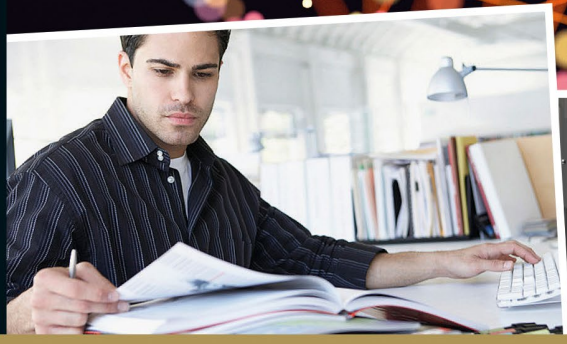
Jürgen Sieben



```
    dbms_lob.wr
  commit;
end;
    /
PL/SQL-Prozedur

SQL> selec
  2    fro
```

Oracle PL/SQL

Das umfassende Handbuch

Das Standard-werk

- Performante und skalierbare Anwendungen mit PL/SQL, aktuell zu 19c, 21c und 23c
- Trigger, Stored Procedures, Oracle Packages
- Datensicherheit und Konsistenz, PL/SQL in der Datenbank-administration u. v. m.

4., aktualisierte Auflage

Rheinwerk Computing

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 Datenbanklinks, 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 Erklärungen 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 Administratoren wichtiger als für die Entwickler sind. Der häufigste Tabellentyp ist die *Heap Organized Table*, eine »normale« Datenbanktabelle. Zudem bietet Oracle die *Index Organized Table*, die *Global (Private) 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 auf-

nehmen 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 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 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 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 dieses Tabellentyps 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.

Aus der Liste der Voraussetzungen ergibt sich, dass ein erster Kandidat in den sogenannten Lookup-Tabellen besteht, also Tabellen, die lediglich benannte Werte unter einem Schlüssel speichern. Beispiele wären eine Tabelle mit den Anreden oder akademischen Titeln, aber auch Berufslisten und viele weitere. Diese Tabellen eignen sich, weil sie normalerweise ausschließlich über ihren Primärschlüssel angesprochen werden (sie werden für das Berichtswesen im Regelfall zur eigentlichen Tabelle gejoint) und andererseits keine weiteren Beziehungen mit anderen Tabellen eingehen. Zudem sind die Zeilen solcher Tabellen im Regelfall sehr kurz.

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>
```

Listing 4.1 Beispielanweisung zur Erstellung einer temporären Tabelle

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.

Version 18c der Datenbank fügt einen weiteren Typ temporärer Tabellen hinzu, die *private temporäre Tabelle*. Im Unterschied zur globalen temporären Tabelle ist sie nicht allen Benutzern bekannt, sondern kann innerhalb einer Session erstellt, genutzt und anschließend gelöscht werden. Der Name einer solchen Tabelle muss allerdings einer Konvention gehorchen, die durch einen Initialisierungsparameter der Datenbank festgelegt wird. Ist hier nichts Weiteres bestimmt, muss der Name der Tabelle mit `ORA$PTT_` beginnen.

Analog zum Erstellen einer globalen temporären Tabelle, können wir auch für die private temporäre Tabelle festlegen, ob diese eine Transaktion überleben soll. Allerdings geht es hier nicht darum, die Daten zu behalten (das würde sie ebenfalls tun, wenn die Option gesetzt ist), sondern grundsätzlich darum, ob diese Tabelle anschließend überhaupt noch existiert.

```
Create private temporary table ora$ptt_my_private_temp (
  id number,
  description  varchar2(20 char)
) on commit drop|preserve definition;
```

Listing 4.2 Beispielsyntax zur Anlage einer privaten temporären Tabelle

Listing 4.2 zeigt die Erstellung einer entsprechenden Tabelle mit der Wahlmöglichkeit `drop` oder `preserve` in der letzten Klausel. Es fällt mir etwas schwer, sinnvolle Einsatzbereiche für diesen Tabellentyp zu finden, und ich vermute, dass die Kompatibilität mit anderen Datenbanken eine Rolle bei der Einführung gespielt hat. Wir werden diesen Tabellentyp in diesem Buch nicht benötigen.

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). Version 11g erweitert dies um das Verfahren *Interval*, das 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. Umgekehrt: 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 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 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 rownum, rowid, emp_last_name, emp_job_id
  2    from hr_employees;

    ROWNUM ROWID              EMP_LAST_NAME             EMP_JOB_ID
---------- ------------------ ------------------------- ----------
         1 AAAkfSAAhAAABD1AAA King                      AD_PRES
         2 AAAkfSAAhAAABD1AAB Kochhar                   AD_VP
         3 AAAkfSAAhAAABD1AAC De Haan                   AD_VP
...
107 Zeilen ausgewählt.
```

Listing 4.3 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 Datenbankobjekt (wie z. B. eine Tabelle oder ein Index) hat eine interne, eindeutige 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 demo_user.hr_employees e
  6    join dba_data_files f
  7      on dbms_rowid.rowid_to_absolute_fno(
  8           e.rowid, 'DEMO_USER', 'HR_EMPLOYEES') = f.file_id
  9   where rownum < 10;

ROWID              FILE_NAME                      BLOCK_NUMBER POS_IN_BLOCK
------------------ ------------------------------ ------------ ------------
AAAkfSAAhAAABD2AAK C:\...\USERS01.DBF             4342                   10
AAAkfSAAhAAABD2AAJ C:\...\USERS01.DBF             4342                    9
AAAkfSAAhAAABD2AAE C:\...\USERS01.DBF             4342                    4
AAAkfSAAhAAABD1AAA C:\...\USERS01.DBF             4342                    0
...
```

Listing 4.4 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 Hardwarepointer 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 nur wenige 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 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` oder, für die Indizierung von großen Textmengen, mit dem `Oracle Text`-Index. 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, muss aber nicht so sein. Umgekehrt: 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 Perfor-

mance 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. Diese Reduzierung ist erheblich, man spricht als Daumenregel von einer Verlängerung des Schreibprozesses um Faktor 3 pro Index. 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 abhängt, wie der Länge der Zeile und damit der Anzahl der Datenblöcke, die gelesen werden müssen.

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 ergibt 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 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, ergibt 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. Aus diesem Grund kann es sinnvoll sein, die Reihenfolge der Spalten eines Indexes nicht nach der Selektivität zu sortieren, sondern nach der Häufigkeit der Verwendung in `select`-Anweisungen. Wird also häufiger nach dem Nachnamen als dem Vornamen gesucht, sollte sich dies in der Reihenfolge der Spalten im Index niederschlagen, einfach, weil nun häufiger die gesuchte Spalte an erster Position im Index steht. Stellen Sie sich hierfür einfach als Analogie die Suche in einem Telefonbuch vor: Es ergibt keinen Sinn, dort nach einem Vornamen zu suchen, weil das gesamte Telefonbuch durchgelesen werden müsste.

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 ist die Entscheidung nicht digital, sondern ein Index kann in so vielen Werten wählen, wie in einen Block passen. Daher kann ein Suchschritt zwischen sehr vielen unterschiedlichen Pfaden unterscheiden und mit wenigen Suchschritten enorme Datenmengen

durchsuchen. Ein B*Baum-Index bei Oracle ist aus diesem Grund 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 Knoten 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 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 emp_last_name, emp_first_name, emp_hire_date
  2    from hr_employees
  3   where emp_last_name like 'K%'
EMP_LAST_NAME             EMP_FIRST_NAME       EMP_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 BATCHED
|*  2 |   INDEX RANGE SCAN
----------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMP_LAST_NAME" LIKE 'K%')
       filter("EMP_LAST_NAME" LIKE 'K%')
```

Listing 4.5 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 angewendet 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 `rowid`s gespeichert. Das ist die Standardeinstellung. Soll jeder Begriff lediglich genau einmal im Index enthalten sein dürfen, wird dies durch das Schlüsselwort `unique` bei der Erstellung des Index vermerkt:

```
create unique index emp_email_uq
    on hr_employees(emp_email);
```

Listing 4.6 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 »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 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 hr_employees_idx
on hr_employees(emp_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 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* (*Data Manipulation Language*)-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 4.7 Beispiel zum Einsatz eines funktionsbasierten Index

Nebenbei: Eine solche Anweisung verlangt nach einem erläuternden Kommentar, damit ein wohlmeinender Kollege diese Abfrage nicht refaktorisiert.

Funktionsbasierte Indizes können ebenfalls Unique sein wie alle B*Baum-Indizes. Sie können außerdem 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 er-

gäbe, sondern auch 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 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 für viele weitere Ziele 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 eine ganze Reihe weitere schöne Dinge für uns. Sie werden ganz einfach erzeugt:

```
SQL SQL> create or replace view employee_vw
  2  as
  3  select emp_last_name, job_title, dep_name, loc_city
  4    from hr_employees
  5    join hr_departments
  6      on emp_dep_id = dep_id
  7    join hr_jobs
  8      on emp_job_id = job_id
  9    join hr_locations
 10      on dep_loc_id = loc_id;
```

```
View wurde erstellt.

SQL> select *
  2    from employee_vw;

EMP_LAST_NAME   JOB_TITLE                 DEP_NAME             LOC_CITY
--------------- ------------------------- -------------------- ----------
Whalen          Administration Assistant  Administration       Seattle
Hartstein       Marketing Manager         Marketing            Toronto
Fay             Marketing Representative  Marketing            Toronto
Raphaely        Purchasing Manager        Purchasing           Seattle
...
107 rows selected.
```

Listing 4.8 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: 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 automatisch 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 4.9 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. Der Grund ist einfach: Sie enthält persistente Daten.

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), oder 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 Codedateien 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.

Die angesprochene Sprachenvielfalt dieser Datenbankobjekte erfährt neuerdings einen Schub durch die *Multi Language Engine* (MLE). Durch diese Technologie, die auf der *GraalVM* (*https://www.graalvm.org*) beruht, ist es möglich, Code zum Beispiel in JavaScript oder Python zu formulieren und innerhalb der Datenbank auszuführen. Hier ist einiges in Bewegung, allerdings werde ich mich auf PL/SQL konzentrieren.

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.

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 erarbeitet 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 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 27.02.2023
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 weit verbreitete Konvention stellt Variablen ein Präfix voran, um sie von Parametern und vor allem von gleichnamigen Tabellenspalten zu unterscheiden. Ich verwende `l_` (für *local*).
- 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 JavaScript-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 oder in die entsprechenden Kapitel meines SQL-Buchs.

- 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, muss 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 anders aktivieren: 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 22.2.)

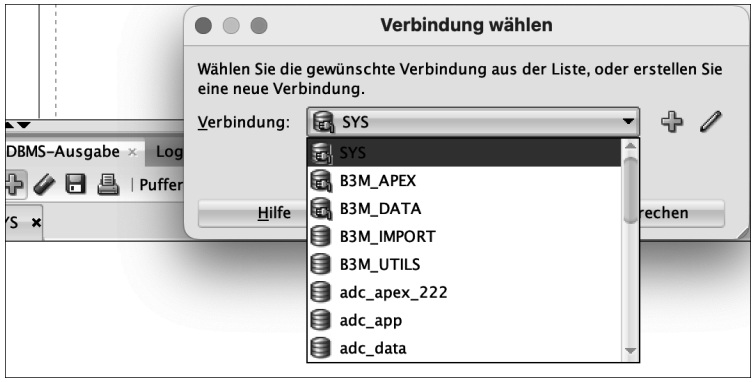


Abbildung 7.1 Server-Output in SQL Developer aktivieren

- 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 Err**or **M**essage), 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 nötig, 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, 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 Aufrufalternativen 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 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 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 Variablendeklarationen von anderen Variablen ableiten

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 zu verschachteln. 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 Pseudocode-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 er 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 werden, 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 als *gespeicherte Prozedur* bezeichnet. Von diesem allgemei-

nen 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 Funktionalität 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, die eine Prozedur erzeugt, 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 128 Zeichen lang sein und keine Leerzeichen enthalten. Bei Versionen vor 12.2 sind nur 30 Zeichen 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 Codeformatierer die Schreibweise ändern.

Anschließend können wir die Prozedur benutzen:

```
SQL> call print_return_date();
Rückgabe am 27.09.2022
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 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 sie 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 aufgeführt. Allerdings ist die Syntax um die Richtungsangabe des Parameters erweitert. Hier stehen die Schlüsselwörter `in`, `out` oder `in out` zur Verfügung. Prozeduren

unterstützen jeweils 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`) ergänzen, indem Sie die Klausel `nocopy` hinzufügen. Dies kann insbesondere bei 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 20.02.2023
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 Codebeispiel 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 schlagen einige PL/SQL-Entwickler auch vor, 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 und Variablen werden sehr häufig nach Spalten einer Tabelle benannt. Die Bezeichner sollten keinesfalls (syntaktisch dürfen sie es, aber dies kann zu fehlerhaftem Code führen) genauso lauten wie die Spaltenbezeichner. Mit dieser Konvention sind Sie diesbezüglich aus dem Schneider und profitieren zudem 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 20.02.2023
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 Sie noch einen Moment, ich komme 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 20.02.2023
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 innerhalb einer größeren Prozedur als Hilfsprozedur im Deklarationsteil der umgebenden Prozedur verwendet werden. Dabei nutzen wir die Fähigkeit von PL/SQL, Blöcke ineinanderzuschachteln, 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);
20.02.2023
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 so-

wohl 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);
20.02.2023
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 ich im nächsten Abschnitt besprechen werde) 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 Sie, wenn Sie unbedingt möchten, auch gleichzeitig nutzen. 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 bedeutet 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('24.01.2023', 'DD.MM.YYYY'), 30);
print_return_date(date '2023-02-24'));
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 die Aufrufvariante mit Zuweisungsoperator auch in SQL verwenden, aber ein solcher Aufruf funktioniert nicht mit call. Der Grund: Ginge dies, wäre die Funktion bei Oracle anders implementiert als im ISO-Standard. Da andererseits diese Funktion lediglich aus Gründen der Kompatibilität mit dem 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 '2023-01-24', 30);
-- explizit, weil nur ein Parameter verwendet wurde:
print_return_date(
  p_start_date => to_date('24.01.2023', '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

Umgekehrt 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 Ihren 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. Was auch wieder nicht stimmt, 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 19, »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 byte);
/
```

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 1 .. p_char_list.count 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 werde ich 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 hatte ich 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
----------
20.02.2023
```

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 Sie 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: 20.02.23
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 `5` ff.) 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: Sie 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 antreffen können: den Triggern und den Packages.

Inhalt

TEIL I Grundlagen

3 Aufbau der Datenbank aus Sicht eines Programmierers 57

TEIL II Die Sprache PL/SQL

7 Die Blockstruktur und Syntax von PL/SQL 233

Materialien zum Buch

Auf der Webseite zu diesem Buch stehen folgende Materialien für Sie zum Download bereit:

- **alle Beispielprogramme**

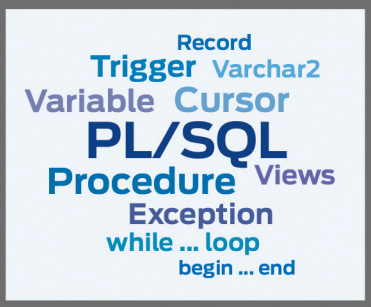
Gehen Sie auf *www.rheinwerk-verlag.de/5739* und klicken Sie dort auf den Reiter MATERIALIEN. Sie sehen die herunterladbaren Dateien samt einer Kurzbeschreibung des Dateiinhalts. Klicken Sie auf den Button HERUNTERLADEN, um den Download zu starten. Je nach Größe der Datei (und Ihrer Internetverbindung) kann es einige Zeit dauern, bis der Download abgeschlossen ist.

Oracle PL/SQL

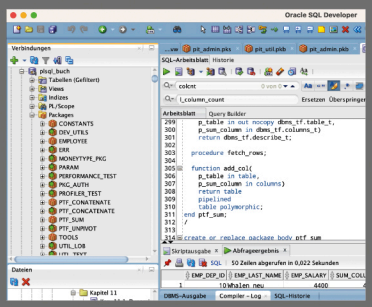
» Ein außergewöhnlicher Erfahrungsschatz «
DOAG

Holen Sie das Maximum aus Ihrer Datenbank heraus!

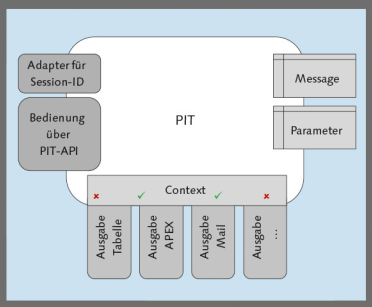
Sie sind Anwendungsentwickler und möchten Ihre Datenbankfunktionen mit PL/SQL erweitern? Hier finden Sie alles, was Sie für die Praxis brauchen, perfekt aufbereitet von Oracle-Profi Jürgen Sieben: ein strukturierter Einstieg mit allen Grundlagen der Oracle-Datenbank sowie Techniken für Fortgeschrittene, inklusive Workshops.



Grundlagen von PL/SQL



Sprachkonstrukte und Konzepte



Fortgeschrittene Techniken

Die Oracle-Datenbank grundlegend verstehen

Jürgen Sieben erklärt Ihnen Aufbau und Arbeitsweise der Datenbank bis ins Detail. Sie erfahren alles über Speicherstrukturen, Parameter, Passwortdateien, Schemata, Tablespaces, Datenbankobjekte und mehr.

PL/SQL in der Praxis einsetzen

Erweitern Sie den Befehlsumfang von SQL durch eigene PL/SQL-Funktionen und erfahren Sie mehr über die Arbeit mit großen Datenstrukturen. Von einfachen Beispielen führt Sie Jürgen Sieben zu komplexen, sofort einsetzbaren Anwendungen.

Professionell mit JSON, XML und OOP arbeiten

Ob Performance-Tuning, Fehlerbehandlung, objektorientierte Programmierung, JSON oder XML – mit diesem umfassenden Handbuch können Sie vielschichtige Aufgabenstellungen mit PL/SQL lösen. So werden Sie zum Datenbank-Profi!

Beispielmaterialien zum Download

Jürgen Sieben ist ausgewiesener Oracle-Experte. Als Entwickler, Schulungsleiter und Hochschuldozent beherrscht er alle relevanten Fachgebiete, von PL/SQL über Performance-Optimierung bis hin zur Administration.

Aus dem Inhalt

Rheinwerk Computing

€ 89,90 [D] € 92,50 [A]

Gedruckt in Deutschland
Papier aus nachhaltiger Waldwirtschaft
Mineralölfreie Druckfarben

Für Windows und Linux
Netzwerke & Datenbanken
ISBN 978-3-8362-9630-4