` ist allerdings nur in Kombination mit einer hierarchischen Abfrage erlaubt und zeigt die Schachtelungstiefe an. Mit der Klausel `start with` geben wir an, wo der Einstieg in die Hierarchie liegen soll. In unserem Beispiel suchen wir einen Mitarbeiter ohne Manager, also den Chef. Schließlich wird die `order by`-Klausel in unserem Beispiel noch durch das Schlüsselwort `siblings` erweitert, was die Datenbank anweist, die Sortierung innerhalb der jeweiligen Ebene durchzuführen, und zwar nur für die Kindelemente eines gemeinsamen Elternelements. Diese Sortierung wäre mit ISO-SQL extrem aufwendig.

Auch diese Abfrage kommt ohne weitere Sichten auf die gleiche Tabelle aus, unterstützt beliebig viele Ebenen und skaliert linear. Zudem löst sie noch weitere, bei näherem Nachdenken in ISO-SQL nur äußerst schwer zu lösende Probleme, wie z. B. die Sortierung der Mitarbeiter innerhalb einer hierarchischen Ebene nach Name, das Entdecken von Zirkelbezügen in der Hierarchie und viele weitere Funktionen.

4.7.3 Error Logging

Das Problem, das durch das Error Logging gelöst wird, ergibt sich aus der *Atomizität* von SQL-Anweisungen: Entweder gelingt die gesamte Anweisung, oder sie wird komplett zurückgenommen. Normalerweise ist das natürlich eine sehr segensreiche Eigenschaft von SQL, doch gibt es auch Situationen, in denen man sich etwas mehr Kontrolle erhofft. Stellen wir uns eine Anweisung vor, die 10.000 Zeilen in eine Tabelle als Ergebnis einer `select`-Anweisung einfügt. Die Tabelle, in die die Zeilen gefüllt werden sollen, wird durch Constraints und Trigger geschützt. Nun reicht es, dass bei einer der 10.000 Zeilen ein Fehler auftaucht, um die gesamte `insert`-Anweisung rückgängig zu machen. Schöner wäre ein Mechanismus, der diese Fehler abfängt und die fehlerhaften Zeilen in eine Fehlertabelle schreibt. Nachdem die `insert`-Anweisung vollständig abgearbeitet worden ist, können wir uns dann um die fehlerhaften Daten kümmern.

In der Vergangenheit waren es Anforderungen wie diese, die Entwickler zu PL/SQL greifen ließen: Nur hier war es möglich, eine Zeile einzufügen und im Fehlerfall den Fehler abzufangen und den fehlerhaften Datensatz zu ignorieren. Das Hauptproblem, das wir hier haben, ist allerdings bereits im letzten Satz angeklungen: *eine* Zeile. PL/SQL zwingt uns, das Einfügen der Daten zu serialisieren, um bei jeder Zeile zu prüfen, ob ein Fehler vorliegt oder nicht. Da diese Problematik häufig auftaucht, zwingt dies die Entwickler zudem, immer gleichen Code für verschiedene Tabellen zu implementieren. Doch es geht auch anders: Oracle bietet für alle DML-Befehle (`insert`, `update`, `delete` und `merge`) die Option, Fehler automatisch in eine Fehlertabelle schreiben zu lassen. Sehen wir uns einmal an, wie so etwas funktioniert.

In unserem Beispiel möchten wir 100 Zeilen in die Tabelle `ORDERS` einfügen lassen. Wir wissen, dass wir in etwa 3–5 % der Zeilen Probleme mit den Daten haben könnten, möchten uns aber erst nach dem Einfügen mit diesen Problemen beschäftigen. Um dieses Problem zu lösen, gehen wir einen etwas umfangreicheren Weg als minimal erforderlich, doch bietet uns diese Vorgehensweise die übersichtlichste Information. Sehen wir uns zunächst einmal die Definition der Tabelle `ORDERS` an:

```
SQL> connect oe/oe
Connect durchgeführt
SQL> desc orders
Name          Null?    Typ
------------- -------- ------------------------------
ORDER_ID      NOT NULL NUMBER(12)
ORDER_DATE    NOT NULL TIMESTAMP(6) ITH LOCAL TIME ZONE
ORDER_MODE             VARCHAR2(8)
CUSTOMER_ID   NOT NULL NUMBER(6)
ORDER_STATUS           NUMBER(2)
ORDER_TOTAL            NUMBER(8,2)
SALES_REP_ID           NUMBER(6)
PROMOTION_ID           NUMBER(6)
```

Listing 4.17 Ausgabe der Tabellenstruktur

Wir beginnen damit, eine Tabelle erstellen zu lassen, die unsere fehlerhaften Daten aufnehmen soll. Dieser Schritt ist nicht unbedingt erforderlich. Unterlassen wir ihn, wird Oracle automatisch eine Fehlertabelle mit der Bezeichnung `ERR$_`, gefolgt von den ersten 25 Buchstaben des Tabellennamens, erstellen, in die die Daten eingefügt werden. Zudem müssen wir wissen, dass LOB-Datentypen sowie abstrakte Datentypen (ADT) nicht unterstützt werden.

Um die Tabelle zu erstellen, benutzen wir das mitgelieferte PL/SQL-Paket `dbms_errlog`. Dieses Paket ist in PL/SQL programmiert und stellt uns eine Methode zur automatischen Erstellung einer passenden Fehlertabelle zur Verfügung. Das nachfolgende Bei-

spiel zeigt den Aufruf, trotz der noch nicht komplett erläuterten Syntax. Doch denke ich, dass die Idee schon klar werden wird. Zwar haben wir in der Beispieltabelle keine nicht unterstützten Datentypen, doch möchten wir dennoch verhindern, dass eventuelle Fehler bei der Erstellung der Fehlertabelle auftreten, daher belegen wir den letzten Parameter mit dem Wahrheitswert `TRUE`:

```
SQL> begin
  2    dbms_errlog.create_error_log(
  3      dml_table_name => 'ORDERS',
  4      err_log_table_name => 'ERRLOG_ORDERS',
  5      skip_unsupported => true);
  6  end;
  7  /
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Die etwas seltsam anmutende Schreibweise `dml_table_name => 'ORDERS'` dient dazu, nur einige Parameter der Methode `dbms_errlog.create_error_log` gezielt mit Werten zu belegen. Wir werden dieses Prinzip später noch genauer erläutern. Sehen wir uns nach diesem Aufruf die Tabelle an, die für uns erstellt wurde:

```
SQL> desc ERRLOG_ORDERS
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)
ORDER_ID              VARCHAR2(4000)
ORDER_DATE            VARCHAR2(4000)
ORDER_MODE            VARCHAR2(4000)
CUSTOMER_ID           VARCHAR2(4000)
ORDER_STATUS          VARCHAR2(4000)
ORDER_TOTAL           VARCHAR2(4000)
SALES_REP_ID          VARCHAR2(4000)
PROMOTION_ID          VARCHAR2(4000)
```

Eine Frage, die hier auftauchen könnte, ist, warum die Datentypen aller Spalten der `ORDERS`-Tabelle zu `varchar2(4000)` verändert wurden. Der Grund ist, dass die Spaltenwerte in generischer Weise in die Ausgabetabelle übernommen werden. Dafür bietet sich das Format `varchar2` an, denn es kann sowohl Zahlen als auch Datumsangaben und Texte darstellen. Da zudem auch zu lange Texte für eine Spalte ausgegeben werden können sollen, ist die Länge auf den Maximalwert eingestellt. Alles andere wäre

auch eher seltsam: Wird eine Zeichenkette in der Zieltabelle abgewiesen, weil sie zu lang ist, und kann diese Zeichenkette anschließend nicht in die Errortabelle eingefügt werden, weil sie auch dafür zu lang ist, wäre das nicht im Sinne des Erfinders.

Nachdem nun der Boden bereitet worden ist, müssen wir einige Daten erzeugen, die wir in die Tabelle einfügen können. Ich habe dafür einfach eine Kopie der Daten der Tabelle erzeugt, den Primärschlüsselwert auf einen Wertebereich erweitert, der nicht in Gebrauch ist, und anschließend die Daten mit gezielten Fehlern »geimpft«:

```
SQL> insert into orders
  2         (order_id, order_date, order_mode,
  3          customer_id, order_status, order_total,
  4          sales_rep_id, promotion_id)
  5  select order_id + 200, sysdate, order_mode,
  6         case customer_id
  7           when 118 then 318
  8           else customer_id end,
  9         order_status,
 10         case customer_id
 11           when 116 then -3
 12           else order_total end,
 13         sales_rep_id, promotion_id
 14    from orders
 15   where rownum <= 100;
```

Mit dieser Anweisung wird dem Kunden `118` eine falsche Kundennummer mitgegeben sowie dem Kunden `116` eine ungültige Bestellmenge, die durch einen `check`-Constraint in der Tabelle `ORDERS` auf `>= 0` geprüft wird. Führen wir die Anweisung aus, erhalten wir einen Fehler zurück, der gleichzeitig dafür sorgt, dass die gesamte `insert`-Anweisung zurückgenommen wird:

```
insert into orders
*
FEHLER in Zeile 1:
ORA-02290: CHECK-Constraint (OE.ORDER_TOTAL_MIN) verletzt
```

Da wir jedoch eine Fehlertabelle vorbereitet haben, können wir diese Anweisung so ergänzen, dass die fehlerhaften Daten in die Fehlertabelle geschrieben werden und die `insert`-Anweisung gelingt:

```
SQL> insert into orders
  2         (order_id, order_date, order_mode,
  3          customer_id, order_status, order_total,
  4          sales_rep_id, promotion_id)
```

```
 5  select order_id + 200, sysdate, order_mode,
 6         case customer_id
 7           when 118 then 318
 8           else customer_id end,
 9         order_status,
10         case customer_id
11           when 116 then -3
12           else order_total end,
13         sales_rep_id, promotion_id
14    from orders
15   where rownum <= 100
16     log errors into errlog_orders ('daily_import')
17  reject limit 10;
96 Zeilen wurden erstellt.
```

Die Angabe am Ende der Anweisung weist die Datenbank an, fehlerhafte Daten in die Fehlertabelle zu schreiben. Gleichzeitig haben wir der `insert`-Anweisung einen Namen (Oracle nennt dies ein *Tag*) gegeben, damit die Fehler dieser Anweisung später in der Fehlertabelle (Spalte `ORA_ERR_TAG$`) leichter zu finden sind. Außerdem legen wir fest, dass wir mit höchstens zehn falschen Datensätzen rechnen. Würden mehr Fehler auftreten, würde die Anweisung zurückgenommen und ein entsprechender Fehler ausgegeben. Sehen wir uns nun die Tabelle `ERRLOG_ORDERS` an:

```
SQL> select ora_err_mesg$, ora_err_optyp$, order_id
  2    from errlog_orders;
ORA_ERR_MESG$                                      TYP ORDER_ID
-------------------------------------------------- --- --------
ORA-02290: CHECK-Constraint                          I     2636
  (OE.ORDER_TOTAL_MIN) verletzt
ORA-02290: CHECK-Constraint                          I     2569
  (OE.ORDER_TOTAL_MIN) verletzt
ORA-02291: Integritäts-Constraint                    I     2571
  (OE.ORDERS_CUSTOMER_ID_FK) verletzt -
  übergeordneter Schlüssel nicht gefunden
ORA-02290: CHECK-Constraint                          I     2628
  (OE.ORDER_TOTAL_MIN) verletzt
```

Listing 4.18 Beispiel für den Einsatz der »log errors«-Klausel

Beachten Sie in diesem Zusammenhang: Die Einträge in der Fehlertabelle bleiben bestehen, selbst wenn die `insert`-Anweisung mit `rollback` zurückgenommen wird. Die Einfügung in die Fehlertabelle wird von Oracle im Rahmen einer *autonomen*

Transaktion innerhalb der »eigentlichen« Transaktion, die durch die `insert`-Anweisung ausgelöst wurde, bearbeitet und überlebt daher auch das nachfolgende Zurücknehmen der `insert`-Anweisung.

Dieses Beispiel zeigt uns, auf welch einfache Weise Oracle die Arbeit mit Daten unterstützt: Im Extremfall wäre es lediglich erforderlich, die `insert`-Anweisung durch die `log errors`-Klausel zu erweitern. Natürlich könnte in einem realen Szenario diese Technik mit einer Methode kombiniert werden, die die Transaktion zurückrollt, einen Bericht aus den fehlerhaften Daten erstellt und diesen an die ausführende Anwendung zurückliefert. Vielleicht ist es sogar möglich, die aufgetretenen Fehler selbstständig zu korrigieren. Sehr wichtig ist dabei, dass diese Herangehensweise die `insert`-Anweisung als *eine einzige Anweisung* erhält und die Datenbank nicht zwingt, die Daten zeilenweise in die Datenbank einzufügen: Ich werde Ihnen in Abschnitt 5.8, »Workshop: Einfluss der Programmierung«, zeigen, wie riesig der Performance-Gewinn ist, wenn die Datenbank mit Datenmengen arbeiten kann und nicht gezwungen wird, jede Zeile einzeln zu bearbeiten. Die `log errors`-Klausel zeigt, dass immer mehr Aufgaben auf reines SQL verlagert werden können, gerade auch aus der Motivation heraus, diese Art der Mengendatenverarbeitung möglichst oft zu nutzen.

4.7.4 Fazit

Das waren drei kleine Beispiele aus einer Überfülle an Erweiterungen, die Oracle dem SQL-Standard hinzufügt und die zum Teil bereits in neuere Versionen des Standards eingeflossen sind. Natürlich unterstützen nicht alle Datenbanken diese Art von Abfrage. Daher müssen Sie in einem generischen Ansatz solche Erweiterungen ignorieren. Doch: Ist das sinnvoll? Kann auf Dauer eine Funktionalität ignoriert werden, nur weil eine der unterstützten Datenbanken keine entsprechende Funktionalität anbietet? Dies kommt einer Strategie gleich, die sich immer am schwächsten Glied der Kette orientiert und damit auch immer die schlechtestmögliche Performance und Skalierbarkeit aufweist.

Es kommt hinzu, dass Eigenentwicklungen von Funktionen, die es in SQL (oder, wie wir später sehen werden, in PL/SQL) bereits gibt, normalerweise komplizierter umzusetzen sind als Lösungen in SQL. Sehen wir uns dazu noch einmal die hierarchische Abfrage von oben an, stellen wir fest, dass die gleiche Funktionalität mit Anwendungscode niemals auf so einfache und schnelle Art zu erzielen gewesen wäre. Zudem hat mich die »Entwicklung« dieser Abfrage etwa 3 bis 5 Minuten gekostet, und ich bin vergleichsweise sicher, dass die Funktionalität korrekt implementiert ist, denn die SQL-Funktionen sind bereits tausendfach getestet und in vielen Produktionsumgebungen im Einsatz.

Als Fazit kann man sagen: Die Erweiterungen von Oracle sind eingeführt worden, um bekannte Flaschenhälse von SQL zu erweitern und die Performance kritischer Abfragen zu verbessern. Machen Sie keinen Gebrauch von diesen Erweiterungen, verschenken Sie Performance und Skalierbarkeit in einem später nicht mehr gutzumachenden Umfang. Daher sollten alle Funktionen der Datenbank, und zwar in ihrer aktuellsten Version, auch genutzt werden.

Kapitel 7

Die Blockstruktur und Syntax von PL/SQL

Genug der Vorbereitung: Nun geht es an die Definition der Sprache PL/SQL und an die Strukturen, die Sie kennen müssen, um eigene Programme entwerfen zu können. Dieses Kapitel führt Sie zunächst in die grundlegenden Strukturen ein. Sie lernen die Blockstruktur sowie die wichtigsten Anweisungen von PL/SQL kennen.

PL/SQL ist keine Programmiersprache, sondern eine prozedurale *Erweiterung* von SQL. Daher können Sie kein erfolgreiches PL/SQL-Programm schreiben, wenn Sie sich nicht ein solides Fundament im Oracle-Dialekt von SQL erworben haben. Ich begreife dieses Buch ebenfalls als einen Wegweiser durch diese Erweiterung und setze daher SQL-Grundlagen voraus, werde Ungewöhnliches aber natürlich erklären.

Nun zu PL/SQL: Sie werden in diesem Kapitel die Strukturen und syntaktischen Besonderheiten kennenlernen, die Sie beim Schreiben von PL/SQL berücksichtigen müssen. Erfahrungsgemäß verschwinden diese Probleme sehr schnell, wenn Sie sich bemühen, die Beispiele, die wir besprechen, am Rechner nachzuvollziehen. Das Schreiben von PL/SQL ist der beste Lehrmeister der Syntax. Die harte Schule wäre dabei das Werkzeug *SQL*Plus*, und wenn Sie es etwas komfortabler lieben, nehmen Sie den *SQL Developer*.

Zunächst werde ich die Blockstruktur von PL/SQL erläutern. Anschließend kümmern wir uns um die Kontrollstrukturen, also um bedingte Anweisungen und Schleifen. Danach wenden wir uns dem zentralen Thema von PL/SQL zu: den Kollektionen. Diese Kollektionen sind es maßgeblich, in denen Daten bearbeitet und transportiert werden. Das Verständnis dieser Konzepte ist von zentraler Bedeutung für die Programmierung der Datenbank. Anschließend betrachten wir die Integration von SQL in PL/SQL und vor allem die Unterschiede zwischen PL/SQL und SQL, z. B. bezüglich der Datentypen. Ein Abschnitt zu dynamischem SQL in PL/SQL rundet das Kapitel ab, und es schließt mit einem kurzen Überblick über die Oracle-Online-Dokumentation, eine wichtige Informationsquelle zu SQL ebenso wie zu PL/SQL.

7.1 Das Grundgerüst: der PL/SQL-Block

PL/SQL-Programme können innerhalb oder außerhalb der Datenbank eingesetzt werden. Wird ein PL/SQL-Programm in einer Datei außerhalb der Datenbank programmiert, hat es keinen Namen und wird daher *anonym* genannt. Im Gegensatz dazu kann ein PL/SQL-Programm innerhalb der Datenbank in verschiedenen Ausprägungen auftreten: als *Prozedur*, *Funktion*, *Trigger* oder *Package*. Dieser Abschnitt gibt Ihnen eine Einführung in diese verschiedenen Strukturen von PL/SQL-Programmen und zeigt die wichtigsten Einsatzbereiche auf.

Ein PL/SQL-Programm ist grundsätzlich immer in einer Blockstruktur aufgebaut und wird daher auch oft einfach Block genannt. So hat es sich z. B. eingebürgert, von einem *anonymen Block* zu sprechen, wenn ein PL/SQL-Programm nicht in der Datenbank gespeichert, sondern direkt ausgeführt werden soll. Diese Blockstruktur besteht aus folgenden Bereichen:

- **Deklarationsteil**
  In diesem optionalen Teil werden Variablen und Strukturen definiert. Der Deklarationsteil wird über die Schlüsselwörter `declare` (im Fall eines anonymen Blocks oder eines Triggerblocks) oder `as` bzw. `is` (im Fall von Prozeduren, Funktionen oder Packages) eingeleitet. Die Schlüsselwörter `as` und `is` sind dabei synonym und können gleichwertig verwendet werden.
- **Ausführungsteil**
  Dieser obligatorische Teil implementiert die eigentliche Funktionalität. Er wird durch das Schlüsselwort `begin` eingeleitet. Alle Variablen, die hier verwendet werden, müssen (mit wenigen Ausnahmen) im Deklarationsteil bekannt gemacht worden sein.
- **Fehlerbehandlungsteil**
  Auch dieser Teil ist optional und wird durch das Schlüsselwort `exception` eingeleitet. Taucht im Ausführungsteil ein Fehler auf, verzweigt PL/SQL sofort in den Fehlerbehandlungsteil. Sollte dieser Teil nicht vorhanden sein, wird der Fehler an die aufrufende Umgebung propagiert, bis entweder ein Fehlerbehandlungsteil im aufrufenden Umfeld den Fehler behandelt oder das gesamte Programm mit der Fehlermeldung abbricht.
- **Ende des Blocks**
  Der Block wird durch das obligatorische Schlüsselwort `end` beendet.

Die Syntax entspricht in weiten Teilen der von SQL, so können also die gleichen Kommentarzeichen etc. genutzt werden. PL/SQL erweitert die Syntax von SQL lediglich dort, wo entsprechende Konstrukte in SQL nicht verfügbar sind. Wir sehen diese Ähnlichkeit z. B. auch an der Deklaration von Variablen, die sich lesen wie die Deklaration einer Spalte einer Tabelle in SQL.

Sehen wir uns einmal einen anonymen Block an, wie er z. B. innerhalb eines Skripts in SQL*Plus ausgeführt werden könnte. Ich erläutere zunächst nur grob, was dort zu sehen ist, werde die Details später jedoch nachliefern. Um dieses Beispiel nachzuvollziehen, können Sie den folgenden Quell-Code in SQL*Plus oder im SQL Developer im SQL-Fenster eingeben:

```
SQL> set serveroutput on
SQL> declare
  2    -- Deklarationsteil, hier werden Variablen deklariert
  3    l_end_time varchar2(25);
  4  begin
  5    -- Ausführungsteil. Hier wird gearbeitet
  6    l_end_time :=
  7      to_char(
  8        next_day(sysdate + interval '30' day, 'MON'),
  9        'DD.MM.YYYY');
 10    dbms_output.put_line('Rückgabe am ' || l_end_time);
 11  exception
 12    -- Fehlerbehandlungsteil
 13    when others then
 14      dbms_output.put_line('Fehler: ' || sqlerrm);
 15  end;
 16  /
Rückgabe am 27.02.2017
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Listing 7.1 Beispiel eines einfachen anonymen Blocks

Neben den Schlüsselwörtern `declare`, `begin`, `exception` und `end` fallen einige syntaktische Besonderheiten auf:

- Zeile `3`:
  Eine Variable wird definiert, indem ihr Name und anschließend ihr Typ angegeben werden, ähnlich wie das auch in SQL zur Deklaration der Spaltentypen einer Tabelle gemacht wird. Vereinfacht gesagt, stehen alle SQL-Datentypen plus `Boolean` für Wahrheitswerte zur Verfügung. Alle Anweisungen enden mit einem Semikolon und können sich über mehrere Zeilen erstrecken. Eine weitverbreitete Konvention stellt Variablen ein `l_` (für *local*) voran, um sie von Parametern und vor allem gleichnamigen Tabellenspalten zu unterscheiden.
- Zeile `6`:
  Der Zuweisungsoperator `:=` ist erfahrungsgemäß ein Stolperstein beim Erlernen der Sprache PL/SQL, denn das einfache Gleichheitszeichen dient lediglich zum Vergleichen von Werten (ist also lediglich ein mathematischer Operator). Bevor Sie sich, mit einem Java-Hintergrund z. B., über diese etwas altertümlich wirkende

Zuweisung lustig machen, denken Sie an den Unterschied von =, == und === in diesen Sprachen ... Ich lese den Zuweisungsoperator als »soll sein gleich«. Die Berechnung des Datums erfolgt mit herkömmlichen Funktionen aus SQL, hier also: »Die Rückgabe soll erfolgen am nächsten Montag nach heute plus 30 Tagen.« In PL/SQL wird häufig mit Datumsangaben gerechnet. Daher lohnt sich ein Blick in die Oracle-Dokumentation zu Datumsfunktionen.

- Zeile 10:
  Hier wird eine Funktion aus dem Package `dbms_output` aufgerufen. Ein Package kann, vereinfacht gesagt, Prozeduren und Funktionen unter einem Namen sammeln. Oracle liefert bereits fertige Sammlungen mit, deren Namen oft mit `dbms_` beginnen. Die Prozedur `put_line` gibt Text auf der Konsole aus, falls diese das zulässt. Damit SQL*Plus diesen Text auch wirklich ausgibt, musste vor dem Aufruf des Blocks die SQL*Plus-Anweisung `set serveroutput on` gesetzt werden. Täten wir dies nicht, würde kein Fehler auftauchen, aber auch kein Text ausgegeben. Nutzen Sie den SQL Developer, müssen Sie die Ausgabe aktivieren. Das geht so: Menü Ansicht • DBMS-Ausgabe wählen, anschließend eine Datenbankverbindung bestimmen, die im Fenster Ausgaben anzeigen soll, wie in Abbildung 7.1 gezeigt. (Das kann von der Version des SQL Developers abhängen. Die Abbildung zeigt Version 4.2.)

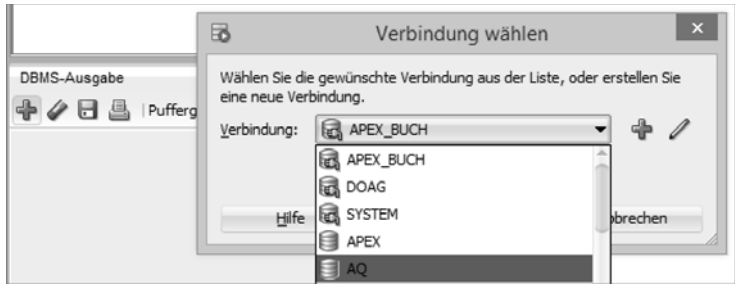

Abbildung 7.1 Server-Output in SQL Developer aktivieren

- Zeile 13:
  Im Fehlerbehandlungsteil benutzt dieses Beispiel eine bedingte Anweisung, die so nur im Fehlerteil erlaubt ist: `when ... then`. Dies erlaubt uns, zwischen verschiedenen Fehlern zu unterscheiden und entsprechend zu reagieren. Ich verwende hier `others`, um lediglich einen Standardfehler-Handler einzurichten. Diese Anweisung besagt, dass, egal, welcher Fehler auftritt, die Anweisungen hinter dieser Anweisung auszuführen sind.
- Zeile 14:
  Alles, was dieser anonyme Block als Fehlerbehandlung tut, ist, den Fehler auszugeben. Dafür verwende ich eine vordefinierte Oracle-Variable `sqlerrm` (für *SQL Error Message*), die den Text der Fehlermeldung enthält.

- Zeile 16:
  SQL*Plus wird durch die Eingabe des Schlüsselworts `declare` oder `begin` in einen speziellen PL/SQL-Modus gesetzt. Dieser Modus ist erforderlich, weil das Semikolon in PL/SQL nicht mehr in jedem Fall das Ende einer Anweisung, die sofort auszuführen ist, markiert, sondern innerhalb des Programms eine Anweisung beendet. Um den PL/SQL-Modus zu verlassen, ist es erforderlich, ein einzelnes »/« in eine Zeile einzutragen. Dies veranlasst SQL*Plus, den anonymen Block zu parsen und auszuführen. Verwenden Sie den SQL Developer, ist dieses Zeichen nicht zwingend erforderlich. Hier reicht es, den Code auszuführen (Taste F9).

  Allerdings hilft das Steuerzeichen auch im SQL Developer, denn dieses Zeichen sagt dem Editor, dass hier der PL/SQL-Block aufhört. Haben Sie mehrere Anweisungen im Fenster des SQL Developers stehen, reicht es dann, den Cursor in den gewünschten Block zu stellen. SQL Developer wählt beim Ausführen den korrekten Bereich aus, was nicht ginge, wenn das Steuerzeichen fehlte. Vergessen Sie bitte nicht, das einzelne »/« in den Umgebungen, die dies erfordern, also z. B. in SQL*Plus und natürlich in Skriptdateien, die durch SQL*Plus ausgeführt werden, auch zu verwenden. Fehlt das Zeichen dort, wird das Programm nicht korrekt geparst und bricht mit einer Fehlermeldung ab.

Typisch für anonyme PL/SQL-Blöcke ist die Verwendung in Skriptdateien, insbesondere in Kombination mit dem Aufruf der mitgelieferten Packages von Oracle. Sehen Sie sich folgendes Beispiel für eine solche Verwendung an (Sie benötigen eventuell weitergehende Rechte, um diese Anweisung ausführen zu dürfen. Dieser Aufruf wurde als Datenbankbenutzer `system` ausgeführt):

```
SQL> begin
  2    dbms_xdb.setHttpPort(8080);
  3  end;
  4  /
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Mit diesem Aufruf stellen wir den Port für den in die *XML Database* (XDB) integrierten Webserver auf Port `8080` ein. Dafür benutzen wir wiederum ein von Oracle mitgeliefertes Package mit dem Namen `dbms_xdb`, in dem sich die Prozedur `setHttpPort` befindet. Diese Prozedur erwartet einen *Parameter* für die Portnummer, die die Prozedur anschließend setzt. Aufgaben dieser Art lassen sich nur über einen PL/SQL-Aufruf durchführen. Selbst wenn Sie eine grafische Oberfläche nutzen, um diese Einstellungen vorzunehmen (dieses Beispiel könnten wir auch über das Database Control nachvollziehen), wird diese Oberfläche eine ähnliche Anweisung an die Datenbank senden.

Dieses Beispiel zeigt übrigens gleichzeitig den minimalen Aufwand, den Sie für einen anonymen Block betreiben müssen: Hier haben wir den Deklarationsteil ebenso weggelassen wie den Fehlerbehandlungsteil; es bleiben lediglich die verpflichtenden

Schlüsselwörter `begin` und `end` übrig. Ist der anonyme Block so simpel wie im obigen Beispiel und besteht nur aus einem einzigen Funktionsaufruf, kann – noch kürzer – SQL genutzt werden, um die Funktion aufzurufen:

```
SQL> call dbms_xdb.sethttpport(8080);
Aufruf wurde abgeschlossen.
```

Listing 7.2 Aufruf-Alternativen von PL/SQL-Blöcken

Der Befehl `call` ist Teil von ISO-SQL und unterstützt den Aufruf von gespeicherten Prozeduren. Alternativ, aber SQL*Plus-proprietär, wäre auch noch der SQL*Plus-Befehl `exec` oder `execute` verwendbar; ich rate Ihnen jedoch wegen der geringeren Portabilität dieser Anweisung davon ab.

7.1.1 Deklaration von Variablen

Im Beispiel oben haben Sie bereits gesehen, wie eine Variable deklariert wird. Syntaktisch funktioniert das ähnlich wie die Deklaration einer Tabellenspalte, und wir hatten bereits gesagt, dass auch alle SQL- und PL/SQL-Datentypen zur Verfügung stehen. Es gibt allerdings noch einige kleine Spielarten der Deklaration, die ich Ihnen im Folgenden zeigen möchte:

Initialisierung von Variablen

Bei der Deklaration einer Variablen kann dieser direkt auch ein Startwert zugewiesen werden. Dieser Wert wird der Variablen über den Zuweisungsoperator `:=` oder das Schlüsselwort `default` direkt bei der Deklaration zugewiesen:

```
SQL> declare
  1   l_end_time varchar2(25);
  2   l_day_amount binary_integer := 30;
  3  begin
  4    -- Ausführungsteil. Hier wird gearbeitet
  5    l_end_time :=
  6      to_char(
  7        next_day(sysdate + l_day_amount, 'MON'), 'DD.MM.YYYY');
  8    dbms_output.put_line('Rückgabe am ' || l_end_time);
  9  exception
 10    when others then
 11      dbms_output.put_line('Fehler: ' || sqlerrm);
 12  end;
 13  /
```

Listing 7.3 Deklarationsalternativen von Variablen

Der Datentyp `binary_integer` (eine synonyme Bezeichnung lautet `pls_integer`) ist eine Ganzzahl von 32 Bit Länge, hat also, grob gesagt, den Wertebereich von –2 Milliarden bis +2 Milliarden. Zudem können Variablen auch noch Constraints beinhalten sowie durch weitere Angaben in den erlaubten Wertebereichen eingeschränkt werden. Ich werde nun nicht für jede Option einen Beispiel-Code abdrucken, sondern Ihnen lediglich die verschiedenen Optionen zeigen:

```
l_day_amount binary_integer not null default 30;
l_day_amount binary_integer range 0..60;
```

Ebenso ist es möglich, eine Variable zur Konstanten zu erklären. In diesem Fall kann die Variable später nicht mehr geändert werden und bedarf daher natürlich auch eines Initialisierungswertes:

```
l_day_amount constant binary_integer default 30;
```

Listing 7.4 Weitere Deklarationsoptionen von Variablen

Dann ist bemerkenswert, dass Variablen analog zu bereits bestehenden Variablen deklariert werden können, indem ihnen der Typ der anderen Variablen zugewiesen wird:

```
l_tax_rate number;
l_full_vat_rate l_tax_rate%TYPE := 0.19;
l_reduced_vat_rate l_tax_rate%TYPE := 0.07;
```

Listing 7.5 Ableitung von Variablen-Deklarationen von anderen Variablen

Im Beispiel oben wird eine Variable deklariert. Die nachfolgenden Steuersätze spezifizieren dann auf Basis des allgemeinen Typs die konkreten Steuersätze. Durch den Verweis auf die Variable `l_tax_rate` gelten nun auch für die abgeleiteten Werte die Einschränkungen bezüglich des Bereichs. Natürlich ist auch das Attribut `type` nicht an die Großschreibung gebunden, sondern, wie PL/SQL generell, *case-insensitive*. Für Konstanten und solche Attribute hat es sich allerdings eingebürgert, die Großschreibung zu verwenden. Der Vorteil der Verwendung dieses Attributs besteht in der Verwaltung der Abhängigkeiten durch Oracle: Ändert sich der Typ der Variablen `l_tax_rate`, ändern sich die Typen der davon abgeleiteten Variablen analog mit. Etwas konkreter werde ich Ihnen die Optionen, die hier verwendet werden können, noch in Kapitel 9, »Datentypen in PL/SQL«, vorstellen, für den ersten Überblick sollen uns diese Optionen hier reichen.

7.1.2 Schachtelung von Blöcken zur Fehlerbehandlung

Anonyme Blöcke werden, wie bereits beschrieben, in administrativen Skripten eingesetzt, sie haben jedoch auch eine Funktion über diesen Einsatzbereich hinaus. Wie Sie

bereits gesehen haben, definieren Blöcke einen Rahmen, in dem Variablen gelten und innerhalb dessen eine Fehlerbehandlung durchgeführt werden kann. Zudem verfügt PL/SQL über die Fähigkeit, Blöcke ineinander verschachteln zu können. Aus der Kombination dieser drei Eigenschaften ergeben sich weitere Einsatzmöglichkeiten für anonyme Blöcke in der Programmierung. So können anonyme Blöcke genutzt werden, um innerhalb eines größeren Zusammenhangs einen Abbruch durch einen Fehler zu verhindern und normal weiterzuarbeiten, wie in diesem Pseudo-Code-Beispiel angedeutet:

```
begin
  --führe Arbeiten aus
  begin
    -- hier kommt die Anweisung, die einen Fehler auslösen
    -- könnte, wie zum Beispiel eine select-Anweisung,
    -- die keine Zeile findet
  exception
    -- hier wird dieser (erwartete) Fehler abgefangen
    when no_data_found then null; -- Fehler ignorieren
  end;
  -- fahre mit den normalen Anweisungen fort.
exception
  -- Fehlerbehandlungsteil des umgebenden Blocks
end;
```

Listing 7.6 Schachtelung von PL/SQL-Blöcken

Dieses Verfahren wird oft in Schleifenkonstruktionen angewendet, in denen viele Datensätze bearbeitet werden. Sollte innerhalb einer Schleife ein Fehler auftreten, der nicht in einem separaten Block innerhalb der Schleife abgefangen wird, hätte dies zur Folge, dass die gesamte Schleife abbräche und zum `exception`-Teil des umgebenden Blocks verzweigte. Sollte also ein erwarteter Fehler auftauchen, wird dieser gezielt behandelt. Diese Herangehensweise ist sinnvoll, denn alle (nicht erwarteten) anderen Fehler werden nach wie vor ausgelöst und nicht unterdrückt. Fraglich ist hier, ob die Übersetzung »Fehler« für eine erwartete Ausnahme angemessen ist. Vielleicht sollten wir bei einem erwarteten Auftreten einer `exception` tatsächlich von einer *Ausnahme* und ansonsten von einem *Fehler* sprechen.

7.1.3 Gültigkeitsbereich von Variablen

Durch die Schachtelung von PL/SQL-Blöcken lässt sich jedoch auch der Gültigkeitsbereich von Variablen eingrenzen, da eine Variable immer nur innerhalb des Blocks gilt. Eventuell kann durch dieses Verfahren der Code übersichtlicher gestaltet werden, weil Variablen, die lediglich in einem sehr kleinen Zusammenhang genutzt wer-

den, auch in diesem Zusammenhang deklariert werden können. Zudem unterstützt PL/SQL die *Maskierung* von Variablen, bei der die Deklaration einer Variablen im eingeschachtelten Block die Deklaration einer gleichnamigen Variablen im umfassenden Block überdeckt:

```
SQL> declare
  2    l_my_test varchar2(20 char) := 'Willi Müller';
  3  begin
  4    declare
  5      l_my_test varchar2(20 char);
  6    begin
  7      my_test := 'Alfred Peter';
  8      dbms_output.put_line('Innerer Block: ' || l_my_test);
  9    end;
 10    dbms_output.put_line('Äußerer Block: ' || l_my_test);
 11  end;
 12  /
Innerer Block: Alfred Peter
Äußerer Block: Willi Müller
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Listing 7.7 Gültigkeitsbereich und Sichtbarkeit von Variablen

PL/SQL unterstützt dieses Verfahren, ich jedoch nicht: Ich empfehle Ihnen dieses Verfahren nur, wenn Sie die wahre Intention Ihres Codes aktiv verschleiern möchten. Bedenken Sie aber, dass es meistens Sie selbst sind, die/der den Code nach kurzer Zeit schon nicht mehr verstehen wird. Zusammenfassend sollten wir wohl sagen, dass die Schachtelung von anonymen Blöcken normalerweise nur dann durchgeführt werden sollte, wenn eine gezielte Ausnahmebehandlung innerhalb eines größeren Kontextes vorgenommen wird, nicht jedoch mit Bezug auf den Gültigkeitsbereich von Variablen.

7.2 Prozeduren

Soll ein PL/SQL-Block in der Datenbank gespeichert werden, um häufiger ausgeführt werden zu können, benötigt er zunächst einmal einen Namen. Diese gespeicherten PL/SQL-Blöcke werden dann als *gespeicherte Prozedur* bezeichnet. Von diesem allgemeinen Oberbegriff leiten wir dann als Spezialfall noch die Funktion ab. Da ein Programm, das häufiger gebraucht wird, an Funktionalität gewinnt, wenn es mit unterschiedlichen Parametern genutzt werden kann, müssen wir uns ansehen, welche Optionen der Parameterübergabe PL/SQL zur Verfügung stellt. Schließlich stellt sich die Frage nach der Organisation von PL/SQL-Code, wenn sehr viel Funktionali-

tät in der Datenbank umgesetzt wird. Sehen wir uns also einmal die verschiedenen Prinzipien genauer an.

Im einfachsten Fall wird ein anonymer PL/SQL-Block als Prozedur ohne Parameter angelegt. In diesem Fall verhält sich der PL/SQL-Block genauso wie sein anonymer Vorgänger; er erhält lediglich einen Namen und kann unter diesem ausgeführt werden. Die Anweisung, um eine Prozedur zu erzeugen, ist eine `create`-Anweisung in SQL. Die Syntax sieht eine Reihe von Optionen zur Erstellung vor. Sehen wir uns die Anweisung zunächst im Überblick an:

```
create [or replace] procedure <Name der Prozedur>
[(<Parameterdeklaration)>]
as/is
<Defintion mit begin - exception - end>
```

Listing 7.8 Prinzipielle Syntax zur Erstellung einer Prozedur

Eckige Klammern zeigen optionale Schlüsselwörter an. Beachten Sie bitte, dass nun der Deklarationsteil nicht mehr mit `declare` eingeleitet wird, sondern mit `as`/`is`. Nehmen wir also unseren anonymen Block aus Abschnitt 7.1, »Das Grundgerüst: der PL/SQL-Block«, würde aus dem anonymen Block die Prozedur `print_return_date`:

```
SQL> create or replace procedure print_return_date
  2  as
  3    l_end_time varchar2(25);
  4  begin
  5    l_end_time :=
  6      to_char(next_day(
  7        trunc(sysdate) + interval '30' day,
  8       'MON'), 'DD.MM.YYYY');
  9    dbms_output.put_line('Rückgabe am ' || l_end_time);
 10  exception
 11    when others then
 12      dbms_output.put_line('Fehler: ' || sqlerrm);
 13  end print_return_date;
 14  /
Prozedur wurde erstellt.
```

Beachten Sie, dass wir den Namen der Prozedur im abschließenden `end` der Prozedur wiederholen. Dies ist syntaktisch nicht unbedingt erforderlich, erleichtert aber die Navigation im Quell-Code, da es die Übersichtlichkeit erhöht. Der Name, den wir dieser Prozedur gegeben haben, folgt den Namenskonventionen für Oracle-Datenbankobjekte:

- Er darf nur aus den Buchstaben (ohne Umlaute und sonstige Sonderzeichen) und Zahlen sowie den Sonderzeichen `$`, `_` und `#` bestehen, wobei Oracle die beiden Sonderzeichen `$` und `#` nicht empfiehlt (aber selbst verwendet ...).
- Er muss mit einem Buchstaben beginnen.
- Er darf maximal 30 Zeichen lang sein und keine Leerzeichen enthalten. Erst ab Version 12.2 sind Bezeichner mit 128 Zeichen Länge erlaubt.
- Da PL/SQL nicht case-sensitive ist, können sich zwei Prozeduren nicht durch Groß- und Kleinschreibung voneinander unterscheiden. Aus dem gleichen Grund wird in PL/SQL normalerweise auch nicht mit *CamelCase* gearbeitet, zumal viele automatische Code-Formatierer die Schreibweise ändern.

Anschließend können wir die Prozedur benutzen:

```
SQL> call print_return_date();
Rückgabe am 27.09.2017
Aufruf wurde abgeschlossen.
```

Listing 7.9 Beispiel einer einfachen Prozedur

Bevor wir fortfahren, sollten wir uns die Anweisung etwas genauer ansehen. Die `create`-Anweisung ist in unserem Beispiel durch `or replace` erweitert worden. Diese Option besagt, dass die Prozedur ersetzt werden soll, falls sie bereits besteht. Auf diese Weise erleichtern wir uns die Entwicklung einer Prozedur, denn falls Fehler auftreten, muss die Prozedur nicht zunächst gelöscht und anschließend neu angelegt werden. Zudem werden bei diesem Verfahren alle Rechte erhalten, die Sie anderen Benutzern an dieser Prozedur erteilt haben. Löschten und erstellten Sie die Prozedur neu, hätte dies zur Folge, dass alle Berechtigungen zur Ausführung dieser Prozedur ebenfalls neu eingerichtet werden müssten. Sie kennen ein solches Verfahren eventuell auch aus der Definition einer View, die ebenfalls über diese Möglichkeit verfügt. Wie immer bei Oracle: Warten Sie bitte nicht auf eine Warnmeldung oder irgendeinen Hinweis, eventuell bestehende Prozeduren werden sofort und unwiderruflich überschrieben, sobald Sie diese Anweisung ausführen!

Theoretisch dürfte der Name einer Prozedur oder Funktion sogar gegen alle oben genannten Konventionen bis auf die maximale Länge von 30 bzw. 128 Byte (denken Sie an Umlaute etc., die in Unicode zwei Byte benötigen!) verstoßen, dann nämlich, wenn Sie den Funktionsnamen in doppelte Anführungszeichen setzen, wie Sie das z. B. auch bei Spaltenaliassen machen können:

```
SQL> create or replace function "3 x schwarzer Kater"
  2    return number
  3  as
```

```
  4  begin
  5    return 3*7;
  6  end;
  7  /
Function created.
SQL> select "3 x schwarzer Kater"
  2    from dual;
3 x schwarzer Kater
-------------------
                 21
```

Listing 7.10 Ein ganz besonders schlechtes Beispiel

Und wenn wir schon einmal bei Namenskonventionen sind: Es ist prinzipiell möglich, auch Umlaute im Namen zu verwenden, auch ohne doppelte Anführungszeichen. Für beide Verfahren gilt, analog zum Jugendschutzgesetz: Nicht alles, was gesetzlich erlaubt ist, müssen die Eltern auch gestatten. Meine Empfehlung lautet daher: Machen Sie von diesen Möglichkeiten keinen Gebrauch.

Nebenbei: Wo wird diese Prozedur eigentlich gespeichert? Ich weiß aus den Kursen, die ich zum Thema gebe, dass diese Frage stets für Verwirrung sorgt. Die Antwort: Der Code ist Teil des *Data Dictionarys*, also der Metadaten der Datenbank. Daher liegt der Code in Tabellen des Benutzers `SYS` und wird ebenso behandelt wie z. B. die Beschreibung einer Tabelle. Beide Datenbankobjekte gehören dem Eigentümer, in unserem Fall also dem Benutzer `SCOTT`. Wir können uns diese Einträge in den Tabellen des Data Dictionarys über eine View anzeigen lassen. Diese View heißt `USER_SOURCE` und liefert für unsere Prozedur folgende Ausgabe:

```
SQL> select line, text
  2    from user_source
  3   where name = 'PRINT_RETURN_DATE';

LINE TEXT
----- --------------------------------------------------
    1 procedure print_return_date
    2 as
    3   l_end_time varchar2(25);
    4 begin
    5   l_end_time :=
    6     to_char(
    7       next_day(sysdate + interval '30' day,
    8                'MON'),
    9       'DD.MM.YYYY');
   10   dbms_output.put_line('Rückgabe am ' || l_end_time);
```



```
   11 exception
   12   when others then
   13     dbms_output.put_line('Fehler: ' || sqlerrm);
   14 end print_return_date;
14 Zeilen ausgewählt.
```

Listing 7.11 Ausgabe der Prozedur aus der View »USER_SOURCE«

Diese View ist eine sehr ergiebige Quelle für alle Arten von Analysen zum Code, wie Sie sich sicher vorstellen können.

7.2.1 Prozeduren mit Parametern

Um nun die Einsatzbereiche dieser kleinen Funktion zu erweitern, soll es möglich sein, eine Anzahl von Tagen sowie ein Datum zu übermitteln, aus denen das Rückgabedatum errechnet wird. Dazu müssen wir zwei Parameter vereinbaren, die beim Aufruf der Prozedur mit angegeben werden. PL/SQL unterscheidet drei verschiedene Parameterarten:

- **Eingabeparameter**
  Diese Parameter sind der Standardtyp. Sie bedeuten, dass in ihnen Werte an die Prozedur übergeben und dort verarbeitet werden. Beachten Sie aber, dass Eingabeparameter in der Prozedur nicht geändert werden können, sie können lediglich gelesen werden!
- **Ausgabeparameter**
  Dieser Parametertyp kann keine Werte entgegennehmen, sondern liefert berechnete Werte an die aufrufende Umgebung zurück.
- **Ein-/Ausgabeparameter**
  Dieser Parametertyp ist eine Kombination aus den beiden vorherigen. Er erwartet eine Eingabe über diesen Parameter, verändert diese und liefert sie auf gleichem Wege zurück.

Parameter werden nach dem Namen der Prozedur in einer Klammer als kommaseparierte Liste angegeben, ähnlich wie Spalten einer Tabelle deklariert werden. Analog zu diesem Verfahren wird zunächst der Name des Parameters und dann der Datentyp angegeben. Allerdings ist die Syntax um die Richtungsangabe des Parameters erweitert. Hier stehen die Schlüsselwörter `in`, `out` oder `in out` zur Verfügung. Prozeduren unterstützen sowohl mehrere Ein-, Aus- und Ein-/Ausgabeparameter in der Prozedurdeklaration. Damit kann eine Prozedur nicht nur mehrere Werte aufnehmen, sondern auch zurückliefern. Lassen Sie das Schlüsselwort für die Richtungsangabe weg, behandelt PL/SQL diese Parameter als Eingabeparameter. Für die Parameter einer Prozedur gilt, dass alle SQL-Datentypen verwendet werden dürfen. Dies gilt

auch für komplexe Typen wie `XMLType` oder ähnlich. Natürlich dürfen Sie auch die in PL/SQL zusätzlich existierenden Datentypen `boolean` oder strukturierte Datentypen verwenden.

Ein-/Ausgabeparameter werden bei der Übergabe an die Prozedur kopiert, die Kopie wird in der Prozedur bearbeitet und nach Abschluss der Berechnung in die Parametervariable umkopiert. Dieses Verhalten stellt sicher, dass der Parameter nach einem Fehler der Prozedur unverändert bestehen bleibt. Veränderte PL/SQL die Parametervariable direkt und träte dann im weiteren Verlauf der Prozedur ein Fehler auf, wäre der Zustand der Parametervariablen so, wie er zum Zeitpunkt des Fehlers war. Damit wäre die Forderung aus dem ACID-Konzept verletzt, das ja vorsieht, dass bei einem Fehler die Daten unverändert wiederhergestellt werden. Dies ist durch das Arbeiten an der Kopie ausgeschlossen: Im Fehlerfall wird die unveränderte Kopie der Parametervariablen zurückgegeben (zumindest wenn der auftretende Fehler in der Prozedur selbst behandelt wurde). Der Nachteil dieser Vorgehensweise ist allerdings, dass der doppelte Speicher für diesen Parameter gebraucht wird. Haben Sie also eine Situation, in der Sie damit leben können, dass die Parametervariable direkt geändert wird, und wäre der zusätzliche Speicherverbrauch nicht tolerabel, können Sie die Parameterdeklaration für den Modus `in out` (und überraschenderweise auch `out`) dadurch ergänzen, dass Sie die Klausel `nocopy` hinzufügen. Dies kann insbesondere bei sehr großen LOB-Strukturen (auch `XmlType`-Instanzen) Sinn ergeben:

```
create my_proc(p_huge_text in out nocopy clob)  is ...
```

Listing 7.12 Verwendung der Direktive »nocopy«

Nun wird direkt auf dem Eingabeparameter gearbeitet und damit der Speicherverbrauch reduziert. Diese Option gilt naturgemäß nur für den Modus `in out`, nicht für den Modus `in`, denn in diesem Fall ist der Eingabeparameter nicht änderbar.

Nun definieren wir die Prozedur mit Parametern. Beachten Sie bitte auch, wie die Parameter innerhalb der Prozedur die hartcodierten Datums- bzw. Tagesangaben ersetzen:

```
SQL> create or replace
  2  procedure print_return_date
  3    (p_start_date in date,
  4     p_day_amount in number)
  5  as
  6    l_end_time varchar2(25);
  7  begin
  8    l_end_time :=
  9      to_char(
 10        next_day(
```



```
 11          p_start_date + p_day_amount, 'MON'), 'DD.MM.YYYY');
 12    dbms_output.put_line('Rückgabe am ' || l_end_time);
 13  exception
 14    when others then
 15      dbms_output.put_line('Fehler: ' || sqlerrm);
 16  end print_return_date;
 17  /
Prozedur wurde erstellt.
SQL> call print_return_date(sysdate, 24);
Rückgabe am 20.09.2017
Aufruf wurde abgeschlossen.
```

Listing 7.13 Beispiel für eine einfache Prozedur mit Eingabeparametern

Sie sehen, wie sich der Aufruf der Prozedur verändert. Ich habe im Code-Beispiel oben eine Konvention verwendet, die relativ üblich ist: Parameter einer Prozedur oder Funktion erhalten häufig ein Präfix `p_`, während die Variablen, die innerhalb des PL/SQL-Blocks vereinbart werden, das Präfix `l_` erhalten. Alternativ wird von einigen PL/SQL-Entwicklern auch vorgeschlagen, die Präfixe `i_`, `o_` und `io_` zu verwenden und damit die Richtung der Parameter anzuzeigen. Ich persönlich finde das aber eher unübersichtlicher.

Merke

Erfahrene Programmierer werden sich gelangweilt abdrehen und sagen: Schon wieder einer, der mir vorschreiben möchte, wie ich Variablen zu benennen habe. Doch gibt es im Umfeld von PL/SQL ein zusätzliches Argument für eine solche Konvention, die Sie bedenken sollten, bevor Sie diesen Ratschlag ablehnen: Parameter werden sehr häufig nach Spalten einer Tabelle benannt. Die Namen dieser Parameter sollten keinesfalls (syntaktisch dürfen sie es, aber dies kann zu fehlerhaftem Code führen) genauso heißen wie die Spalten. Mit dieser Konvention sind Sie diesbezüglich aus dem Schneider und profitieren zudem noch von der besseren Dokumentation Ihres Codes.

Wie können wir uns einen Ausgabeparameter vorstellen? Vielleicht erweitern wir unsere Prozedur dadurch, dass wir das Rückgabedatum nicht direkt ausgeben, sondern über einen Ausgabeparameter an die aufrufende Umgebung zurückgeben. Doch löschen wir zunächst unsere alte Prozedur, die brauchen wir nun nicht mehr (ich werde den Namen der Prozedur ändern, daher funktioniert das `replace` hier nicht mehr):

```
SQL> drop procedure print_return_date;
Prozedur wurde gelöscht.
```

Anschließend erstellen wir die neue Variante:

```
SQL> create or replace procedure get_return_date
  2    (p_start_date in date,
  3     p_day_amount in number,
  4     p_return_date out date)
  5  as
  6  begin
  7    p_return_date :=
  8      next_day(p_start_date + p_day_amount, 'MON');
  9  end get_return_date;
 10  /
Prozedur wurde erstellt.
```

Listing 7.14 Erweiterung der Funktion um einen Ausgabeparameter

Ich habe gleich eine ganze Reihe von Änderungen durchgeführt. Zunächst habe ich die Prozedur umbenannt: `print` ist nicht mehr, was diese Prozedur tut, sondern `get`. Daher heißt die Prozedur nun `get_return_date`. Zudem ist ein Rückgabeparameter vereinbart worden, der das Datum der Rückgabe *als Datum* zurückgibt. Warum nicht als fertig umgewandelte Zeichenkette? Zunächst einmal erweitert es den Einsatzbereich der Prozedur, wenn ein Datum zurückgeliefert wird, weil mit dem Rückgabewert noch sinnvoll gerechnet, sortiert etc. werden kann. Zudem sollte die Formatierung einer Variablen das Allerletzte sein, was Sie mit einer Variablen tun. Die Formatierung ist häufig kulturspezifisch und nicht selten von den Einstellungen Ihrer Datenbank oder der Anwendung oder auch der aktuellen Session abhängig. Diese Funktionalität sollten Sie nicht ohne Not übersteuern. Es entfallen in unserer Prozedur demnach alle Umwandlungen in eine Zeichenkette sowie die Logik zum Ausdruck.

Als Letztes habe ich in diesem Beispiel die Fehlerbehandlung komplett herausgenommen. Das ist zwar nicht so schön, soll aber im Moment einmal so hingenommen werden, weil ich zu einer sinnvollen Fehlerbehandlung noch etwas mehr Vorwissen aufbauen muss. Leben wir also für dieses Beispiel damit.

Um diese Prozedur nun nutzen zu können, müssen wir jetzt außerhalb dieser Prozedur eine Variable definieren, die den Ausgabeparameter der Prozedur aufnehmen kann. Anschließend möchten wir das Datum schlicht ausdrucken. Also erstellen wir einen anonymen Block, der diese Prozedur aufruft und das Ergebnis ausgibt. Wenn Sie den Lerneffekt erhöhen möchten, schreiben Sie doch die folgende Prozedur nicht einfach ab, sondern erstellen Sie sie selbst:

```
SQL> declare
  2    p_return_date date;
  3  begin
```



```
  4    get_return_date(sysdate, 24, p_return_date);
  5    dbms_output.put_line('Rückgabe am ' ||
  6      to_char(p_return_date, 'DD.MM.YYYY'));
  7  end;
  8  /
  9  /
Rückgabe am 20.09.2017
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Listing 7.15 Beispiel einer einfachen Prozedur mit Ausgabeparametern

Falls Sie den PL/SQL-Code selbst entwickelt haben, hatten Sie den Reflex, anstelle von

```
  4    get_return_date(sysdate, 24, p_return_date);
```

so etwas wie

```
  4    l_return_date := get_return_date(sysdate, 24);
```

zu schreiben? Dann geht es Ihnen wie mir, dann hätten Sie eine Funktion verwenden wollen, doch warten wir noch einen Moment, wir kommen auch noch zu dieser Variante.

Etwas unschön ist nun, dass ein Parameter für das Startdatum und ein weiterer Parameter für das Rückgabedatum erforderlich sind. Das Startdatum wird eventuell nach der Umwandlung gar nicht mehr benötigt und könnte doch eigentlich auch auf dem gleichen Weg zurückgegeben werden, wie es hineinkommt, oder? Dies wäre ein Anwendungsfall für einen Ein-/Ausgabeparameter. Sehen wir uns diese Variante einfach einmal an:

```
SQL> create or replace procedure get_return_date (
  2    p_process_date in out date,
  3    p_day_amount in number)
  4  as
  5  begin
  6    p_process_date := next_day(
  7      p_process_date + p_day_amount, 'MON');
  8  end get_return_date;
  9  /
Prozedur wurde erstellt.
```

Listing 7.16 Erweiterung um einen Ein-/Ausgabeparameter

Da ich nun nicht definitiv zwischen Start- und Rückgabedatum unterscheiden kann, habe ich den Parameter `p_process_date` genannt. Interessant ist nun, wie dem Parameter der neue Wert zugewiesen wird, indem der Parameter neu berechnet wird.

Vielleicht kommt Ihnen das etwas seltsam vor, es entspricht inhaltlich jedoch dem in allen Programmiersprachen üblichen Verfahren.

```
i := i + 1;
```

Leider steht in PL/SQL keine Entsprechung für die sehr praktischen Operatoren anderer Programmiersprachen wie etwa `i += 1;` aus Java (oder JavaScript etc.) zur Verfügung, die die gleiche Aufgabe wie unser Beispiel-Code erledigen. Für den Aufruf müssen wir nun ebenfalls etwas ändern: Unserer Variablen muss bereits vor dem Aufruf unserer Prozedur ein Startwert zugeordnet werden, damit dieser der Prozedur korrekt übergeben werden kann:

```
SQL> declare
  2    l_process_date date;
  3  begin
  4    l_process_date := sysdate;
  5    get_return_date(l_process_date, 24);
  6    dbms_output.put_line('Rückgabe am ' ||
  7      to_char(l_process_date, 'DD.MM.YYYY'));
  8  end;
  9  /
Rückgabe am 20.09.2017
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Listing 7.17 Aufruf einer Prozedur mit Ein-/Ausgabeparametern

Benannte Prozeduren können nicht nur »für sich« eingesetzt werden, sondern auch im Zusammenhang einer größeren Prozedur als Hilfsprozedur im Deklarationsteil der umgebenden Prozedur verwendet werden. Dabei nutzen wir die Fähigkeit von PL/SQL, Blöcke ineinander schachteln zu können, und kombinieren dies mit der Benennung eines Blocks. Eine solche geschachtelte Prozedur ist für den umgebenden Block dann unter diesem Namen verfügbar, für die Umgebung außerhalb des aufrufenden Blocks aber nicht sichtbar. Diese Prozedur ist also sozusagen privat. Bevor Sie sich eine solche Programmierweise aber angewöhnen, sage ich gleich, dass Sie dafür einen leistungsfähigeren Mechanismus kennenlernen werden, nämlich das Package. In der Praxis sollte das folgende Beispiel relativ selten auftauchen:

```
SQL> create or replace procedure print_return_date (
  2    p_start_date in date,
  3    p_day_amount in number)
  4  as
  5    l_internal_date date;
  6    -- Hilfsprozedur
  7    procedure get_return_date
  8      (p_process_date in out date,
  9       p_day_amount in number)
 10    as
 11    begin
 12      p_process_date :=
 13        next_day(p_process_date + p_day_amount, 'MON');
 14    end get_return_date;
 15  begin
 16    l_internal_date := p_start_date;
 17    get_return_date(l_internal_date, p_day_amount);
 18    dbms_output.put_line(
 19      to_char(l_internal_date, 'DD.MM.YYYY'));
 20  end print_return_date;
 21  /
Prozedur wurde erstellt.
SQL> call print_return_date(sysdate, 24);
20.09.2017
Aufruf wurde abgeschlossen.
```

Listing 7.18 Einfache Prozedur mit geschachtelter Hilfsprozedur

Hier gibt es eine Besonderheit, die Sie beachten sollten: Die umgebende Prozedur `print_return_date` vereinbart einen Eingabeparameter `p_start_date`. Im Gegensatz dazu vereinbart die eingelagerte Hilfsprozedur `get_return_date` einen Ein-/Ausgabeparameter. Wir können nun nicht einfach das Startdatum als Parameter an die Hilfsprozedur weitergeben, weil dieser Parameter nicht verändert werden kann (er ist ein einfacher Eingabeparameter). Aus diesem Grund wird der Eingabeparameter `p_start_date` in Zeile `16` auf eine lokale Datumsvariable `l_internal_date` umkopiert, mit der dann die Berechnung des Rückgabedatums durchgeführt wird. Beachten Sie also bitte, dass Sie Eingabeparameter innerhalb der Prozedur nicht verändern können. Sollten Sie das einmal vergessen, ist das im Übrigen auch nicht schlimm: Der Compiler wird Sie zuverlässig daran erinnern ...

7.2.2 Formen der Parameterzuweisung

In den obigen Beispielen hatten wir immer die Parameter so an die Prozedur übergeben, wie das in Programmiersprachen am gängigsten ist: positionell. Damit meine ich, dass die Parameter in der gleichen Reihenfolge übergeben wurden, wie sie in der Prozedur definiert sind. Dazu gibt es allerdings noch einige Variationen, und zwar sowohl zur Definition von Parametern als auch zur Übergabe an die Prozedur. Sehen wir uns diese Variationen einmal etwas genauer an.

Zum einen ist es möglich, einen Parameter über seinen Namen anzusprechen und ihm gezielt einen Wert zuzuweisen. Der Vorteil dieser Methode ist, dass die Reihenfolge der Parameter nicht bekannt sein muss. Der Nachteil besteht darin, dass der genaue Name des Parameters bekannt sein muss. Wenn ein Parameter über seinen Namen zugewiesen wird, wird folgende Notation verwendet:

```
SQL> call print_return_date(
  2    p_day_amount => 24,
  3    p_start_date => sysdate);
20.07.2009
Aufruf wurde abgeschlossen.
```

Listing 7.19 Explizite Parameterübergabe

In diesem Beispiel habe ich bewusst die Reihenfolge der Parameter verändert. Hier lernen wir also einen neuen Zuweisungsoperator kennen, nämlich `=>`. Warum nicht auch hier der »übliche« Zuweisungsoperator `:=` verwendet wurde, hat wohl den Grund, dass die Schreibweise mit dem Zuweisungsoperator ebenfalls existiert (und zwar bei optionalen Parametern, die wir im nächsten Abschnitt besprechen) und man daher Konfusion vermeiden wollte. Allerdings ist diese Art der Zuweisung über den Zuweisungsoperator `=>` nicht deshalb eingeführt worden, um die Reihenfolge der Parameter zu übergehen, sondern, um im Zusammenhang mit *optionalen Parametern* nur die Parameter zu belegen, denen Sie einen vom Standard abweichenden Wert zuweisen möchten. Zu optionalen Parametern erfahren Sie mehr im nächsten Abschnitt.

Die beiden Arten der Zuweisung können, wenn Sie unbedingt möchten, auch gleichzeitig genutzt werden. Es ist allerdings guter Stil, sich für ein Verfahren zu entscheiden und dies für eine Prozedur durchzuhalten. Im Regelfall ist die positionelle Zuordnung kürzer zu schreiben, es sei denn, in einer Prozedur mit vielen optionalen Parametern werden nur wenige mit abweichenden Werten belegt. Die explizite Zuordnung über den Parameternamen ist hingegen meistens klarer, weil sie das Wissen über die Parameter und deren Bedeutung besser ausdrückt.

7.2.3 Optionale Parameter

Ein Parameter kann mit einem Vorgabewert belegt werden und wird dadurch optional. Optional meint in diesem Zusammenhang, dass er beim Aufruf nicht verpflichtend angegeben werden muss. Wird er weggelassen, wird der Vorgabewert eingefügt. Die Zuweisung eines Vorgabewertes zu einem Parameter wird hier wieder über den bereits bekannten Zuweisungsoperator `:=` oder aber über das Schlüsselwort `default` durchgeführt. Formen wir also unsere Prozedur so um, dass sie optionale Parameter enthält:


```
SQL> create or replace procedure print_return_date (
  2    p_start_date in date := sysdate,
  3    p_day_amount in number := 24)
  4  as
     ...
 19  end print_return_date;
 20  /
Prozedur wurde erstellt.
```

Listing 7.20 Umformung der Prozedur mit optionalen Parametern

Wird eine Prozedur auf diese Weise definiert, kann sie auf verschiedene Weise aufgerufen werden. Hier sehen Sie eine Übersicht über einige verschiedene Optionen zum Aufruf (jeweils ohne `call` ...):

```
print_return_date(to_date('24.01.2017', 'DD.MM.YYYY'), 30);
print_return_date(date '2017-02-24'));
print_return_date(p_day_amount => 30);
print_return_date(sysdate, p_day_amount => 30);
print_return_date();
```

Listing 7.21 Optionale Parameter und ihre Aufrufvarianten

Bitte nehmen Sie sich einen Moment Zeit, um sich die verschiedenen Varianten zu erklären. Alle Varianten funktionieren, allerdings nicht alle mit dem SQL-Befehl `call`. Der Grund: `call` ist in ISO-SQL definiert, und dort gibt es keine explizite Übergabe eines Parameters mit dem Zuweisungsoperator `=>`. Diese Aufrufvariante ist also an PL/SQL gebunden.

Merke

Um die Verwirrung komplett zu machen: Sie können ab Version 11 der Datenbank die Aufrufvariante mit Zuweisungsoperator auch in SQL verwenden, denn ab dieser Version ist dies dort erlaubt, aber auch in dieser Version funktioniert ein solcher Aufruf im Umfeld von `call` nicht. Der Grund: Ginge dies, wäre die Funktion bei Oracle anders implementiert als im ISO-Standard. Da andererseits diese Funktion lediglich aus Gründen der Kompatibilität zum ISO-Standard in Oracle implementiert wurde, wäre durch die Erweiterung genau dieses Ziel wieder verfehlt. Es ist halt ein wenig kompliziert ...

[!]

Fehlt ein Parameter, wird für ihn der Vorgabewert angenommen. Interessant wäre, zu prüfen, ob folgender Aufruf gelingt:

```
print_return_date(30);
```

Das ist deshalb interessant, weil hier positionell eigentlich der erste Parameter mit einem Wert belegt wird. Dieser Parameter ist allerdings vom Typ `date`, was mit der Zahl `30` nicht belegt werden kann. Würde PL/SQL dies erkennen und versuchte, für den ersten Parameter den Vorgabewert zu nehmen und den zweiten Wert positionell mit der Zahl `30` zu belegen, könnte der Aufruf aber dennoch erfolgreich sein. Sehen wir uns an, was passiert:

```
SQL> call print_return_date(30);
call print_return_date(30)
     *
FEHLER in Zeile 1:
ORA-06553: PLS-306: Falsche Anzahl oder Typen von Argumenten
           in Aufruf von 'PRINT_RETURN_DATE'
```

Listing 7.22 Grenzen der Parameterzuordnung

Könnte, hätte, wollte: Zum Glück wird hier ein Fehler ausgelöst. Wenn der Aufruf einer Prozedur vom Abwägen irgendwelcher Alternativen abhinge, wäre das Ergebnis bei komplexeren Verhältnissen kaum noch vorherzusagen. Daher bitte ich Sie, bei Ihrer Programmierung folgendes Vorgehen einzuhalten:

- Eine Prozedur wird entweder komplett positionell oder komplett explizit aufgerufen, nicht gemischt.
- Bei optionalen Parametern sollten die Parameter explizit zugewiesen werden, damit klar wird, was genau passiert.
- Im Regelfall werden Prozedurparameter positionell übergeben, wenn ohnehin alle Parameter angegeben werden und die Prozedur ansonsten zu kompliziert zu lesen wäre. Abweichend davon werden komplexere Prozeduren mit vielen Parametern oft auch explizit aufgerufen, wenn dadurch die Arbeitsweise der Prozedur klarer dokumentiert wird.

Nachfolgend also eine Liste der möglichen Aufrufformen, die korrekt *und* verständlich sind:

```
-- positionell, weil alle Parameter belegt sind:
print_return_date(date '2017-01-24', 30);
-- explizit, weil nur ein Parameter verwendet wurde:
print_return_date(
  p_start_date => to_date('24.01.2017', 'DD.MM.YYYY'));
print_return_date(p_day_amount => 30);
-- Der Aufruf der Prozedur kann auch Vorgabewerte beinhalten:
print_return_date(sysdate, 24);
```

Listing 7.23 Empfohlene Aufrufvarianten für Prozeduren mit Parametern

Noch ein Wort zur dritten Strichaufzählung der Liste der Empfehlungen. Im folgenden Beispiel habe ich die Prozedur positionell aufgerufen. Ist diese Schreibweise klar genug? Dazu gibt es normalerweise keine einfache und stets gültige Empfehlung. Es gibt eine ganze Menge »trivialer« und häufig verwendeter Funktionen, bei denen der positionelle Aufruf vollkommen ausreicht, weil jeder unmittelbar versteht, was passiert:

```
my_text := replace(my_text, 'Willi', 'Peter');
```

In diesem Beispiel wäre ein expliziter Aufruf nicht zielführend (wie ich auch daran erkenne, dass ich nun zunächst einmal die Namen der Parameter nachschlagen muss, ich kannte sie also selbst nicht, was ein Indiz dafür ist, dass ich diese Funktion immer positionell aufrufe):

```
my_text := replace(
             srcstr => my_text,
             oldsub => 'Willi',
             newsub => 'Peter');
```

Listing 7.24 Zwei Varianten des Aufrufs einer allgemein bekannten Funktion

Andersherum ist der Aufruf mit benannten Parametern in vielen Umgebungen unerlässlich, weil eine Aneinanderreihung von sieben Parametern wirklich nichts über den Sinn der Prozedur und ihrer Parameter aussagt:

```
dbms_network_acl_admin.add_privilege(
  'my_acl', 'SCOTT', true, 'connect', null, systimestamp, null);
```

Verglichen damit schafft der explizite Aufruf mehr Klarheit (wenngleich man immer noch wissen muss, worum es hier eigentlich geht, das stimmt natürlich):

```
dbms_network_acl_admin.add_privilege(
  acl => 'my_acl',
  principal => 'SCOTT',
  is_grant => true,
  privilege => 'connect',
  position => null,
  start_date => systimestamp,
  end_date => null);
```

Listing 7.25 Zwei Varianten des Aufrufs bei einer komplexen Prozedur

Wofür Sie sich entscheiden, sollte erst in zweiter Linie von der Tipparbeit abhängen. Die Klarheit Ihres Codes steht immer im Mittelpunkt. Wenn Sie mir diesen etwas nervigen Hinweis gestatten: Auch ein Code, der »nur mal eben schnell« als Prototyp ent-

worfen wurde, sollte dieses Kriterium erfüllen, denn erfahrungsgemäß schaffen es insbesondere diese Prototypen später auch unverändert in die Produktion und machen den Nachfolgern (und Ihnen selbst) das Leben schwer.

Dass Oracle sich nicht an meine Empfehlung hält, Parameter mit einem Präfix zu kennzeichnen, ist unverzeihlich, aber wohl auch unveränderlich ... Allerdings: Oracle kennt durchaus Prozeduren, die solche Präfixe enthalten. Doch ist Oracle natürlich noch einmal in einer anderen Situation als Sie, wenn Sie eine Anwendung programmieren: Oracle kann einmal benannte Parameter nicht mehr ändern, weil dadurch weltweit Code nicht mehr korrekt funktionieren würde. Das ist in Ihrem Code normalerweise anders, nutzen Sie also die Chance, Ihre Schnittstelle zu Ihrem Code zu vereinheitlichen.

7.2.4 Beliebig viele Parameter an eine Methode übergeben

Ich möchte gern noch eine weitere Spielart der Parameterübergabe mit Ihnen besprechen, auch wenn ich hierfür ein wenig vorgreifen muss. Es geht um die Frage, ob es möglich ist, beliebig viele Parameter an eine Methode zu übergeben. Die Antwort ist einfach: Nein. Und auch wieder nicht, denn es gibt einen Ausweg. Beliebig viele Parameter können in vielen Programmiersprachen übergeben werden, doch eint die allermeisten Programmiersprachen die Eigenheit, dass diese Parameter vom gleichen Datentyp sein müssen, z. B. eine Liste von Zeichenketten. Und genau so etwas können wir in PL/SQL simulieren, allerdings mit einem kleinen Umweg. Wir können in SQL einen Datentyp definieren, der es uns erlaubt, beliebig viele Zeichenketten in einer Tabelle zusammenzufassen. Dieser Typ ist eine *Nested Table*, die wir uns in Kapitel 9, »Datentypen in PL/SQL«, und dann noch genauer in Kapitel 17, »Objektorientierung«, ansehen werden. In ihrer einfachsten Form wird eine Nested Table als Liste von Zeichenketten in SQL auf folgende Weise angelegt:

```
create or replace type char_table as table of varchar2(4000);
/
```

Listing 7.26 Anlage einer Nested Table in SQL

Beachten Sie bitte den abschließenden Schrägstrich, da sich SQL*Plus bei der Anlage eines Typs im PL/SQL-Modus befindet. Ist der Typ definiert, kann er in SQL auf folgende Weise verwendet werden:

```
select char_table('A', 'B', 'C')
  from dual;
```

Listing 7.27 Verwendung einer Nested Table

In SQL und auch in PL/SQL können wir nun mit diesem Typ arbeiten. Da es möglich ist, beliebige Typen als Parameter einer Funktion zu definieren, können wir auch eine solche Nested Table als Parameter übergeben, und wir haben einen Ersatz für beliebig viele Zeichenkettenparameter. Wichtig wäre nun noch, wie die einzelnen Bestandteile dieser Nested Table in PL/SQL verwendet werden können, doch ist dies über eine einfache Schleife ganz einfach:

```
create or replace procedure test_func(
  (p_char_list in char_table)
as
  l_result varchar2(32767);
begin
  for i in p_char_list.first .. p_char_list.last loop
    l_result := l_result || p_char_list(i);
  end loop;
  dbms_output.put_line(l_result);
end;
/
```

Listing 7.28 Beispiel für eine Prozedur mit beliebig vielen Parametern

Diese Schleifen werden wir im nächsten Kapitel ausführlicher besprechen. Mir lag hier daran, Ihnen ein kurzes Beispiel zu geben, das Sie später auch wiederfinden werden, um Ihnen die Verwendung zu zeigen. Im weiteren Verlauf des Buches werden wir diese Strategie noch häufiger verwenden.

7.3 Funktionen

Bei der Diskussion von Prozeduren hatten wir bereits Ein-/Ausgabeparameter kennengelernt. Der Vorteil, den die allgemeine Prozedur bietet, ist, dass sowohl mehrere Ein- und Aus- als auch Ein-/Ausgabeparameter verwendet werden können. Der Nachteil der Verwendung von Aus- oder Ein-/Ausgabeparametern besteht darin, dass eine Variable vorhanden sein muss, um die Ergebnisse der Prozedur aufnehmen zu können. Oft ist dies zu umständlich und in einem Fall sogar unmöglich: wenn wir Prozeduren verwenden wollen, um SQL in seinem Funktionsumfang zu erweitern. In SQL können keine Variablen deklariert werden; daher können Prozeduren auch keine Werte an SQL zurückliefern. Um den Funktionsumfang von SQL zu erweitern, benötigen wir einen anderen Mechanismus, um die Ergebnisse der Prozedur zurückzugeben. Dieser Mechanismus muss implizit, ohne Variable, auskommen. Eine Prozedur, die auf diese Weise Werte zurückliefert, nennen wir eine *Funktion*.

Grundsätzlich ist der Aufbau der Funktion der Prozedur sehr ähnlich, nur dass bei ihr ein Typ vereinbart werden muss, den die Funktion zurückliefert. Die Funktion kann (zumindest über den Rückgabetyp) lediglich einen Wert zurückliefern. Sie werden später allerdings sehen, dass diese Typen sehr umfangreich und mächtig sein können. Für den Moment nutzen wir die Funktion allerdings nur im Standgas, und zwar, um unsere fantastische »Rückgabedatumsberechnungsprozedur« fit für SQL zu machen.

Sehen wir uns an, auf welche Weise Funktionen definiert werden:

```
SQL> create or replace function get_return_date (
  2    p_start_date in date := sysdate,
  3    p_day_amount in number := 24)
  4    return date
  5  as
  6  begin
  7    return next_day(p_start_date + p_day_amount, 'MON');
  8  end get_return_date;
  9  /
Funktion wurde erstellt.
```

Listing 7.29 Einfache Funktion

Wir stellen fest, dass zunächst einmal das Schlüsselwort `procedure` durch `function` ersetzt wurde. Das ist noch nicht wirklich überraschend. Außerdem wird nach der Parameterdeklaration noch in Zeile `4` die Klausel `return <Datentyp>` eingefügt. In unserem Fall möchten wir, dass die Funktion einen Datumstyp zurückgibt. Die eigentliche Implementierung muss nun nichts weiter tun, als die Datumsberechnung durchzuführen und das Ergebnis über die Anweisung `return` (in Zeile `7`) an die aufrufende Umgebung zurückzugeben. Nun können wir die Funktion aus SQL heraus aufrufen:

```
SQL> select get_return_date as ergebnis
  2    from dual;
ERGEBNIS
----------
20.09.2017
```

Ansonsten sehen wir bei der Erstellung unserer Funktion viele Bekannte wieder. Die Funktion umfasst wiederum beliebig viele (nun ja) Eingabeparameter, die mittels Vorgabewerten optional gemacht werden können. Natürlich stehen uns zum Aufruf alle Varianten zur Verfügung, die wir bereits oben gesehen haben. Interessant wäre allerdings, einmal zu versuchen, eine Funktion mit Ein-/Ausgabeparametern zu erstellen. Ist das möglich? Machen wir einen Versuch:


```
SQL> create or replace function get_return_date (
  2    p_start_date in out date,
  3    p_day_amount in number := 24)
  4    return date
  5  as
  6  begin
  7    p_start_date :=
  8      next_day(p_start_date + p_day_amount, 'MON');
  9    return p_start_date;
 10  end;
 11  /
Funktion wurde erstellt.
```

Listing 7.30 Einfache Funktion mit Ausgabeparameter

Was soll *das* nun? Tatsächlich scheint dieser Weg zu funktionieren. Oracle warnt in der Dokumentation allerdings vor diesem Weg und empfiehlt, so etwas nicht zu tun, weil Funktionen frei von Nebeneffekten sein sollen, was speziell bedeutet, dass keine Variablen außerhalb der Funktion durch die Funktion geändert werden dürfen. Was allerdings, nebenbei bemerkt, nicht bedeutet, dass Oracle selbst sich konsequent an diese Empfehlung hielte … Egal, wir können uns vorstellen, dass zumindest in SQL diese Funktion auch nicht zu benutzen ist:

```
SQL>  select get_return_date(p_day_amount => 30)
  2     from dual;
 select get_return_date(p_day_amount => 30)
        *
FEHLER in Zeile 1:
ORA-06553: PLS-306: Falsche Anzahl oder Typen von Argumenten
                    in Aufruf von 'GET_RETURN_DATE'
```

Listing 7.31 Ausgabeparameter sind in SQL-Anweisungen nicht erlaubt.

Das ergibt Sinn, denn wir haben keine Variable, die die Werte von `p_start_date` aufnehmen könnte, da SQL so etwas wie Variablen nicht kennt. Doch wenn wir schon einmal dabei sind, wollen wir auch sehen, ob dieser Extremfall in PL/SQL funktioniert:

```
SQL> declare
  2    l_internal_date date;
  3  begin
  4    l_internal_date := sysdate;
  5    dbms_output.put_line(
  6      'Rückgabedatum ist: ' ||
  7      get_return_date(l_internal_date, 30));
```

```
 8    dbms_output.put_line(
 9      'Ausleihdauer: ' ||
10      trunc(l_internal_date - sysdate) || ' Tage');
11  end;
12  /
Rückgabedatum ist: 30.09.13
Ausleihdauer: 35 Tage
PL/SQL-Prozedur erfolgreich abgeschlossen.
```

Listing 7.32 Funktion mit zwei »Rückgabekanälen«

Wenn Sie sich diesen anonymen Block etwas genauer ansehen, stellen Sie fest, dass einmal das Rückgabedatum der Funktion ausgegeben wird (Zeile `5ff`.) und dass anschließend mit dem – in der Zwischenzeit durch die Funktion geänderten – Datum gerechnet wird. Nachdem wir in Zeile `4` der Variablen `date` das aktuelle Systemdatum zugewiesen haben, ziehen wir nach dem Aufruf der Funktion das Systemdatum in Zeile `10` von diesem Datum ab (das `trunc` dient dazu, die Zeit aus dem Datum auf Mitternacht abzurunden, um ein glattes Ergebnis zu erhalten). Normalerweise sollten wir eigentlich eine `0` als Ergebnis erhalten, doch sehen wir an der Ausgabe, dass wir tatsächlich sowohl ein Datum zurückgegeben als auch die Variable geändert haben. Doch ist das sinnvoll? Mir fällt eigentlich nur ein Fall ein, bei dem so etwas akzeptabel erscheinen *könnte*.

Stellen wir uns eine Prozedur vor, die verschiedene Ausgabeparameter berechnet. Der aufrufende Block möchte wissen, ob die Berechnung aus Sicht der Prozedur erfolgreich verlaufen ist. Nun könnte man darüber nachdenken, in diesem Fall anstelle eines Fehlers, den die Prozedur auslöst (oder nicht), eine Funktion mit einem Rückgabewert vom Typ `boolean` (`true` | `false` | `null`) zu definieren, die `true` liefert, wenn alles in Ordnung ist. Die einzelnen Ergebnisse können dann über die Ausgabeparameter erfragt werden.

Eine solche Verwendung von Ausgabeparametern in Funktionen könnten Sie dann tolerieren, wenn das Entwicklerteam dies zulässt und daher damit rechnet, dass solche Funktionen auftauchen können. Dennoch bleibt ein ungutes Gefühl zurück: Normalerweise sollten diese wenig intuitiven Konstruktionen vermieden werden. Sie erinnern an eine Gießkanne, die nicht nur aus dem Ausguss, sondern auch noch aus allen möglichen anderen Öffnungen Wasser austreten lässt.

Lassen Sie uns hier einen vorläufigen Strich ziehen: Wir werden im weiteren Verlauf des Buches noch umfassendere Konzepte zu Prozeduren und Funktionen kennenlernen, die die Mächtigkeit, aber auch die Komplexität dieser Blöcke noch deutlich erhöhen. Seien Sie also gespannt! Nun wende ich mich erst den weiteren Formen zu, in denen wir Blöcke noch antreffen können: den Triggern und den Packages.

7.4 Datenbanktrigger

Ein Trigger ist, das hatten wir schon in einigen kurzen Beispielen gesehen, ein Block, der implizit von der Datenbank aufgerufen wird, wenn ein definiertes Ereignis eintritt. Wir werden uns in Kapitel 11, »Events in der Datenbank: Programmierung von Triggern«, noch eingehend mit den logischen Problemen der Triggerprogrammierung auseinandersetzen. Daher werde ich hier nur den Trigger bezüglich des Aufrufs mit den Prozeduren und Funktionen vergleichen.

Die Definition eines Triggers unterscheidet sich von der einer normalen Prozedur oder Funktion nicht nur dadurch, dass eine andere Einleitungsklausel verwendet wird, sondern auch dadurch, dass die eigentliche Funktionalität des Triggers über einen *anonymen PL/SQL-Block* an die Triggerdefinition angehängt wird. Daher wird, wenn Variablen in einem Trigger benötigt werden, auch nicht das Schlüsselwort `as` oder `is` verwendet, sondern `declare`.

Sehen wir uns einen Beispieltrigger an:

```
SQL> connect hr/hr
Connect durchgeführt.
SQL> create or replace
  2  trigger update_job_history
  3    after update of job_id, department_id
  4    on employees
  5    for each row
  6  declare
  7    l_end_date date := sysdate;
  8  begin
  9    add_job_history(
 10      :old.employee_id,
 11      :old.hire_date,
 12      l_end_date,
 13      :old.job_id,
 14      :old.department_id);
 15  end;
 16  /
Trigger wurde erstellt.
```

Listing 7.33 Beispiel für einen einfachen Trigger

Wir erkennen einige Klauseln, die festlegen, wann der Trigger ausgeführt werden soll. Die Alternativen werden wir uns noch genauer ansehen; hier reicht es uns, zu verstehen, dass dieser Trigger immer dann ausgeführt wird, *nachdem* (`after update`) eine Zeile der Tabelle `employees` aktualisiert worden ist – genauer gesagt: nachdem eine der Spalten `job_id` oder `department_id` aktualisiert worden ist. Der Trigger wird, wie

wir an der Klausel `for each row` erkennen, einmal für jede geänderte Zeile ausgeführt und nicht nur einmal für die gesamte `update`-Anweisung.

Ein weiterer wesentlicher Unterschied zu einer Prozedur oder einer Funktion ist, dass wir an einen Trigger (naturgemäß) keine Parameter übergeben können, da er sich ja in SQL abspielt, wo es keine Variablen gibt. Dazu gibt es eigentlich nur eine Ausnahme: Wenn ein Trigger, wie im obigen Beispiel, für jede Zeile, die geändert wird, aufgerufen wird, stehen standardmäßig die beiden Pseudovariablen `new` und `old` zur Verfügung. Diese beiden Pseudovariablen stellen eine Struktur dar, die einer Zeile der Tabelle entspricht, auf die sich der Trigger bezieht. Auf die einzelnen Spaltenwerte können Sie über die Notation `:new.<Spaltenname>`, also z. B. `:new.employee_id`, zugreifen.

Interessant ist jedoch, wie diese Pseudovariable angesprochen werden muss: Wir erkennen den Doppelpunkt vor der Variablen. Diese spezielle Notation wird immer dann verwendet, wenn die Variable nicht in PL/SQL, sondern in der aufrufenden Umgebung (in diesem Fall: SQL) deklariert wurde. Das klingt zunächst seltsam, bei näherem Hinsehen ist aber nachvollziehbar, was hier passiert:

- Innerhalb einer SQL-Anweisung wird ein Ereignis ausgelöst. Diesem Ereignis werden die aktuellen Werte der Zeile mitgegeben, an der gerade gearbeitet wird (je nach Zusammenhang: alter und neuer Zustand).
- Diese Werte werden der PL/SQL-Umgebung über die Pseudovariablen `new` bzw. `old` zugänglich gemacht. Da diese Variablen schon durch SQL gefüllt werden müssen, sind sie also auch dort definiert und nicht in PL/SQL.
- Um dem anonymen Block klarzumachen, dass die Deklaration der Variablen nicht innerhalb von PL/SQL erfolgt ist, wird der Doppelpunkt vorangestellt.

Ein ähnliches Muster sehen wir auch dann, wenn eine Variable z. B. in SQL*Plus vereinbart und dann an PL/SQL übergeben wird.

Doch: Warum eigentlich *Pseudovariable*? Nun, wäre es eine »normale« Variable, ließe sie sich z. B. als Parameter an eine Prozedur übergeben. Das ist aber nicht der Fall. Die Pseudovariablen `new` und `old` existieren ausschließlich im Umfeld des anonymen Blocks, der durch den Trigger ausgeführt wird. Wollen Sie die Werte der Spalte als Parameter an eine Prozedur oder Funktion übergeben, müssen Sie die Werte in entsprechende Prozedurparameter umkopieren. Die Pseudovariablen fungieren eher als eine Art »Anfasser« für die entsprechenden Spaltenwerte, stellen selbst aber keine Variablen im eigentlichen Sinne dar.

7.5 Packages

Auch die Packages sehen wir uns in Kapitel 12, »Packages«, noch eingehend an; daher werde ich hier nur eine ungefähre Vorstellung vom Einsatzbereich geben. Ein Package

ist eine Gruppe von Prozeduren, Funktionen und Variablen unter einem gemeinsamen Dach. Allerdings bieten Ihnen Packages darüber hinaus einige wesentliche Vorteile:

- Sie strukturieren Ihren Code und helfen Ihnen, den Überblick zu behalten.
- Sie bieten ein Konzept zur Trennung öffentlicher und privater Hilfsprozeduren.
- Sie erlauben das Überladen von Prozeduren und Funktionen, womit gemeint ist, dass der gleiche Methodenname mit unterschiedlichen Parametern verwendet werden kann.
- Sie bieten Performance-Vorteile, weil das gesamte Package auf einmal in den Speicher geladen wird und anschließend ohne weiteres Nachladen direkt zur Verfügung steht.

Sehen wir uns aber zunächst einmal das grobe Konzept an. Packages werden in zwei Schritten definiert: Zunächst wird die *Package-Spezifikation* und dann der *Package-Körper* erstellt.

7.5.1 Package-Spezifikation

Die Package-Spezifikation deklariert die öffentlich zugängliche Schnittstelle zum Package. Alles, was Sie hier deklarieren, kann daher von außen gesehen und angesprochen werden. Prozeduren und Funktionen werden lediglich über ihren Namen, die Parameter und (bei Funktionen) ihren Rückgabewert deklariert, aber nicht implementiert. Die Implementierung erfolgt später – im Package-Körper. Im folgenden Beispiel sehen Sie die Deklaration eines Packages zur Sammlung von Hilfswerkzeugen:

```
SQL> create or replace package tools
  2  as
  3    g_std_day_amount number;
  4
  5    function get_xml_date_string (
  6      p_date_in in date,
  7      p_with_time in char default 'Y')
  8      return varchar2;
  9
 10    function get_return_date(
 11      p_start_date in date default sysdate,
 12      p_day_amount in number default g_std_day_amount)
 13      return date;
 14
 15    procedure print_return_date(
 16      p_start_date in date default sysdate,
 17      p_day_amount in number default g_std_day_amount);
```

```
 18  end tools;
 19  /
Package wurde erstellt.
```

Listing 7.34 Eine einfache Package-Spezifikation

Die Anweisung hält vor allem Bekanntes bereit: Das Schlüsselwort `package` ersetzt `procedure` oder `function`, doch ansonsten findet sich die Deklaration von Variablen, Prozeduren und Funktionen im Deklarationsbereich des Packages. Auffällig ist allerdings, dass dieser Block keinen Implementierungsteil, eingeleitet durch `begin`, enthält. Die Implementierung erfolgt im Package-Körper.

Ich habe noch ein kleines Extra hinzugefügt: In Zeile `3` definiere ich eine Variable `g_std_day_amount` vom Typ `number`, der ich hier noch keinen Standardwert zuweise. (Dies werden wir später im Package-Körper tun.) Dieser Variablen ist wieder ein weit verbreitetes Präfix vorangestellt, nämlich `g_`, das für *global* steht und andeutet, dass diese Variable global für das gesamte Package deklariert wurde. Alle Prozeduren und Funktionen dieses Packages können global deklarierte Variablen verwenden.

Anschließend verwende ich diese Variable in der Deklaration der Prozeduren und Funktionen. Auf diese Weise kann ich das Standardverhalten meiner Prozeduren und Funktionen auf einfache Art einstellen, indem ich den Wert der Variablen ändere. Objektorientierte Entwickler werden einen solchen direkten Zugriff auf eine Package-Variable möglicherweise nicht gutheißen, sondern den Zugriff über eine Funktion oder Prozedur (allgemein: eine Methode) bevorzugen. Doch befinden wir uns einerseits nicht im Bereich der objektorientierten Programmierung, und andererseits können Sie, wenn Sie mögen, diesen Weg über eine `get`- bzw. `set`-Methode natürlich auch einschlagen. In diesem Fall deklarierten Sie die Variable nicht hier, sondern lediglich die Methode, die die Variable ändert oder ausgibt. Die Variable selbst wandert dann in den Package-Körper, wo sie gegen den direkten Zugriff geschützt ist. Allerdings ist diese Methode dann nicht als Vorgabewert für eine andere Methode nutzbar, wie ich das oben getan habe.

Packages werden allerdings durchaus auch im oben gezeigten Sinn als »Sammelbecken« für global gültige Variablen und Konstanten genutzt, so z. B. für Programmversionen, Firmennamen, Umrechnungsfaktoren etc.

7.5.2 Package-Körper

Die Erstellung eines Package-Körpers folgt zunächst der Definition der Package-Spezifikation, diesmal erweitert um die Implementierung der Prozeduren und Methoden. Zudem wird die Definition von Variablen, die bereits in der Package-Spezifikation deklariert wurden, im Package-Körper nicht wiederholt. Allerdings können weitere Variablen, Prozeduren und Funktionen deklariert werden, die lediglich innerhalb des

Packages sichtbar sind. Zudem kann der Package-Körper selbst einen Implementierungsteil enthalten, der lediglich einmal pro Session ausgeführt wird, nämlich beim Laden des Packages. In diesem Teil werden normalerweise Initialisierungsarbeiten durchgeführt. Sehen wir uns ein Beispiel für die Deklaration eines Package-Körpers an:

```
SQL> create or replace package body tools as
  2
  3    /* Private Methoden */
  4    procedure initialize
  5    as
  6    begin
  7      -- Initialisiere g_std_day_amount auf 24
  8      g_std_day_amount := 24;
  9    end initialize;
 10
 11    /* Öffentliche Methoden */
 12    function get_xml_date_string(
 13      p_date_in in date,
 14      p_with_time in char default 'Y')
 15      return varchar2
 16    as
 17      l_format_mask varchar2(30) := 'YYYY-MM-DD';
 18    begin
 19      if lower(p_with_time)
 20       in ('y', 'j', 'yes', 'ja', 'true', 'wahr', '1')
 21       then l_format_mask := l_format_mask || '"T"HH24:MI:SS';
 22      end if;
 23      return to_char(p_date_in, l_format_mask);
 24    end get_xml_date_string;
 25
 26    procedure print_return_date(
 27      p_start_date in date default sysdate,
 28      p_day_amount in number default g_std_day_amount)
 29    as
 30    begin
 31      dbms_output.put_line(
 32       'Rückgabedatum ist: ' ||
 33        get_return_date(p_start_date, p_day_amount));
 34    end print_return_date;
 35
 36    function get_return_date(
 37      p_start_date in date default sysdate,
```

```
 38      p_day_amount in number default g_std_day_amount)
 39      return date
 40    as
 41    begin
 42      return next_day(p_start_date + p_day_amount, 'MON');
 43    end get_return_date;
 44
 45  begin
 46    initialize;
 47  end tools;
 48  /
Package-Body wurde erstellt.
```

Listing 7.35 Ein einfacher Package-Körper

Wenn Sie sich den Code ansehen, fällt auf, dass zum Ende die Prozedur `initialize` aufgerufen wird. Diese Prozedur wird lediglich beim ersten Laden des Packages in einer Session ausgeführt und setzt die öffentlich sichtbare Variable `g_std_day_amount` auf den Wert `24`, *bevor* eine der Prozeduren und Funktionen aufgerufen werden kann. Daher haben wir anschließend einen gültigen Standardwert, der allerdings nicht in der Spezifikation definiert wurde und daher erst zur Laufzeit festgelegt werden konnte. Mit diesem Mittel könnte der Wert z. B. aus anderen Tabellen kopiert oder in Abhängigkeit von anderer Logik bestimmt werden. Beachten Sie: Die Funktion `initialize` kann nur innerhalb des Packages aufgerufen werden, außerhalb des Packages ist sie nicht sichtbar.

Dieses Package stellt lediglich einige der Funktionen zusammen, die Sie ja bereits in dieser oder ähnlicher Form kennen. Beachten Sie auch, auf welche Weise die Prozedur `print_return_date` implementiert wurde: Sie ruft die Funktion `get_return_date` auf. Diese Art der Programmierung erfordert, dass die Funktion `get_return_date` bereits bekannt ist, wenn sie aufgerufen wird. In unserem Beispiel ist das kein Problem, denn die Funktion `get_return_date` ist eine öffentliche Funktion, die bereits in der Package-Spezifikation deklariert wurde. Handelt es sich um eine interne Hilfsfunktion (wie etwa die Funktion `initialize`), muss die Definition vor dem Aufruf erfolgt sein. Daher werden Hilfsfunktionen normalerweise als Erstes in Package-Körpern implementiert.

Theoretisch ist es ebenfalls möglich, innerhalb eines Package-Körpers eine Hilfsmethode zunächst nur zu deklarieren und, z. B. am Ende des Packages, zu implementieren. Dieser sehr selten genutzte Fall wird als *Vorwärtsdeklaration* bezeichnet. Manchmal kann dieser Mechanismus erforderlich sein, z. B. dann, wenn zwei Methoden sich gegenseitig aufrufen. In diesem Fall kann die Implementierung der einen Methode erst erfolgen, nachdem die andere bekannt ist, und umgekehrt. Aus diesem Dilemma hilft uns dann die Vorwärtsdeklaration.

7.5.3 Aufruf von Prozeduren und Methoden in Packages

Den Aufruf einer Package-Prozedur kennen Sie bereits: Wir hatten ja verschiedentlich mit den mitgelieferten Prozeduren von Oracle gearbeitet, z. B. mit dem Package `dbms_output`. Wir verwenden unser Package ganz genauso:

```
SQL> call tools.print_return_date();
Rückgabedatum ist: 23.09.13
Aufruf wurde abgeschlossen.
```

Analog können wir Funktionen des Packages auch in SQL verwenden:

```
SQL> select tools.get_return_date(
  2                 day_amount => 36) ergebnis
  3    from dual;
ERGEBNIS
----------
07.10.2017
```

Listing 7.36 Aufruf von Package-Prozeduren und -Funktionen

Merke

Beachten Sie bitte einen syntaktischen Unterschied zwischen einem anonymen Block und der `call`-Anweisung: In der `call`-Anweisung ist es erforderlich, dem Prozedurnamen ein leeres Klammerpaar mitzugeben, wenn Sie keine Parameter übergeben, in einem anonymen Block ist dies zwar möglich, aber nicht verpflichtend.

[!]

Zusammenfassend kann man sagen, dass Packages zunächst einmal die Übersichtlichkeit erhöhen. Dafür ist nicht nur verantwortlich, dass Funktionen, die zu einem Themengebiet gehören, zusammengestellt werden, sondern auch, dass Funktionen, die lediglich Hilfsaufgaben wahrnehmen, von der direkten Benutzung ausgenommen und lediglich in der Implementierung eines Packages vorhanden sind. Dies erhöht nicht nur die Übersichtlichkeit, sondern auch die Sicherheit Ihres Codes. In Kapitel 12, »Packages«, werden Sie weitere mächtige Funktionen von Packages kennenlernen. Doch bereits hier sei gesagt: Packages sollten Ihr Standard bei der Programmierung von Datenbanken sein. Einzelne Prozeduren und Funktionen gehören normalerweise nicht in eine Anwendung.

7.6 Ausführungsrechte von PL/SQL-Blöcken

Normalerweise ist alles ganz einfach: PL/SQL-Blöcke werden mit den Rechten des Benutzers ausgeführt, der sie auch erstellt hat. Bei benannten PL/SQL-Blöcken kann

das aber eine interessantere Frage werden: Stellen wir uns vor, wir erhielten das Recht, eine Prozedur oder ein Package eines anderen Benutzers zu verwenden. Wir selbst sind mit relativ geringen Rechten an der Datenbank angemeldet, während der Autor des Packages oder der Prozedur sehr umfangreiche Rechte hatte. Mit welchen Rechten wird denn nun die Prozedur ausgeführt?

Auch hier gilt das Grundprinzip, dass die Ausführungsrechte des Eigentümers zugrunde gelegt werden. Das bedeutet, dass die Prozedur all das tun darf, was der Eigentümer tun darf, nicht aber (nur) das, was derjenige tun darf, der die Prozedur aufruft. Die Prozedur soll es dem Aufrufenden ermöglichen, das zu tun, was die Prozedur tun soll, unabhängig davon, ob der Aufrufende dies auch ansonsten tun dürfte. Dadurch können wir sehr genau festlegen, auf welche Weise ein Benutzer z. B. Daten einsehen oder verändern darf. Oracle nennt eine solche Prozedur (Funktion, Package ...) eine DR-Prozedur (DR = *Definers Right*). Dies ist der Standard. An einem Szenario können wir uns diesen Mechanismus klarmachen: Sie möchten eine API für den Zugriff auf Daten programmieren. Die Anwender sollen allerdings nicht die Möglichkeit haben, die Tabellen direkt zu bearbeiten. Stattdessen erstellen Sie eine Reihe von Packages, deren Methoden für die Verwaltung der Daten zuständig sind. Anschließend vergeben Sie lediglich Ausführungsrechte an den Prozeduren an andere Benutzer, nicht aber Lese- oder Schreibrechte an den Tabellen. Ruft ein solcher Anwender eine Prozedur auf, ändert die Prozedur dennoch die Daten, weil der Eigentümer der Prozedur Zugriffsrechte auf die Tabelle, der aufrufende Benutzer diese jedoch nicht besitzt.

Manchmal ist es jedoch besser, wenn eine Prozedur zentral definiert werden kann, aber von jedem Benutzer mit seinen eigenen Rechten ausgeführt wird. Eine solche Prozedur nennt Oracle eine IR-Prozedur (IR = *Invokers Right*). Denken wir uns dazu ein Szenario, in dem die verschiedenen Benutzer jeweils gleiche Tabellen besitzen. Dies könnte z. B. der Fall sein, wenn mehrere Datenbanken in verschiedenen Regionen eingesetzt werden und die jeweiligen Datenbanken zwar die gleichen Tabellen, aber nur die regional gültigen Daten besitzen. Die aufrufenden Benutzer haben Zugriff auf die lokalen Tabellen, nicht aber der Eigentümer der Prozedur. Nun können die aufrufenden Anwender die Daten (nur) ihrer Tabellen ändern, während gleichzeitig die gesamte Code-Basis für alle Anwender zentral gepflegt werden kann, weil die Prozedur die Rechte des aufrufenden Benutzers zugrunde legt. Hätten wir hier den Ansatz DR gewählt, müssten wir den Code auf allen Datenbanken im Namen der entsprechenden Anwender neu erstellen, da der zentrale Eigentümer der Packages die Zugriffsrechte auf die verschiedenen Tabellen ja nicht besitzt.

Um eine Prozedur zu erstellen, die mit den Rechten des ausführenden Benutzers arbeitet, muss bei der Deklaration der Prozedur, Funktion oder des Packages die Klausel `authid` gesetzt werden. Standardmäßig wird die Klausel mit dem Wert `definer` belegt. Wir können uns also die Deklaration einer »normalen« DR-Prozedur wie folgt vorstellen:


```
create or replace procedure my_proc
  authid definer ...
```

Im Gegensatz dazu wird eine IR-Prozedur so definiert:

```
create or replace procedure my_proc
  authid current_user ...
```

Wir werden diesen Mechanismus in den weiteren Kapiteln noch häufiger einsetzen. Zur Illustration können wir uns aber folgende kleine Prozedur ansehen, die den Namen des aktuellen Schemas des angemeldeten Benutzers ausgibt. Dazu benutzen wir die Funktion `sys_context`, die den Zugriff auf einen Kontext gewährt. Wir verwenden den Kontext `USERENV`, der für den angemeldeten Benutzer einige Parameter bereithält. Ich bin davon überzeugt, dass Sie den Ablauf der Prozedur und das Ergebnis selbst interpretieren können:

```
SQL> connect scott/tiger
Connect durchgeführt.

SQL> create or replace procedure print_schema
  2    authid current_user
  3  as
  4  begin
  5     dbms_output.put_line('Aktuelles Schema: ' ||
  6       sys_context('USERENV', 'CURRENT_SCHEMA'));
  7  end print_schema;
  8  /
Prozedur wurde erstellt.

SQL> grant execute on print_schema to public;
Benutzerzugriff (Grant) wurde erteilt.

SQL> connect hr/hr
Connect durchgeführt.

SQL> set serveroutput on
SQL> call scott.print_schema();
Aktuelles Schema: HR
Aufruf wurde abgeschlossen.
```

Zum Vergleich sehen Sie hier die gleiche Methode mit dem Standardverfahren:

```
SQL> connect scott/tiger
Connect durchgeführt.
```

```
SQL> create or replace procedure print_schema
  2    authid definer
  3  as
  4  begin
  5    dbms_output.put_line('Aktuelles Schema: ' ||
  6      sys_context('USERENV', 'CURRENT_SCHEMA'));
  7  end print_schema;
  8  /
Prozedur wurde erstellt.

SQL> grant execute on print_schema to public;
Benutzerzugriff (Grant) wurde erteilt.

SQL> connect hr/hr
Connect durchgeführt.

SQL> set serveroutput on
SQL> call scott.print_schema();
Aktuelles Schema: SCOTT

Aufruf wurde abgeschlossen.
```

Listing 7.37 Beispiel für die Verwendung der Klausel »authid current_user«

Wie Sie sehen, wird nur bei einer IR-Prozedur der Name des aktuellen Benutzers ausgegeben. Die DR-Prozedur arbeitet im Namen ihres Eigentümers und weiß nicht, welcher Benutzer sie aufgerufen hat.

7.7 Compiler-Anweisungen (Pragma)

PL/SQL wird vor der Ausführung kompiliert, und das Kompilat wird im Data Dictionary hinterlegt, falls es sich um einen benannten PL/SQL-Block handelt. Der Kompiliervorgang kann durch Parameter beeinflusst werden. Diese Compiler-Anweisungen werden durch die Klausel `PRAGMA` vereinbart. Oracle hält eine Reihe dieser Compiler-Anweisungen bereit, die zum Teil außerhalb des Fokus dieses Buches liegen. Andere Anweisungen werden allerdings durchaus häufiger eingesetzt. Sehen wir uns jetzt einige wichtige Compiler-Anweisungen an.

7.7.1 Die autonome Transaktion

Stellen Sie sich vor, Sie möchten den Zugriff auf eine Tabelle protokollieren. Sobald ein Benutzer z. B. eine Löschaktion auf eine Tabelle ausführt, möchten Sie davon

erfahren. Dazu bietet sich eventuell ein Trigger an, der immer dann, wenn eine Löschaktion auf die Tabelle ausgeführt wird, eine Zeile in eine Audit-Tabelle einfügt, die diesen Zugriff protokolliert. Da beide Anweisungen, die `delete`-Anweisung auf die Tabelle sowie die `insert`-Anweisung auf die Audit-Tabelle, transaktionsgeschützt werden, werden also auch beide Anweisungen durch ein anschließendes `rollback` widerrufen. Der Audit-Eintrag verschwindet, wenn die Löschaktion zurückgenommen wird.

Doch ist dieses Verhalten oftmals nicht erwünscht: Uns interessiert, dass die Aktion bereits *versucht* wurde, denn dieser Versuch könnte bereits einen Verstoß gegen geltende Geschäftsregeln darstellen. Doch wie können wir Teile einer Transaktion erhalten, andere aber nicht? Das ist so nicht möglich. Oracle bietet für solche Fälle allerdings die Möglichkeit, eine »innere« Transaktionsklammer in einer bereits bestehenden Transaktionsklammer zu öffnen und diese separat zu bestätigen. Diese innere Transaktionsklammer muss dann durch eine `commit`-Anweisung bestätigt werden, damit die Daten unabhängig von der äußeren Transaktion bestehen bleiben.

Dieses Verhalten erreichen wir, indem wir in einem PL/SQL-Block das Pragma `autonomous_transaction` vereinbaren. Wir teilen damit dem Compiler mit, dass für diese Prozedur eine eigene Transaktionsklammer geöffnet wird, die unabhängig von der äußeren Transaktionsklammer ist und innerhalb der Prozedur durch `commit` oder `rollback` geschlossen wird:

```
create procedure audit_entry (<parameter_list>)
as
  pragma autonomous_transaction
begin
  insert into audit_table values(<parameter_list>);
  commit;
end audit_entry;
```

Listing 7.38 Beispiel für die Verwendung des Pragmas »autonomous_transaction«

Achtung: Autonome Transaktionen außerhalb dieses Beispiels können sehr unangenehme logische Konsequenzen haben. Die Anwendungsgebiete sind deutlich kleiner, als Sie möglicherweise zunächst vermuten. Ein weiterer Hinweis: Auditing von Datenbanken programmieren Sie nicht selbst (oder nur im wirklich gut begründeten Ausnahmefall), weil Oracle in der Datenbank eine ungleich mächtigere Audit-Möglichkeit bereithält. Bei Interesse sehen Sie sich doch einmal den *Database Security Guide* an.

7.7.2 Initialisierung eigener Fehler

Wenn man Anwendungen schreibt, stellt sich sehr bald auch die Frage, ob man eigene Fehlermeldungen definieren kann. In PL/SQL stehen uns dafür einige Metho-

den zur Verfügung, die Sie in Kapitel 14, »Exception«, noch genauer kennenlernen und in Kapitel 20, »Workshop: PL/SQL Instrumentation Toolkit (PIT)«, zur Meisterschaft führen werden. Ein Berührungspunkt der Fehlerbehandlung mit einer `pragma`-Anweisung tritt allerdings dann auf, wenn ein Oracle-Fehler, der noch keinen Oracle-Namen erhalten hat, im Fehlerbehandlungsteil aufgefangen werden soll. Oracle hat einige häufige Fehler mit einer Konstanten benannt, die im Fehlerbehandlungsteil herangezogen werden können, um auf diesen spezifischen Fehler zu reagieren. Andere Fehler haben noch keine solche Benennung und können daher nicht explizit behandelt werden. Daher muss eine *unbenannte* Ausnahme von Oracle mit einem Namen verbunden werden. Dies geschieht wie folgt über ein Pragma:

```
SQL>declare
  1   deadlock_detected EXCEPTION;
  2   pragma exception_init(deadlock_detected, -60);
  3 begin
  4    null; -- Hier wird irgendetwas getan,
  5          -- um einen ORA-00060-Fehler zu provozieren
  6 exception
  7    when deadlock_detected then
  8       raise; -- Hier wird der Fehler behandelt
  9 end;
```

Listing 7.39 Beispiel für die Deklaration einer Fehlerkonstanten

Fehler, die von Oracle ausgelöst werden, aber noch keine Bezeichnung haben, können also auf diese Weise benannt und im Fehlerbehandlungsteil abgefangen werden.

7.8 Best Practices

Welche Empfehlungen können für die Verwendung der einzelnen Blocktypen gegeben werden? Zunächst einmal die Empfehlung, die Entwicklung von Anwendungen in PL/SQL, wo immer möglich, über Packages zu realisieren.

- Packages strukturieren Code.
- Packages bieten Performance-Vorteile.
- Packages schaffen die Möglichkeit der Trennung von Deklaration und Implementierung.
- Packages trennen öffentliche von privaten Methoden.
- Packages erlauben das Überladen von Methoden (siehe Kapitel 12, »Packages«).

Prozeduren und Funktionen sind die Arbeitspferde der PL/SQL-Programmierung. Wenn die Geschäftsregeln erfordern, dass beim Einfügen oder Ändern von Daten

mehrere Tabellen harmonisiert werden müssen, ist Ihr erster Reflex möglicherweise, das mit einem Trigger zu erledigen. Einfacher und klarer ist dies aber häufig in Prozeduren und Funktionen – als der Zugriffsschicht über den Tabellen – umzusetzen. Zudem hat der Ansatz, eine API auf Daten mithilfe von Packages und Zugriffsmethoden zu realisieren, gegenüber dem Einsatz von Triggern den Vorteil, dass die aufrufende Programmumgebung die Details des Datenmodells nicht kennen muss, denn sie benötigt lediglich die Kenntnis der Parameter der öffentlichen API. Daher empfiehlt sich dieser Ansatz im Vergleich zu einem triggerbasierten Ansatz.

Bezüglich der Ausführungsrechte von Prozeduren, Funktionen und Packages ist der Standard, Definer-Rights-Prozeduren zu verwenden. Sollten Sie von diesem Ansatz abweichen (müssen), ist eine gute Dokumentation dieses Ansatzes wichtig, damit eventuelle Fehler, die sich aus dieser Situation ergeben, schnell erkannt werden. Je nach Situation kann es aber auch erforderlich sein, eine ganze Gruppe von Packages mit Invokers Rights auszustatten, z. B. um Code für mehrere Datenbanken zentral entwickeln zu können. Dann ist dieser Ansatz sicher besser, als gleichen Code in mehreren Datenbanken zu duplizieren.