Kapitel 1
Einführung

Dieses Buch stellt Ihren Begleiter für die ersten Schritte mit der Oracle-Datenbank dar. Wir werden gemeinsam Aufbau und Arbeitsweise der Datenbank erkunden und die Sprache SQL erlernen, mit deren Hilfe wir die Daten der Datenbank für uns verfügbar machen.

Also, Oracle und SQL sollen es sein. Warum SQL, warum Oracle? Eine Frage, die Sie sich möglicherweise noch nicht gestellt haben oder deren Beantwortung aus Ihrer Situation heraus trivial erscheint. Ich habe sie mir gestellt, um eine Idee davon zu bekommen, aus welchem Grund Sie sich für dieses Buch interessieren oder es kaufen möchten. In meinen Kursen schildern die Teilnehmer häufig ähnliche Gründe für ihre Teilnahme: Da sind die Mitarbeiter der Fachabteilungen, die für Berichte schon SQL genutzt haben und ihr Wissen strukturieren und vertiefen möchten. Da sind die Anwendungsentwickler, die Kenntnisse in Programmiersprachen haben und genauer wissen möchten, wie die Datenbank tickt. Da sind die Umsteiger, die andere Datenbanksysteme kennen und überrascht waren, dass einige Dinge bei Oracle anders funktionieren. Und da sind schließlich die Mitarbeiter, die Projekte anderer Kollegen übernommen haben und vor einer Menge Abfragen stehen, die sie nicht verstehen.

1.1 Für wen ist dieses Buch geschrieben?

Egal, aus welchem Grund Sie sich für SQL interessieren, für mich ist entscheidend, dass möglichst viele von Ihnen von diesem Buch profitieren. SQL ist eine Abfragesprache, mit der Sie Daten aus einer Datenbank ermitteln, anlegen oder ändern können. SQL ist jedoch vor allem eine Jahre währende Beschäftigung mit einer extrem mächtigen und vielseitigen Programmiersprache. Mein Buch muss daher nicht nur beim ersten Lesen interessant sein, sondern soll darüber hinaus Ihren Weg mit SQL ein Stück begleiten können. Um beides zu vereinen, habe ich mich zu einem spiralförmigen Ansatz entschlossen: Wir beginnen damit, das Gebiet aus größerer Höhe zu umkreisen, und ignorieren dabei noch Details und Hinterhältigkeiten. Später bespreche ich die Gebiete ein zweites Mal, zum Teil in anderem Zusammenhang, um die gesamte Bandbreite von SQL in Oracle zu zeigen. Die Kapitel des ersten Teils

bauen aufeinander auf, ich empfehle, sie in der vorgegebenen Reihenfolge zu lesen. Einige Kapitel können Sie auch später lesen, wenn Sie ein anderes Thema mehr interessiert. Falls dies möglich ist, weise ich zu Beginn der Kapitel darauf hin.

Ich gehe davon aus, dass Sie dieses Buch komplett im Selbststudium oder begleitend zu einem Einstiegskurs in SQL benutzen. Der erste Weg ist für viele der härtere, also muss ich das zum Teil komplexe Thema möglichst nachvollziehbar darstellen. Einzelne Kapitel, bei denen mir dies sinnvoll erschien, enden mit einem Übungsblock, um Ihnen die Chance zu geben, mit Textaufgaben SQL zu üben. Doch auch wenn Ihnen über die ersten Hürden in einer Schulung hinweggeholfen wird, werden Sie SQL in wenigen Tagen nicht erlernen können. Sie werden sich über die Grundlagen hinaus entwickeln und schwierigere Themen erobern wollen. Hierfür biete ich Ihnen genügend Material zur Vertiefung Ihres Wissens an.

Ich finde es wichtiger, zu erklären, *warum* etwas getan werden muss, als *wie* etwas getan werden muss. Anders gesagt: Dieses Buch ist nur zum Teil eine Referenz zu SQL, in der Sie im Index einen Befehl nachschlagen können und alle Optionen und Verwendungsmöglichkeiten aufgelistet bekommen. Das übernehmen die Online-Ressourcen, die bei Oracle auf einem hohen Niveau sind, wesentlich besser. Eine SQL-Referenz ist naturgemäß von der Version der Datenbank abhängig und wäre in Buchform veraltet, bevor sie ausgeliefert wird. Ich gebe der Erläuterung des Zusammenhangs den Vorzug vor der Auflistung von Funktionen. Ich möchte den zur Verfügung stehenden Platz nutzen, um Ihnen ein Verständnis der Denkweise und Möglichkeiten von SQL zu geben. Die Online-Dokumentation sorgt begleitend für den syntaktischen Unterbau.

1.2 Aufbau des Buches

Das Buch ist in mehrere Teile untergliedert, dem Gedanken folgend, dass ich zunächst die Grundlagen sowohl der Datenbank als auch der Abfragesprache SQL besprechen möchte. Danach folgt ein Teil, der sich mit der Anwendung von SQL in konkreten Einsatzszenarien auseinandersetzt und weiter gehende technologische Konzepte erläutert. Zu ausgewählten Kapiteln (zu den einführenden insbesondere) biete ich darüber hinaus Übungen an, deren Lösung Sie mit einer kurzen Darstellung der Strategie online auf *www.rheinwerk-verlag.de/4627* finden.

1.2.1 Teil I – Einführung und Grundlagen

Im ersten Teil des Buches werde ich das nötige Vorwissen erläutern, das Sie benötigen, um SQL zu erlernen. Diese Kapitel haben daher noch nichts mit SQL direkt zu tun, bereiten aber die Basis, sowohl technisch als auch vom Verständnis her.

Kapitel 2 – Verwendete Werkzeuge und Skripte

In diesem Kapitel beschreibe ich, wie eine Oracle-Datenbank installiert und konfiguriert wird. Die Beschreibung ist so gehalten, dass Sie eine Datenbank einrichten können, die Sie für die Beispiele des Buches benötigen. Zudem erläutere ich das Programm SQL Developer, mit dem wir in diesem Buch die SQL-Anweisungen erstellen werden.

Kapitel 3 – Konzept einer relationalebn Datenbank

Dieses Kapitel erläutert, was eine Datenbank ausmacht und welche Anforderungen an solche Systeme gestellt werden. Wir werden untersuchen, warum es sinnvoll ist, Daten auf Tabellen zu verteilen, und welche grundlegenden Regeln hierbei beachtet werden müssen. Zudem werde ich Ihnen die – überraschend einfachen – Spielregeln für relationale Datenbanken erläutern. Das Kapitel führt aber auch in SQL ein und erläutert, woher diese Sprache kommt und was man damit machen kann. Schließlich können Sie Ihr Wissen an einem bestehenden Datenmodell des Benutzers `HR` ausprobieren, um zu verstehen, auf welche Weise Datenbanken modelliert werden.

1.2.2 Teil II – Die SELECT-Anweisung

Der zweite Teil des Buches befasst sich mit den Grundlagen der Sprache SQL sowie mit der Syntax des wichtigsten SQL-Befehls, der `select`-Anweisung, mit deren Hilfe Sie Daten der Datenbank lesen und Auswertungen erstellen können. Alle Kapitel des zweiten Teils enden mit einer kleinen Gruppe von Aufgaben, mit deren Hilfe Sie im Selbststudium Ihr Wissen prüfen können.

Kapitel 4 – Grundlagen: Auswahl und Projektion

Mit diesem Kapitel beginnen wir die Beschäftigung mit der Sprache SQL. Sie werden einfache SQL-Anweisungen schreiben und verstehen. Hier legen wir die syntaktischen Grundlagen, überlegen, wie einzelne Spalten und Zeilen ausgewählt werden können, und beginnen damit, einfache Rechnungen und Operationen an den Daten für eine Auswertung vorzunehmen.

Neben diesen Kernfunktionen werden Sie aber auch bereits leistungsfähigere Fallunterscheidungen anwenden und Pseudospalten, Schlüsselwerte und speziellere Werte, wie etwa den `null`-Wert, kennenlernen. Gerade dieses letzte Thema wird uns bereits hier in logische Randbereiche führen, die bei der Beschäftigung mit Datenbanken allgegenwärtig sind.

Kapitel 5 – Daten aus mehreren Tabellen lesen: Joins

In diesem Kapitel werden wir die Möglichkeiten, die wir in SQL haben, erweitern, indem wir Daten aus mehreren Tabellen abfragen. Mit Hilfe dieser Fähigkeiten ent-

stehen leistungsfähige Berichte, die für die Arbeit mit Datenbanken unerlässlich sind. Das Mittel hierfür sind die sogenannten Joins, deren verschiedene Varianten in diesem Kapitel besprochen werden. In dieses Kapitel fällt aber auch die Verwendung der Mengenoperationen, die – ähnlich einem Join – Daten aus verschiedenen Tabellen kombinieren.

Kapitel 6 – Zeilenfunktionen

Dieses Kapitel erweitert die Kenntnis von SQL um Zeilenfunktionen, die es ermöglichen, die Daten der Tabelle für einen Bericht aufzuarbeiten, zu ändern oder anders darzustellen. Diese Funktionen werden sehr häufig im Berichtswesen eingesetzt, stellen aber gleichzeitig auch den ersten Bereich dar, in dem sich Datenbanken verschiedener Hersteller voneinander unterscheiden, denn nicht alle Funktionen haben standardisierte Bezeichner.

Das Kapitel dient, im Sinne eines ersten Herangehens an diese Funktionen, als Überblickskapitel, das die Zeilenfunktionen so darstellt, dass der besprochene Funktionsumfang für 90 % der Anweisungen ausreicht. Speziellere Optionen werden dann in späteren Kapiteln besprochen. Die Zeilenfunktionen dieses Kapitels werden in Datums-, Text-, mathematische und allgemeine Funktionen unterteilt. Den Abschluss bildet ein kurzes Beispiel zur Programmierung eigener Funktionen mittels der Programmiersprache PL/SQL.

Kapitel 7 – Gruppenfunktionen

Eine weitere Stufe auf der Komplexitätsleiter stellen die Gruppenfunktionen dar, mit deren Hilfe aus Daten einer Tabelle leistungsfähige Berichte erstellt werden. Wir starten in diesem Kapitel mit den Grundfunktionen zur Summierung, Durchschnittsbildung, zu Maximal- bzw. Minimalfunktionen etc. Doch werden auch weiter gehende Konzepte der Gruppenfunktionen besprochen, wie etwa die Gruppierung oder das Filtern von Gruppenfunktionen. Ein Überblick über spezielle Gruppenfunktionen rundet das Kapitel ab.

Kapitel 8 – Unterabfragen

Dieses Kapitel erweitert Ihre Kenntnis über SQL durch Unterabfragen, mit deren Hilfe Hilfsabfragen berechnet werden können, um mit deren Ergebnissen die eigentliche Abfrage beantworten zu können. Die Vermittlung der Fähigkeit, erkennen zu können, wann eine Unterabfrage erforderlich ist, wird der zentrale Schwerpunkt dieses Kapitels sein. Dabei betrachten wir die verschiedenen Formen der Unterabfrage, die skalare, die harmonisierte, aber auch Unterabfragen mit mehreren Zeilen und/oder mehreren Spalten. Zudem werden Sie Unterabfragen in den unterschiedlichsten Klauseln der SQL-Anweisung und die `with`-Klausel kennenlernen.

1.2.3 Teil III – Datenmanipulation und Erzeugung von Datenbankobjekten

Während sich Teil II mit der Syntax und den verschiedenen Formen der `select`-Anweisung beschäftigt, mit deren Hilfe Daten aus einer Tabelle ausgelesen werden können, werden Sie in Teil III Ihre Kenntnis von SQL durch Anweisungen erweitern, die es Ihnen gestatten, Daten innerhalb der Datenbank zu manipulieren. Zudem sehen wir uns an, auf welche Weise Datenbankobjekte wie Tabellen oder Views erstellt werden.

Kapitel 9 – Datenmanipulation

Den Anfang macht ein Kapitel über die Anweisungen zum Einfügen, Ändern und Löschen von Daten. Neben diesen »klassischen« Anweisungen lernen Sie aber auch die `merge`-Anweisung kennen, die sehr leistungsfähig ist und für viele Arbeiten eingesetzt werden kann. Sie werden erkennen, dass Sie sehr vom Wissen profitieren, das Sie sich im zweiten Teil des Buches erarbeitet haben, so dass Sie hier zügig vorankommen werden. Einen gewichtigen Teil dieses Kapitels nimmt aber auch die Diskussion des Transaktionsbegriffs ein, denn dieser Begriff ist für das Verständnis von Datenbanken zentral. Nun werden Sie, nach der allgemeinen Einführung in den Teilen I und II, diesen Begriff konkret im Einsatz sehen. Schließlich zeige ich Ihnen noch, wie Sie mit Fehlern bei der Manipulation sinnvoll umgehen.

Kapitel 10 – Views erstellen

Dieses Kapitel führt in die Arbeit mit Views ein. Diese Datenbankobjekte werde ich über den grünen Klee loben, denn für mich sind Views eines der wichtigsten Hilfsmittel bei der Arbeit mit Datenbanken. Ich werde erläutern, woher meine Begeisterung für Views kommt und wie sie verwendet werden können. Wir werden dabei sowohl einfache als auch komplexe Views besprechen und auch ihre Cousins, die materialisierten Views, darstellen. Eine Diskussion der möglichen Einsatzbereiche rundet das Verständnis ab.

Kapitel 11 – Tabellen erstellen

Konsequenterweise muss natürlich auch das Erstellen von Tabellen besprochen werden. Im Gegensatz zur Erstellung von Views ist bei der Erstellung einer Tabelle jedoch fast immer eine grafische Oberfläche beteiligt, wie in unserem Fall der SQL Developer. Da es keinen Vorteil bringt, die Anweisung händisch zu formulieren, und da die grafischen Werkzeuge stets auch die resultierenden SQL-Anweisungen darstellen können, ist es nicht sinnvoll, jedes syntaktische Detail der Erstellung von Tabellen zu besprechen, zumal dieses Thema sehr stark in Richtung Datenbankadministration abwandert. Wichtiger ist mir in diesem Kapitel daher die Darstellung der verschiedenen Tabellentypen wie der indexorganisierten Tabelle oder der tem-

porären Tabelle, deren Einsatzbereiche ich erläutern werde. Zudem führe ich die aktive Tabelle ein und erläutere die Ideen hinter diesem Konstrukt; daher fällt auch ein kurzer Exkurs zum Thema Trigger in dieses Kapitel.

Kapitel 12 – Indizes erstellen

Wohl kaum ein Thema der Datenbanken wird so kontrovers und leider auch falsch diskutiert wie die Indizierung. Den einen gilt die Indizierung als zentrales Performanz-Tuning-Thema, andere denken ausschließlich an den Aufwand, der für Indizierung betrieben werden muss. Dieses Kapitel erläutert das Prinzip der Indizierung und ordnet Indizes als Bestandteil einer Strategie zur Optimierung der Antwortzeiten ein. Zudem erläutere ich aber auch speziellere Indextypen, die für das einsteigende Verständnis nicht erforderlich sind, im weiteren Verlauf aber recht wichtig werden können. Zu diesen Indextypen gehören Bitmap- sowie Domänenindizes.

Kapitel 13 – Aufbau einer Oracle-Datenbank

Den Abschluss des Teils bildet ein Kapitel, das Ihnen einen Überblick über die Arbeitsweise der Oracle-Datenbank gibt. Zudem erläutere ich die Datentypen, die Oracle für die Verwendung in Tabellen bereitstellt.

1.2.4 Teil IV – Spezielle Abfragetechniken

Dieser vierte Teil wurde erforderlich, um ein Problem aufzulösen, das ansonsten nur sehr schwer zu lösen ist: Viele Anweisungen enthalten hochspezialisierte Optionen, zum Beispiel aus dem Bereich der Internationalisierung, deren Anwendung beim ersten Erläutern schlicht zu detailliert würde. Um dieses Problem zu umgehen, werden in diesem Teil Abfragetechniken und die aus dem Blickwinkel einer Abfragestrategie eingesetzten Werkzeuge erläutert. Spätestens ab diesem Teil ist dieses Buch nicht mehr für Einsteiger geeignet, sondern dient dem fortgeschrittenen SQL-Anwender als Fundgrube für Problemlösungsstrategien und speziellere Optionen.

Kapitel 14 – Analytische Funktionen

Analytische Funktionen sind für viele, die bereits mit SQL arbeiten, eine Offenbarung, weil sie komplexe Fragestellungen einfacher und performanter lösen können als herkömmliche SQL-Strategien. Dieses Kapitel bespricht diesen Typ Funktion, dabei widmen wir uns der Partitionierung, Sortierung und Filterung über Fensterfunktionen, die für diese Gruppe von Funktionen typisch sind. Schließlich werden die analytischen Funktionen, die nur als solche existieren, besprochen und in Anwendungsszenarien gezeigt.

Kapitel 15 – Hierarchische Abfragen

Hierarchische Abfragen belasten SQL bis an die Grenzen seiner Ausdrucksfähigkeit. Da aber parallel die Speicherung hierarchisch organisierter Daten in Datenbanken allgegenwärtig ist, liefert Oracle bereits seit vielen Jahren eine proprietäre Erweiterung für dieses Problem mit. Erst mit der Version 11gR2 ist zudem eine ISO-kompatible Methode der Beantwortung solcher Fragestellungen hinzugekommen.

Kapitel 16 – Arbeiten mit XML

Das Thema XML hat seit Version 9 der Datenbank in jedem neuen Release an Bedeutung gewonnen. Mittlerweile stellt sich eine Oracle-Datenbank wahlweise als relationale oder als XML-Datenbank dar. Dieses Kapitel führt in den Standard SQL/XML ein, beschreibt den Datentyp `XMLType` der Oracle-Datenbank und diskutiert einige einfache Beispiele in der XML-Abfragesprache XQuery, die vollständig in der Oracle-Datenbank implementiert ist. Ein kurzer Ausblick beschäftigt sich mit Techniken zur Indizierung von XML. Aufgrund des Umfangs des Themas kann jedoch lediglich eine Einführung in die Thematik gegeben werden, keine umfassende Diskussion.

Kapitel 17 – JSON

Seit Version 12c unterstützt die Oracle-Datenbank den Datentyp JSON, der insbesondere in der Webentwicklung eingesetzt wird. Die Datenbankversion 12.2 erweitert die Unterstützung, so dass die Einrichtung eines eigenen Kapitels zu diesem Thema sinnvoll erschien. Allerdings komme ich nicht umhin, zum derzeitigen Implementierungsstand kritische Anmerkungen zu machen, daher ist dieses Kapitel eher eine Bestandsaufnahme des aktuellen Standes mit der Hoffnung, dass sich hier zukünftig noch einiges verbessert.

Kapitel 18 – Pivotieren von Daten

Unter der Pivotierung von Daten versteht man das Vertauschen von Spalten und Zeilen eines Berichts. Dieses Thema beherrschen spezielle Anwendungssteuerelemente, doch innerhalb von SQL ist dies eine eher schwierige Übung. Zum einen wird dieses Kapitel eine Do-it-yourself-Methode vorstellen, die von allen Datenbanken beherrscht und seit vielen Jahren eingesetzt wird. Zum anderen stelle ich die mit der Datenbankversion 11g eingeführte neue `pivot`-Klausel vor, mit deren Hilfe dieser Abfragetyp einfacher und zum Teil auch leistungsfähiger umgesetzt werden kann.

Kapitel 19 – Row Pattern Matching

Ebenfalls eine Neuerung der Version 12c der Oracle-Datenbank ist die Fähigkeit der Analyse von Mustern in Datenmengen. Bislang gab es die Unterstützung regulärer Ausdrücke, um in Texten Muster zu erkennen, doch keine Möglichkeit, in vielen Zei-

len einer Tabelle Muster zu erkennen. Da wir bislang ohne diese Möglichkeiten ausgekommen sind, scheint die Erweiterung nicht ein dringendes Bedürfnis zu befriedigen. Ich werde mich in diesem Kapitel aber bemühen, zu erklären, dass dies sehr wohl so ist. An einigen Beispielen zeige ich die zunächst umfangreiche Syntax auf und versuche, eine Lanze für die Verwendung dieser neuen Funktionalität zu brechen.

Kapitel 20 – Die MODEL-Klausel

Seit Version 10g der Oracle-Datenbank verfügt SQL über einen mächtigen Mechanismus, um Daten aus bestehenden Daten abzuleiten und neu zu berechnen. Für diese Anwendungsbereiche, die normalerweise einer Tabellenkalkulation vorbehalten waren, liefert Oracle mit der `model`-Klausel ein weitgehend vollständiges Instrumentarium zur Kalkulation solcher Werte mit. Ungeachtet der vergleichsweise geringen Kenntnis dieser Funktionen in weiten Teilen der SQL-Anwendergemeinde schlägt dieses Kapitel eine Bresche für das Thema und zeigt Einsatzbereich, Syntax und Vorteile dieser Strategie.

Kapitel 21 – Umgang mit Datum und Zeit

Es mag Sie zunächst überraschen, dass in diesem Teil noch ein Kapitel über den Umgang mit Datum und Zeit erforderlich ist. Der Grund liegt in der Berücksichtigung verschiedener Zeitzonen, der Probleme internationalisierter Datenmodelle und nicht zuletzt in der Diskussion über das Für und Wider der ISO-konformen versus der Oracle-konformen Implementierung von Datumsfunktionen. Dieses Kapitel ist also definitiv interessant für alle, die mit Datum und Zeit auf hohem Niveau arbeiten müssen. Das Kapitel bespricht alle Optionen der Erzeugung und Konvertierung von Datumsformaten, auch im multikulturellen Kontext, inklusive und exklusive Zeitzonen und zeigt auf, welche Zeitzonen unterstützt werden und wo dies nachgeschlagen werden kann. Wir beschäftigen uns noch einmal mit dem Intervall, ich zeige die Grenzen der Algebra mit Intervallen auf und vieles mehr. Ein weiterer Schwerpunkt dieses Kapitels ist die Einführung der Flashback-Abfrage, die uns die Entwicklung von Daten über die Zeit darstellen hilft.

Kapitel 22 – Objektorientierung in der Oracle-Datenbank

Dieses Kapitel betrachtet die objektrelationalen Fähigkeiten der Oracle-Datenbank, soweit sie aus dem Blickwinkel von SQL von Interesse sind. Wir hören also dort auf, wo die Programmierung mit diesen Typen beginnt. Neben einer Einführung in die Ideen der Objektorientierung steht hier die Arbeit mit SQL-Typen, Varrays und Nested Tables im Mittelpunkt. Ich werde Vor- und (vor allem) Nachteile der objektrelationalen Speicherung mittels objektorientierter Tabellen diskutieren, aber auch Wege aufzeigen, wie die Fähigkeiten dieses Bereichs sinnvoll eingesetzt werden können, zum Beispiel im Zusammenhang mit objektrelationalen Views.

Kapitel 23 – Datenwarenhaus

Demjenigen, der Datenwarenhäuser kennt, ist klar: So ein Thema kann nicht in einem Kapitel erläutert werden. Mir geht es darum, in diesem Kapitel die Grundlagen eines typischen Datenwarenhauses darzustellen und die Unterschiede zu »normalen« Datenbanken herauszuarbeiten. Natürlich wird der Begriff des *Star-Schemas* hier eine Rolle spielen, aber mir geht es auch um das Problem, das durch diese Modellierung gelöst werden soll. Seit Version 12.2 der Datenbank sind als wichtiges Thema die analytischen Views hinzugekommen, die sich anschicken, die Implementierung eines Datenwarenhauses auf gänzlich neue Füße zu stellen.

Kapitel 24 – Performanzoptimierung von SQL

Es gibt dicke und unendlich komplizierte Bücher zu diesem Thema, daher werde ich mir nicht anmaßen, in einem Kapitel alles Wissenswerte zum Thema sagen zu können. Ich finde aber, dass es sehr wichtig ist, zumindest die Best Practices zu beherrschen, um einigermaßen sicher zu sein, jedenfalls keine groben Fehler bei der Erstellung von Abfragen zu machen. Das Kapitel trägt diese Best Practices zusammen und erweitert sie um Anregungen zum Thema Indizierung, Vermeidung von Umgebungswechseln und um einige einfache Anmerkungen zum Lesen von Ausführungsplänen.

1.2.5 Teil V – Datenbankmodellierung

Der abschließende Teil des Buches kommt bei den meisten anderen Büchern über SQL eigentlich als Erstes: Hier geht es um die Modellierung von Datenbanken. Ich habe dieses Thema bewusst an das Ende des Buches gestellt, denn einerseits benötigt ein großer Teil der Anwender von SQL dieses Wissen nicht, da sie ohnehin nur mit bestehenden Datenmodellen arbeiten. Zum anderen, und das ist das aus meiner Sicht das größere Problem, kommt diese Diskussion einfach viel zu früh. Ein Einsteiger in SQL kämpft mit den Grundlagen und hat daher einfach noch nicht den Überblick, sich um Feinheiten der Modellierung zu kümmern. Dieser Teil ist allerdings auch keine vollständige Darstellung dieses Problemfeldes, sondern versucht, sozusagen »aus der Praxis für die Praxis«, einige wichtige Strategien zu erläutern, ohne das Thema durch allzu viel Theorie zu überladen.

Kapitel 25 – Die Grundlagen der Datenmodellierung

Vielleicht schwer zu glauben, aber wahr: In diesem Kapitel spielen Normalisierungsregeln eine eher untergeordnete Rolle. Ich werde zwar auch erläutern, warum Normalisierungsregeln verwendet werden, sortiere sie allerdings eher in die Kategorie »Hilfsmittel« ein, um ein gutes Datenmodell zu verifizieren. Wichtiger ist mir in diesem Kapitel, übliche Strategien zur Speicherung von Daten in Tabellen zu finden und

aufzuzeigen. Zentrale Fragen sind dabei: Wie gehen wir mit Primär- und Fremdschlüsseln um, wie mit wiederkehrenden Spalten, die etwa das Anlage- oder letzte Änderungsdatum zeigen sollen? Fragen der Indizierung, die sich unmittelbar aus dem Datenmodell ergeben, werden ebenso behandelt wie Konventionen und Überlegungen zur Wahl der korrekten Datentypen. Ich werde einige Namenskonventionen vorstellen, die ich in Projekten als angenehm empfunden habe, ohne Sie allerdings als »Anfänger« abstempeln zu wollen, wenn Sie eine andere Strategie wählen.

Kapitel 26 – Datenmodellierung von Datum und Zeit

Und noch ein Kapitel zum Thema Datum und Zeit! Dieses Kapitel beschäftigt sich mit diesem Komplex aus Sicht der Datenmodellierung: Wie werden Datumsbereiche gespeichert, was verbirgt sich hinter dem Datentyp `WM_PERIOD`, und welche Vorteile bietet es, sich bei Datenwarenhäusern eine Zeitdimension auszuleihen? Ein weiterer, wichtiger Bereich dieses Kapitels sind Strategien zum Logging von Daten sowie zur Historisierung, wo wir uns historisierende und bitemporale Datenmodelle ansehen werden sowie das *Information Lifecycle Management*, das Oracle verwendet, um zeitbezogene Daten zu verwalten.

Kapitel 27 – Speicherung hierarchischer Daten

Auch dieses Kapitel hat im vorigen Teil schon eine Einführung durch die hierarchischen Abfragestrategien erhalten. Nun geht es um die verschiedenen Modellierungstechniken zur Speicherung hierarchischer Daten. In diesem Kapitel werden wir uns eine Erweiterung der Speicherung hierarchischer Daten durch eine ausgelagerte Hierarchietabelle ansehen, aber auch Ideen wie etwa Closure Tables und andere mehr.

Kapitel 28 – Abbildung objektorientierter Strukturen

Ebenfalls als ergänzendes Thema zu Kapitel 22, »Objektorientierung in der Oracle-Datenbank«, ist dieses Kapitel gedacht. Es geht im Kern um das Problem, auf welche Weise Tabellen gestaltet werden können, um Objekte einer Anwendung aufnehmen zu können. Die Kernprobleme stellen dabei das Konzept der Vererbung einerseits und die Behandlung von Kollektionen andererseits dar, denn diese fundamental anders implementierten Zusammenhänge lassen sich nicht ohne Probleme aufeinander abbilden.

Kapitel 29 – Internationalisierung

In diesem abschließenden Kapitel gehe ich der Frage nach, welche Auswirkung eine internationalisierbare Anwendung auf die Speicherung der Daten in der Datenbank hat. Die zentrale Fragestellung lautet hier, welche Strategien zur Speicherung übersetzbarer Daten existieren, denn viele der anderen Probleme (Datumsformate, Sortierungen etc.) sind bereits durch die Datenbank gelöst. In diesem Kapitel werden wir

auf das Thema Zeichensatzcodierung zu sprechen kommen, der wir viele Probleme zu verdanken haben, aber Sie werden auch hinterhältige Datumsformate kennenlernen. Der Schwerpunkt liegt jedoch auf Überlegungen zu Datenmodellen, mit deren Hilfe Stammdaten übersetzbar gespeichert werden können, sowie deren Auswirkungen auf die referenzielle Integrität der Datenbank.

1.3 Anmerkung zur dritten Auflage

Mit dieser dritten Auflage möchte ich der Bitte eines Lesers nachkommen und kurz anreißen, was sich seit der letzten Auflage geändert hat. Dies ist sinnvoll, weil offensichtlich mehrere Auflagen dieses Buches bei meinen Lesern ankommen, wofür ich ausdrücklich danken möchte, damit hätte ich nicht gerechnet.

Die dritte Auflage ist in bewegten Zeiten entstanden, als die Datenbankversion 12.2 veröffentlicht wurde und Oracle die Entscheidung traf, nun auf einen jährlichen Veröffentlichungszyklus umzuschwenken, womit die nächste Datenbankversion den Namen 18c tragen wird. Der Sprung in der Versionsnummer wird funktional aber weniger bedeutend als bei vorhergehenden Major-Releases, daher denke ich, dass Sie auch weiterhin mit dieser Auflage Sinnvolles zu Oracle SQL lernen können.

Natürlich sind die Neuerungen der Version 12.2 in die Datenbank übernommen worden. Neben Änderungen bezüglich der Länge der Bezeichner sind die hervorstechenden Erweiterungen und Neuerungen die analytischen Views, die im Datenwarenhausbereich in der Zukunft wohl einige Beachtung erhalten werden. Der Ausbau der Unterstützung des Datentyps JSON ist ein weiterer Schwerpunkt, auch wenn ich bei meiner kritischen Haltung zur Implementierung dieser Unterstützung bleibe. Hier ist noch vieles offen, die Hoffnung auf Besserung in den nächsten Releases bleibt aber ebenso bestehen.

Ansonsten habe ich in dieser Auflage die Anordnung der Kapitel überarbeitet, so dass ich hoffe, einen noch klareren Aufbau des Buches erreicht zu haben. Relativ viel Zeit ist auch in die Integration all der kleinen Neuerungen der Version 12.2 in den Text geflossen, alle Codebeispiele wurden überarbeitet und, wo sinnvoll, durch weitere Beispiel verbessert. Natürlich sind auch neue Erlebnisse in Projekten und besonders erwähnenswerte Lösungen in das Buch aufgenommen worden. Dann habe ich mich entschlossen, allzu offensichtliche Hinweise auf Eigenarten der Version 10.2 zu entfernen, einfach, weil ich glaube, dass die Zeit über diese Datenbank hinweggegangen ist.

Ebenfalls auf Kritik eines Lesers hin habe ich mich bemüht, die Indizierung des Buches zu verbessern. Das ist – unter uns gesagt – eine etwas unerfreuliche Arbeit, ich hoffe aber, dass das Ergebnis etwas besser nutzbar geworden ist.

Ähnlich wie bei Oracle SQL generell, ist diese dritte Auflage also eher eine Evolution denn eine Revolution, aber in Summe ist doch so gut wie kein Stein auf dem anderen geblieben. Ich hoffe, dass Sie auch diese Auflage gern lesen und wünsche Ihnen bei Ihren Bemühungen um SQL größtmöglichen Erfolg.

1.4 Anmerkung zur zweiten Auflage

Eine zweite Auflage eines Buches ist vor allem einmal Grund, Ihnen, den Lesern, für Ihr Vertrauen zu danken. Ich habe mich bemüht, diese Auflage noch besser als die erste zu machen, und hierzu habe ich natürlich zum einen die Neuerungen der Version 12c integriert, habe herausgeworfen, was schon in Version 11g als veraltet gekennzeichnet war, und insgesamt die Skripte daraufhin durchgesehen, ob sie noch standardkonformer, besser und eleganter geschrieben werden können.

Zum anderen sind aber auch Anregungen und Kritik der Leser in den Text eingeflossen, so zum Beispiel zum XML-Kapitel, das nun kurze Ausflüge zu den Themen XPath und Namensräume enthält, um auch den Lesern, die nicht mit XML gearbeitet haben, einen Einstieg zu ermöglichen. Einen besonderen Dank möchte ich an Anja Uhlig richten, die das neue XML-Kapitel und das Row-Pattern-Matching-Kapitel gelesen und kritisch kommentiert hat. Ich habe die Reihenfolge einiger Kapitel des dritten Teils geändert sowie zwei weitere Kapitel aufgenommen. Neu ist Kapitel 19, »Row Pattern Matching«, einfach weil die ganze Abfragetechnik neu ist in Version 12c, zum anderen habe ich aber auch ein Kapitel zu den Grundlagen des Performanz-Tunings von SQL eingefügt (Kapitel 24), von dem ich stark annehme, dass es bei Ihnen auf ein besonderes Interesse stoßen wird. Das Kapitel legt natürlich nur den Grundstein, diesen aber so, dass Sie mit ziemlicher Sicherheit bereits sehr gutes SQL schreiben werden, wenn Sie sich an die Regeln und Empfehlungen dieses Kapitels halten.

Im Großen und Ganzen war ich hocherfreut und ein wenig beschämt von der sehr guten Kritik, die mir zur ersten Auflage dieses Buches entgegenschallte, und hoffe, dass auch die Neufassung Anlass zu ähnlich positiver Bewertung gibt. Offensichtlich hat man mir meine manchmal etwas wenig »fachbuchtypische«, eher lockere Art zu schreiben nicht übel genommen, was mich freut, weil ich fest davon ausgehe, dass es nicht in erster Linie um höchstmögliche logische Unangreifbarkeit des Autors geht, sondern ausschließlich darum, Ihnen das Problem SQL so einfach und andererseits so umfassend wie möglich zu erläutern. Ich hoffe in diesem Sinne, dass es Ihnen mit dieser Auflage noch leichter fällt, sich auf die faszinierende Reise ins SQL-Land einzulassen.

1.5 Danksagung

Diesmal hat meine Frau angemerkt, dass ich mir nun nicht einbilden solle, jedes Jahr ein Buch schreiben zu wollen. Da wären durchaus noch andere Sachen zu tun, sie hätte da schon eine ganze Liste im Hinterkopf. Doch andererseits hat meine Frau auch als Testleserin der ersten Kapitel maßgeblichen Anteil am Gelingen des Buches (ich hoffe, dass Sie den wohltuenden Einfluss durchaus bemerken) … Daher gilt mein Dank meiner Frau, einerseits für das Verständnis, dass nicht alle Möbel gebaut werden konnten, und andererseits für die Mithilfe und konstruktive Kritik am Buch.

Mein Dank gilt darüber hinaus den Fachlektoren und Herrn Mattescheck vom Rheinwerk Verlag, der dieses Projekt begleitet hat.

Nicht zuletzt gilt mein Dank Ihnen, den Lesern, dafür, dass Sie sich die Zeit nehmen, dieses Buch zu lesen. Ich hoffe, Sie betrachten die investierte Zeit nicht als verloren. Ich weiß, dass ich Ihnen da und dort erhebliche Konzentration abverlange. Aus der Erfahrung meiner Kurse muss ich allerdings sagen, dass leider kein Weg an dieser Lernkurve vorbeiführt. Falls Sie nicht das Gefühl haben, ich stünde Ihrem Verständnis auch noch im Weg, wäre damit schon ein Ziel erreicht, das ich angestrebt habe. Gerne erwarte ich Ihre Rückmeldung, die – über den Verlag – an mich weitergeleitet und von mir beantwortet werden wird, und hoffe, dass Sie mit dem Buch Ihrem Ziel, SQL zu erlernen, ein gutes Stück näherkommen werden.

Kapitel 10
Views erstellen

Als Inner View ist sie uns bereits begegnet, aber auch darüber hinaus ist der Begriff View vielen geläufig, die mit Datenbanken zu tun haben. In diesem Abschnitt möchte ich mich etwas näher mit diesen Views beschäftigen.

Was ist eine View? In meinen Kursen kennen sehr viele Teilnehmer den Begriff, viele können auch sagen, was man mit einer View macht, aber was das genau ist? Da wird es normalerweise recht eng. Dabei ist die Antwort wirklich einfach: Eine View ist eine SQL-Abfrage mit einem Namen, gespeichert in der Datenbank. Weiter nichts. Wichtig: Nur die Abfrage als solche ist gespeichert, nicht aber die Ergebnisse der Abfrage. Die werden beim Abfragen der View dynamisch neu berechnet und ausgegeben. Verschiedentlich höre ich (und schlimmer, lese ich auch in Fachbüchern über SQL) eine Reihe falscher Vorstellungen über Views. Sie benötigten »temporären Speicherplatz«, umfangreichen Festplattenplatz, sie seien langsam oder sonstige Geschichten. Ich bin nicht sattelfest in allen Datenbanken dieser Welt, um diese Geschichten für alle Datenbanken als falsch zu brandmarken, sicher sind sie aber bezüglich Oracle falsch.

Eine View ist eine Abfrage, die Sie in der Datenbank abgelegt haben. Fragen Sie nun eine View ab, heißt das für die Datenbank exakt das Gleiche, als führten Sie die dort gespeicherte Abfrage direkt aus. In beiden Fällen wird die Abfrage auf exakt die gleiche Weise ausgeführt, allerdings hat Oracle im Fall der View den Vorteil, dass es die Abfrage bereits kennt und sie daher schon auf syntaktische Korrektheit hin prüfen konnte. Diese Arbeit entfällt nun. Da aber SQL-Abfragen in einem Cache vorgehalten werden, nachdem Sie einmal ausgeführt wurden, verliert sich dieser Vorteil nach der ersten Ausführung der Abfrage.

10.1 »Normale« Views

Beginnen wir mit den einfachen, normalen Views. Diese Views verhalten sich genauso, wie im Einführungstext beschrieben. Von diesen Views unterscheide ich später dann noch die *materialisierten Views* in Abschnitt 10.4, doch beginnen wir zunächst mit der einfachsten Form.

10.1.1 Was genau ist eine View?

Wenn eine View den Namen EMP_VW für die SQL-Abfrage

```
select ename, job, sal
  from emp
 where deptno = 30;
```

vereinbart hat und Sie nun in einer Abfrage den Namen der View als Tabelle benutzen, wird diese Abfrage

```
select *
  from emp_vw
```

umgeschrieben zu

```
select *
  from (select ename, job, sal
               from emp
        where deptno = 30);
```

Listing 10.1 Was ist eine View?

Und diese letzte Abfrage kennen Sie bereits als *Inner View*. Ob Sie also eine Inner View explizit hinschreiben oder aber die Inner-View-Abfrage unter einem Namen in der Datenbank speichern und über den Namen benutzen, macht überhaupt keinen Unterschied. Theoretisch ist die in der Datenbank gespeicherte View marginal schneller (zumindest bei der ersten Ausführung), weil die zugrunde liegende SQL-Anweisung ja bereits der Datenbank bekannt und somit geparst ist, doch können Sie dies im Regelfall ignorieren. Die Stärken von Views liegen auf anderem Gebiet. Bevor wir uns diese Gebiete ansehen, möchte ich nur kurz klären, was ich damit meine, dass die View in der Datenbank gespeichert sei. Wir haben bereits den Begriff *Data Dictionary* besprochen. Weil das aber lange her ist: Das war die Sammlung aller Metadaten zu unseren Daten, also welche Tabellen gibt es, welche Spalten sind darin enthalten, welche Benutzer existieren, welche Rechte haben diese. In diesem Data Dictionary werden die SQL-Anweisungen als Zeichenketten unter dem Namen des Benutzers abgelegt, dem diese Views gehören. Sind Sie also als Benutzer SCOTT angemeldet, würde Ihre select-Anweisung unter dem gegebenen Namen und dem Eigentümer SCOTT abgelegt. Abbildung 10.1 zeigt die View im SQL Developer.

Sie können sich aber auch direkt in SQL anzeigen lassen, welche Views Ihnen gehören oder welche Sie darüber hinaus benutzen dürfen. Die Daten zu der View sind ja, wie bereits gesagt, im Data Dictionary der Datenbank gespeichert. Das wiederum sind Tabellen, in denen Daten stehen. Da allerdings das Datenmodell des Data Dictionarys recht komplex ist, wird der Zugriff auf diese Tabellen normalerweise über eine Reihe

von Views erledigt, die Oracle bei der Installation der Datenbank bereits angelegt hat. Von diesen Views gibt es eine ganze Menge, zu viele, um alle zu kennen. Allerdings lassen sich die Namen von vielen der Views gut merken, denn sie folgen einer Namenskonvention: Unterschieden wird anhand des Präfixes, ob Ihnen die entsprechende View die Objekte zeigt, auf die Sie Zugriff haben (ALL_), oder diejenigen, die Ihnen gehören (USER_). Der Unterschied liegt darin, dass eine View, die einem anderen Benutzer gehört, Ihnen zur Nutzung zur Verfügung gestellt werden könnte. Administratoren haben zudem noch eine große Zahl an Views, die mit dem Präfix DBA_ beginnen und zeigen, welche Objekte es generell im Data Dictionary gibt.

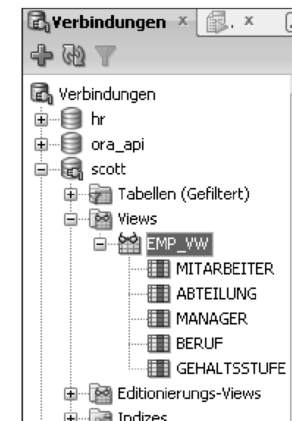

Abbildung 10.1 Die Übersicht über die Views im SQL Developer

Ich möchte Ihnen nun die View EMP_VW im Data Dictionary zeigen und hätte hierfür zwei Möglichkeiten: Die Views USER_VIEWS und ALL_VIEWS. Die zweite der beiden Views frage ich hier ab, weil diese View auch den Namen des Eigentümers enthält:

```
SQL> select owner, view_name, text_length, text_vc
  2    from all_views
  3   where owner 'SCOTT';

OWNER VIEW_NAME  TEXT_LENGTH TEXT_VC
----- ---------- ----------- ---------------------------------
SCOTT EMP_VW             267 select e.ename mitarbeiter,
                                    d.dname abteilung,
                                    m.ename manager, ...
```

Listing 10.2 Abfrage der View ALL_VIEWS

Wie Sie sehen, wird die select-Abfrage genauso abgespeichert, wie Sie sie eingegeben haben. Das ist sozusagen der Beleg: Eine View ist tatsächlich lediglich eine select-Abfrage mit einem Namen. Eine wichtige Besonderheit existiert allerdings:

Wenn Sie eine View definieren als

```
select *
  from emp;
```

wird dies nicht so, sondern in folgender Form abgelegt:

```
select empno, ename, job, mgr, hiredate, sal, comm, deptno
  from emp;
```

Der Platzhalter * wird also zu einer Spaltenliste aller Spalten aufgelöst. Das ist wichtig, falls Sie die Tabellen anschließend ändern möchten. Stellen Sie sich vor, Sie löschen eine Spalte oder benennen eine Spalte der Tabelle um. Nun wird die View ungültig werden, denn die in der View angesprochene Spalte existiert in der Tabelle nicht mehr. Fügen Sie der Tabelle aber eine neue Spalte hinzu, wird die View diese Änderung nicht mitbekommen, also gültig bleiben (weil die zugrunde liegende Abfrage ja nach wie vor gültig ist) und nur die Spalten liefern, die zum Zeitpunkt der Definition der View in der Tabelle enthalten waren. Diese Eigenheit ist einer der Gründe, warum normalerweise der Platzhalter * in `select`-Abfragen nur für Ad-hoc-Abfragen akzeptabel ist. Er hat einfach zu viele Seiteneffekte, wenn Sie in Views oder sonst wie gespeicherten Abfragen verwendet werden.

10.1.2 Wie werden Views erstellt?

Zunächst einmal benötigen Sie, um Views zu erstellen, das Recht, dies zu tun, nämlich das Recht `create view`. Dieses Recht erhalten Sie, falls Sie es nicht bereits besitzen, vom Administrator der Datenbank (sind Sie das selbst, dann müssen Sie die Anweisung hierfür kennen, sie lautet `grant create view to <benutzername>`). Ich bin der festen Überzeugung, dass ein Benutzer, der Daten in einer Datenbank lesen darf, auch in der Lage sein muss, Views anzulegen, denn letztlich kann eine View niemals mehr als der Benutzer ohnehin darf, es ist also keine Erweiterung der Rechte in Bezug auf die Daten. Im Gegenteil unterstützen Views aber die Wiederverwendbarkeit der Abfragen, denn nun können andere Leseberechtigte die Abfragen untereinander zur Verfügung stellen und entsprechend nutzen. Zudem erhält der Administrator im Zweifel einen direkteren Zugang zu problematischen Anweisungen, weil sich diese nicht in einer lokalen Skriptdatei, sondern im Data Dictionary der Datenbank befinden, auf die der Administrator direkten Zugriff hat.

Erstellung von Views

Haben Sie das Recht, eine View zu erstellen, so ist dies ganz einfach. Sie benötigen die Anweisung `create or replace view <Name> as select ...`, um eine SQL-Anweisung als View in der Datenbank zu hinterlegen. Hier sehen Sie ein Beispiel:


```
SQL> create or replace view emp_vw as
  2  select e.ename mitarbeiter,
  3         d.dname abteilung,
  4         m.ename manager,
  5         e.job beruf,
  6         s.grade gehaltsstufe
  7    from emp e
  8    join dept d on e.deptno = d.deptno
  9    left join emp m on e.mgr = m.empno
 10    join salgrade s on e.sal between s.losal and s.hisal;

View wurde erstellt.
```

Listing 10.3 Beispiel für die Erstellung einer View

Das war alles. Nun ist die View nutzbar wie eine Tabelle:

```
SQL> select *
  2    from emp_vw;

MITARBEITER ABTEILUNG      MANAGER    BERUF     GEHALTSSTUFE
----------- -------------- ---------- --------- ------------
KING        ACCOUNTING                PRESIDENT            5
FORD        RESEARCH       JONES      ANALYST              4
SCOTT       RESEARCH       JONES      ANALYST              4
JONES       RESEARCH       KING       MANAGER              4
BLAKE       SALES          KING       MANAGER              4
CLARK       ACCOUNTING     KING       MANAGER              4
ALLEN       SALES          BLAKE      SALESMAN             3
TURNER      SALES          BLAKE      SALESMAN             3
MILLER      ACCOUNTING     CLARK      CLERK                2
MARTIN      SALES          BLAKE      SALESMAN             2
WARD        SALES          BLAKE      SALESMAN             2
ADAMS       RESEARCH       SCOTT      CLERK                1
JAMES       SALES          BLAKE      CLERK                1
SMITH       RESEARCH       FORD       CLERK                1
```

Listing 10.4 Verwendung der View EMP_VW

Als Speicherplatz wurden nur die 267 Byte benötigt, die in der Auswertung oben für diese View angezeigt werden. Na ja, einige Byte mehr schon, aber wirklich kein relevanter Plattenplatz. Fragen Sie die View mit einer `select`-Abfrage ab, wird die View, wie erläutert, als Inner View an der Stelle Ihrer Anweisung eingefügt, an der vorher der Name der View stand. Diese Ersetzung können Sie sich gern als Zeichenoperation

vorstellen, es wird also tatsächlich die Zeichenfolge der View-Abfrage als Inner View in die umgebende Abfrage der äußeren Abfrage eingefügt und erst danach die Abfrage optimiert. Dadurch erhält die Datenbank einen Überblick über die komplette Abfrage und optimiert nach Kräften die gesamte Abfrage.

Ab Version 12c der Datenbank gibt es sogar ein Package, das uns genau anzeigt, welche `select`-Abfrage die Datenbank ausführt. Nachfolgend sehen Sie also das, was die Datenbank tatsächlich tut, wenn Sie die Anweisung aus Listing 10.4 ausführen. Um dieses Ergebnis zu sehen, müssen Sie allerdings ein wenig programmieren können, denn das geht nicht direkt in SQL, sondern nur in PL/SQL. Daher möchte ich Ihnen hier nur die (etwas aufgehübschte) Ausgabe dieses Programms zeigen:

```
SELECT "A1"."MITARBEITER" "MITARBEITER",
       "A1"."ABTEILUNG" "ABTEILUNG",
       "A1"."MANAGER" "MANAGER",
       "A1"."BERUF" "BERUF",
       "A1"."GEHALTSSTUFE" "GEHALTSSTUFE"
  FROM (SELECT "A2"."ENAME_1" "MITARBEITER",
               "A2"."DNAME_7" "ABTEILUNG",
               "A2"."ENAME_9" "MANAGER",
               "A2"."JOB_2" "BERUF",
               "A2"."GRADE_14" "GEHALTSSTUFE"
          FROM (SELECT "A4"."EMPNO_0" "EMPNO",
                        "A4"."ENAME_1" "ENAME_1",
                        "A4"."JOB_2" "JOB_2",
                        "A4"."MGR_3" "MGR",
                        "A4"."SAL_4" "SAL",
                        "A4"."QCSJ_C000000000400000_5" "DEPTNO",
                        "A4"."QCSJ_C000000000400001_6" "DEPTNO",
                        "A4"."DNAME_7" "DNAME_7",
                        "A4"."EMPNO_8" "EMPNO",
                        "A4"."ENAME_9" "ENAME_9",
                        "A4"."JOB_10" "JOB",
                        "A4"."MGR_11" "MGR",
                        "A4"."SAL_12" "SAL",
                        "A4"."DEPTNO_13" "DEPTNO",
                        "A3"."GRADE" "GRADE_14",
                        "A3"."LOSAL" "LOSAL",
                        "A3"."HISAL" "HISAL"
                   FROM (SELECT "A6"."EMPNO_0" "EMPNO_0",
                                "A6"."ENAME_1" "ENAME_1",
                                "A6"."JOB_2" "JOB_2",
                                "A6"."MGR_3" "MGR_3",
                                "A6"."SAL_4" "SAL_4",
```



```
                                "A6"."DEPTNO_5" "QCSJ_C000000000400000_5",
                                "A6"."DEPTNO_6" "QCSJ_C000000000400001_6",
                                "A6"."DNAME_7" "DNAME_7",
                                "A5"."EMPNO" "EMPNO_8",
                                "A5"."ENAME" "ENAME_9",
                                "A5"."JOB" "JOB_10",
                                "A5"."MGR" "MGR_11",
                                "A5"."SAL" "SAL_12",
                                "A5"."DEPTNO" "DEPTNO_13"
                           FROM (SELECT "A8"."EMPNO" "EMPNO_0",
                                        "A8"."ENAME" "ENAME_1",
                                        "A8"."JOB" "JOB_2",
                                        "A8"."MGR" "MGR_3",
                                        "A8"."SAL" "SAL_4",
                                        "A8"."DEPTNO" "DEPTNO_5",
                                        "A7"."DEPTNO" "DEPTNO_6",
                                        "A7"."DNAME" "DNAME_7"
                                   FROM SCOTT."EMP" "A8",
                                        SCOTT."DEPT" "A7"
                                  WHERE "A8"."DEPTNO" = "A7"."DEPTNO"
                                ) "A6",
                                SCOTT."EMP" "A5"
                          WHERE "A6"."MGR_3" = "A5"."EMPNO"(+)
                         ) "A4",
                         SCOTT."SALGRADE" "A3"
                   WHERE "A4"."SAL_4" >= "A3"."LOSAL"
                     AND "A4"."SAL_4" <= "A3"."HISAL"
                  ) "A2"
       ) "A1"
```

Listing 10.5 Wollten wir wirklich wissen, was Oracle tut?

Hm – oder zumindest so ähnlich … Vielleicht ist es ganz gut, dass Oracle diese Dinge vor uns verbirgt. Die Ausgabe oben wird von Oracle übrigens komplett ohne Einrückungen etc. ausgegeben. Da dieser Weißraum ja nur etwas für menschliche Leser ist, kann Oracle intern darauf verzichten.

Für diejenigen, die sich dafür interessieren, wie diese Ausgabe zustande kommt: Sehen Sie online einmal unter `DBMS_UTILITY.EXPAND_SQL_TEXT` nach, da werden Sie fündig.

Die Klausel FORCE

Anmerkungen zum Thema »Erzeugung von Views« gibt es eigentlich nur wenige. Vielleicht sollte ich erwähnen, dass es noch ein optionales Schlüsselwort `force` gibt,

mit dem die View erzeugt werden kann. Diese Klausel legt fest, dass die View in jedem Fall angelegt wird, auch wenn die der View zugrunde liegende Abfrage ungültig ist. Das ist manchmal wichtig, wenn eine ganze Reihe an Views mit Hilfe eines Skripts angelegt werden sollen und diese Views aufeinander aufbauen. Ohne diese Klausel schlüge die Erstellung einzelner Views fehl, wenn sie nicht in der richtigen Reihenfolge angelegt würden. Verwenden Sie diese Klausel nicht, wird standardmäßig die Klausel `no force` angenommen, was Sie auch explizit schreiben könnten. Wird eine View zunächst ungültig angelegt und später durch die Anlage weiterer Views doch gültig, kann sie direkt benutzt werden, denn der erste Zugriff auf eine ungültige View hat zur Folge, dass die Datenbank nun versucht, die View erneut zu parsen. Gelingt es nun, wird die View verwendet.

Hier ist die Anweisung von eben mit der Klausel `force`. Um allerdings zu zeigen, dass die Klausel auch eine Funktion hat, werde ich die Abfrage durch einen Fehler ungültig machen:

```
SQL> create or replace force view emp_wrong_vw as
  2  select e.ename mitarbeiter,
  3         d.dname abteilung,
  4         m.ename manager,
  5         e.jobname beruf,
  6         s.grade gehaltsstufe
  7    from emp e
  8    join dept d on e.deptno = d.deptno
  9    left join emp m on e.mgr = m.empno
 10    join salgrade s
 11      on e.sal between s.losal and s.hisal;

Warnung: View wurde mit Kompilierungsfehlern erstellt.
```

Listing 10.6 Verwendung der Klausel FORCE

Der falsche Spaltenname ist natürlich so nicht erlaubt. Durch die Klausel `force` hat das allerdings der Erstellung selbst nicht im Wege gestanden. Eine andere View des Data Dictionarys, die View `USER_OBJECTS`, gibt Auskunft über den Status dieser View:

```
SQL> select object_name, status
  2    from user_objects
  3   where object_type = 'VIEW';

OBJECT_NAME            STATUS
---------------------- -------
EMP_WRONG_VW           INVALID
EMP_VW                 VALID
```

Listing 10.7 Abfrage des Data Dictionarys

Sie sehen an diesem Beispiel übrigens, dass die Views des Data Dictionarys doch recht interessante Informationen enthalten. Alles, was auf der Oberfläche des SQL Developers gezeigt wird und sich auf die Datenbankstruktur bezieht, kommt aus diesen Views. Administratoren leben von diesen Views, denn sie zeigen, wie es der Datenbank »geht« und welche administrativen Schritte zu unternehmen sind. Bleibt die Frage: Welche Views gibt es denn nun eigentlich? Gegenfrage: Was fragen Sie mich? Fragen Sie doch einmal die View `ALL_OBJECTS`, und filtern Sie über den Objekttyp `VIEW` ...

Sollten Sie dies tatsächlich einmal tun, stellen Sie fest, dass Ihnen auch eine Reihe Views mit dem Präfix `V_$` angeboten werden. Diese Views werden (allerdings mit dem Präfix `V$`, ohne den Unterstrich) verwendet, um Dinge zu erfragen, die nicht permanent im Data Dictionary gespeichert werden, sondern die aktuelle Situation widerspiegeln, wie zum Beispiel die Anzahl aktuell angemeldeter Benutzer, der momentane Speicherverbrauch im Arbeitsspeicher etc. Daher heißen diese Views auch *Performance-Views*. Dass diese Views ohne Unterstrich verwendet werden, hat damit zu tun, dass für diese Views ein Synonym ohne Unterstrich vereinbart wurde. Ein weiteres Präfix lautet `GV$` und ist wichtig, wenn Ihre Datenbank als Cluster mit der sogenannten *RAC*-Option (für *Real Application Cluster*) von Oracle aufgesetzt wurde. In diesem Fall besteht Ihr RDBMS aus mehreren, parallel laufenden Datenbankinstanzen, die gemeinsam unter einem *Service Name* als eine Datenbank erscheinen. In solchen Clustern unterscheiden wir das konkrete System, mit dem wir in unserer Session verbunden sind, und den allgemeineren Service, der aus eventuell mehreren Clustern als Verbund aufgebaut wurde. Möchten Sie Informationen zur Datenbankinstanz, mit der Sie aktuell verbunden sind, nutzen Sie die bereits bekannten Präfixe `V$`. Wenn Sie aber Informationen über mehrere Cluster hinaus benötigen, verwenden Sie das Präfix `GV$`.

Die Klausel WITH CHECK OPTION

Diese Klausel ist schon etwas hinterhältiger zu erklären. Sie hat auf den lesenden Zugriff überhaupt keine Auswirkung, sondern auf den schreibenden Zugriff. Stellen Sie sich vor, Sie hätten eine View auf die Tabelle `EMP` eingerichtet, die lediglich die Mitarbeiter der Abteilung `20` anzeigen soll:

```
SQL> create or replace view emp_dept_20 as
  2  select ename, job, sal, hiredate, deptno
  3    from emp
  4   where deptno = 20
  5    with check option;

View wurde erstellt.
```

Listing 10.8 Eine View mit CHECK-Option

Diese View erfüllt das Kriterium, eine »einfache« View zu sein (was das ist, erläutere ich in Abschnitt 10.1.3, »Einfache und komplexe Views«), und kann daher direkt mit einer update-Anweisung aktualisiert werden. Nun könnten wir uns vorstellen, dass Sie auf den Gedanken kommen, einen etwas unbeliebteren Kollegen schlicht in eine andere Abteilung zu versetzen. Zumindest hätte dies ja zur Folge, dass Sie diesen Mitarbeiter in der View nun nicht mehr sehen:

```
SQL> update emp_dept_20
  2     set deptno = 30
  3   where ename = 'ADAMS';
update emp_dept_20
       *
FEHLER in Zeile 1:
ORA-01402: Verletzung der WHERE-Klausel einer View WITH CHECK OPTION
```

Listing 10.9 Auslösen der CHECK-Option mit einer unerlaubten Datenänderung

Der Fehler wird ausgegeben, weil die check-Option festlegt, dass Sie keine Datenänderung »zulasten Dritter« vornehmen dürfen, also Datenänderungen, die den betroffenen Datenbestand anschließend für die View unsichtbar machen würden. Der naheliegende Grund: Sie können die Auswirkung Ihrer Änderung nicht mehr kontrollieren, weil Sie das Ergebnis der Änderung nicht sehen können. Das ist eine sehr mächtige Funktion, die man im Hinterkopf haben sollte, wenn man in eine solche Situation kommt. Zusätzlich kann dieser Option (und auch der read-only-Option, die wir im übernächsten Abschnitt besprechen) noch die Klausel constraint, gefolgt von einem Bezeichner, nachgestellt werden. Die constraint-Klausel sorgt dafür, dass der Klausel ein Name zugeordnet werden kann. Das ist sinnvoll, damit im Data Dictionary nur händisch benannte Objekte stehen. Fehlt diese Klausel, wird ein vom System erzeugter, eindeutiger Bezeichner verwendet, der nicht sehr sprechend ist und daher den Überblick über das Datenmodell erschwert.

View mit COLLATE-Klausel mit Views

In Version 12.2 der Datenbank hat, wie bereits beschrieben, eine neue Klausel und ein neuer Operator zur Steuerung der Sortierung und Filterung von Daten Einzug gehalten. Dieser Operator kann, wie Sie in Abschnitt 6.5.2, »COLLATE-Operator zur Kontrolle des Sortier- und Vergleichsverhaltens«, gesehen haben, in select-Anweisungen genutzt werden, um eine flexiblere Sortierung und Filterung von Daten vorzunehmen. Es ist, bei Licht betrachtet, keine Überraschung, dass dieser Operator auch in einer View verwendet werden kann. Dies gestattet es, eine View auf eine Tabelle anzulegen, die durch diese erweiterten Möglichkeiten eine vereinfachte Suche erlaubt, ohne dem Benutzer die Formulierung einer collate-Klausel abzuverlangen. Ebenso ist es natürlich möglich, die Klausel im Rahmen einer Gruppierung zu verwenden und diese Abfrage als View zu hinterlegen. Ich möchte auf diese Möglichkeit hinweisen, einmal, weil es eine neue Funktionalität ist, und dann, weil mir in meinen Kursen zu SQL immer wieder klar wird, dass für meine Kursteilnehmer ein Hauptproblem beim Erlernen von SQL ist, Anwendungsgebiete für Funktionen und Optionen zu finden und zu nutzen. Nehmen Sie diese Option also als Anregung.

Die Klausel WITH READ ONLY OPTION

Diese Klausel sollte vom Namen her bereits selbsterklärend sein. Sie wird eingesetzt, um DML-Anweisungen auf Views direkt zu unterbinden. Das scheint sehr sinnvoll, ist jedoch in der Praxis selten. Der Grund: Diese Einschränkung lässt sich einfacher über eine entsprechende Rechtevergabe bewerkstelligen. Das Szenario, nur ganz kurz umrissen: Benutzer A ist Eigentümer der View. Er möchte Benutzer B ein Recht einräumen, auf die View zuzugreifen. Dies macht er durch eine Rechtevergabe. Diese Rechtevergabe kann aber nun zum Beispiel lediglich ein select-Recht umfassen. Dadurch ist Benutzer B nicht in der Lage, DML-Anweisungen auf die View auszuführen. Sollte Benutzer A als Eigentümer der View durch diese Klausel eingeschränkt werden, ist das nur ein sehr schwacher Trost, denn das Recht, auf den zugrunde liegenden Tabellen DML-Anweisungen auszuführen, kann dem Eigentümer der Tabellen nicht entzogen werden. Die Klausel hätte hier also eher kosmetische Gründe. Dennoch soll diese Klausel aus Gründen der Vollständigkeit erklärt werden:


```
SQL> create or replace view emp_dept_30 as
  2  select ename, job, sal, hiredate, deptno
  3    from emp
  4   where deptno = 30
  5    with read only constraint chk_emp_dept_ro;
View wurde erstellt.

SQL> update emp_dept_30
  2     set ename = 'RINGER'
  3   where ename = 'TURNER';
 where ename = 'TURNER'
       *
FEHLER in Zeile 3:
ORA-42399: DML-Vorgang kann auf schreibgeschützter View
           nicht ausgeführt werden

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

Listing 10.10 Verwendung der Option READ ONLY mit CONSTRAINT

Eine Anmerkung zu den beiden Klauseln `check option` und `read only`: Wird eine der beiden Klauseln verwendet, darf die Abfrage der View keine Sortierung über `order by` enthalten.

Auch für den Zweck, sich einmal die Constraints anzeigen zu lassen, steht eine View zur Verfügung. Daran können Sie auch schön sehen, wie Oracle Constraints benennt, die von Ihnen keinen eindeutigen Namen erhalten haben:

```
SQL> select constraint_name, constraint_type c_type, table_name
  2    from user_constraints;

CONSTRAINT_NAME  C_TYPE TABLE_NAME
---------------- ------ -----------------------
SYS_C0013395     V      EMP_DEPT_20
CHK_EMP_DEP_RO   O      EMP_DEPT_30
FK_DEPTNO        R      EMP
PK_DEPT          P      DEPT
PK_EMP           P      EMP
```

Listing 10.11 Abfrage der Constraints, die SCOTT gehören

Der Constraint mit dem systemgenerierten Namen ist ein View-Constraint, das auf die View `EMP_DEPT_20` eingerichtet wurde, also das Constraint für die `check option`. Da mir so etwas nicht gefällt, habe ich in der Anweisung zur Erstellung der View `EMP_DEPT_30` die `read-only`- Klausel durch die `constraint`-Anweisung ergänzt und so den erforderlichen Constraint mit einem Namen versehen. Sie können, wenn Sie ein Name wie `SYS_C0013395` nicht stört, diese Klausel auch weglassen.

10.1.3 Einfache und komplexe Views

Wir unterscheiden ferner noch zwischen einfachen und komplexen Views. Diese Unterscheidung ist für die Datenbank aus gutem Grund getroffen worden, denn einfache Views lassen sich mit DML-Anweisungen ändern, komplexe nicht. Damit ist gemeint, dass eine einfache View direkt über eine `insert`-Anweisung geändert werden kann. Natürlich wird nicht die View dadurch geändert, sondern die der View zugrunde liegende Tabelle. Bei einer einfachen View jedoch ist die Datenbank in der Lage, die `insert`-Anweisung direkt und logisch korrekt auf die Tabelle umzuleiten, weil sie »versteht«, welche Spalten der View welchen Spalten der zugrunde liegenden Tabelle entsprechen. Bei komplexen Views ist das nicht der Fall. Nur: Wann ist eine View eine einfache und wann eine komplexe View? Als Faustregel können Sie sich merken: Gruppenfunktionen und Joins machen eine einfache View zu einer komplexen View, denn nun ist nicht mehr klar, auf welche Weise eine Änderung der Daten auf die unterliegenden Tabellen oder Zeilen weitergeleitet werden soll. Sollten Sie

dies etwas genauer wissen wollen, stellt Oracle Ihnen auch für dieses Problem eine View bereit.

Sehen wir uns die Spalten unserer Views, die wir bisher erzeugt haben, an:

```
SQL> select table_name, column_name,
  2         updatable upd, insertable ins, deletable del
  3    from user_updatable_columns
  4   where table_name in (select view_name
  5                          from user_views);

TABLE_NAME               COLUMN_NAME              UPD INS DEL
------------------------ ------------------------ --- --- ---
EMP_DEPT_20              ENAME                    YES YES YES
EMP_DEPT_20              JOB                      YES YES YES
EMP_DEPT_20              SAL                      YES YES YES
EMP_DEPT_20              HIREDATE                 YES YES YES
EMP_DEPT_20              DEPTNO                   YES YES YES
EMP_DEPT_30              ENAME                    NO  NO  NO
EMP_DEPT_30              JOB                      NO  NO  NO
EMP_DEPT_30              SAL                      NO  NO  NO
EMP_DEPT_30              HIREDATE                 NO  NO  NO
EMP_DEPT_30              DEPTNO                   NO  NO  NO
EMP_VW                   MITARBEITER              NO  NO  NO
EMP_VW                   ABTEILUNG                NO  NO  NO
EMP_VW                   MANAGER                  NO  NO  NO
EMP_VW                   BERUF                    NO  NO  NO
EMP_VW                   GEHALTSSTUFE             NO  NO  NO
EMP_WRONG_VW             MITARBEITER              YES YES YES
EMP_WRONG_VW             ABTEILUNG                YES YES YES
EMP_WRONG_VW             MANAGER                  YES YES YES
EMP_WRONG_VW             BERUF                    YES YES YES
EMP_WRONG_VW             GEHALTSSTUFE             YES YES YES

20 Zeilen ausgewählt.
```

Listing 10.12 Darstellung der aktualisierbaren Spalten der Views

Bei näherer Betrachtung könnte die Frage auftauchen, warum die Spalten der View `EMP_DEPT_20` aktualisierbar sind, die von `EMP_DEPT_30` jedoch nicht (oder haben Sie sich direkt an die Klausel `read only` erinnert?), und auch, ob es vielleicht möglich sein könnte, dass eine Spalte zwar »einfügbar«, nicht jedoch aktualisierbar ist? Je nach logischer Situation kann so etwas durchaus einmal vorkommen. Im Zweifelsfall kön-

nen Sie jedenfalls hier nachsehen, ob Sie bereits eine komplexe oder doch nur eine einfache View erstellt haben.

Nun könnte es natürlich sein, dass Sie gern eine komplexe View mit DML-Anweisungen ändern können möchten. Wie gesagt erlaubt das die Datenbank nicht, jedenfalls nicht direkt: Die Lösung für das logische Problem ist, dass Sie ein kleines Programm in PL/SQL schreiben müssen, um der Datenbank zu erläutern, auf welche Weise Änderungen an der View auf die der View zugrunde liegenden Tabellen weitergeleitet werden sollen. Ein solches Programm wird dann an das Ereignis gebunden, dass eine DML-Anweisung auf die View ausgeführt wurde. Ein Programm, das an ein Ereignis in der Datenbank gebunden wird, haben Sie als Trigger bereits kennengelernt. Ein Trigger auf eine View wird `instead-of`-Trigger genannt, denn er führt DML-Anweisungen *statt auf der View* auf den der View zugrunde liegenden Tabellen aus. Solche Trigger sind logisch hinterhältige Biester. Man kann Sie programmieren, allerdings sollten sie sehr sorgfältig auf Seiteneffekte hin getestet werden. Wie die meisten Programmierprobleme ist auch dieser Trigger für uns außerhalb des Fokus dieses Buches.

10.1.4 Fortgeschrittene Views: Analytische Views

Neu in Version 12.2 der Datenbank sind analytische Views. Deren Einsatzzweck ist eindeutig die Verwendung in Datenwarenhäusern, wo sie dazu dienen, komplexe Abfragen auf verschiedenen Aggregationsstufen zu vereinfachen. Da die hier zugrunde liegenden Prinzipien noch nicht klar sind und auch umfangreiche Erläuterungen von mir erfordern, habe ich die Besprechung dieses Viewtyps nach Kapitel 23, »Datenwarenhaus«, verschoben. Dort ist dann Raum zur Besprechung der Grundlagen, auf denen dieser Viewtyp aufbaut.

10.2 Einsatzbereiche von Views

Dass Views Vorteile haben, habe ich bereits erwähnt. Diese liegen allerdings weniger in der Steigerung der Geschwindigkeit (die wird normalerweise nicht tangiert), sondern in anderen Aspekten.

10.2.1 Kapselung von Logik

Ein wesentlicher Vorteil von Views ist, dass die Abfrage, die zur Erzeugung erforderlich ist, in der Definition der View gekapselt wird und damit für den Anwender der View verborgen ist. Sie können also komplizierte SQL-Anweisungen hinter einer View verbergen und dem Anwender eine einfach zu benutzende Schnittstelle auf Ihre Daten zur Verfügung stellen. Das hat einige Vorteile:

- Ein Kollege mit weiter gehenden SQL-Kenntnissen kann komplexe Views definieren und den Kollegen zur Verfügung stellen.
- Abfragen, die in mehreren Zusammenhängen verwendet werden, müssen nicht stets neu formuliert werden.
- Sollte eine Änderung an der Abfrage der View erforderlich werden, aktualisiert dies immer auch alle auf der View basierenden Abfragen.
- Erfordert die Abfrage komplexes SQL, verbirgt sich diese Komplexität hinter einer einfachen View-Schnittstelle.

Gerade Anwendungsentwickler profitieren von dieser letzten Eigenschaft von Views, denn wenn die Anwendung komplizierte `select`-Abfragen benötigt, müssen diese als einfacher Text in einer fremden Programmiersprache wie Java oder C# geschrieben werden. In diesen Sprachen wirken SQL-Anweisungen stets wie Fremdkörper und, schlimmer noch, können von dort nicht unmittelbar auf Korrektheit geprüft werden. Ob die angesprochenen Spalten existieren oder die Anweisung syntaktisch korrekt ist, wird erst zur Laufzeit der Anwendung ermittelt. Das ist nicht sehr komfortabel. Zudem muss die Anwendung definitiv zu viel über die Details der Datenspeicherung wissen: die Namen der Tabellen, die Spalten darin, die Schlüsselbeziehungen etc. All das sind Interna der Datenbank, die auf der abstrakteren Ebene der Anwendung eigentlich nicht mehr bekannt sein sollten.

Wie Sie in der Definition der View oben sehen, können wir über einfache Spaltenalias und die Verknüpfung mehrerer Tabellen eine beliebig gebaute Schnittstelle zu den Daten realisieren, die Kenntnis der unterliegenden Tabellen ist nun nicht mehr erforderlich, die Spalten tragen Bezeichner, die die Auswertung leicht machen. All das sind Vorteile für Anwendungsentwickler, aber auch für Fachabteilungen, die sich naturgemäß leichter mit auf diese Weise vorbereiteten Views tun als mit den Rohdaten in den Tabellen.

10.2.2 Zugriffsschutz

Ein ganz wesentlicher Vorteil von Views ist, dass nur die Spalten angezeigt werden, die auch angezeigt werden sollen. Diesen Vorteil spielen wir insbesondere im Zusammenhang mit der Rechteverwaltung einer Oracle-Datenbank aus, denn in diesem Zusammenhang ist es möglich, einem Benutzer `WILLI` ein Leserecht auf die View `EMP_VW` des Benutzers `SCOTT` einzuräumen, nicht aber auf die Tabellen, die der View zugrunde liegen. Die View `EMP_VW` zeigt zum Beispiel nicht mehr das Gehalt, sondern nur noch die Gehaltsstufe aus der Tabelle `SALGRADE`. Kein Benutzer, der Lesezugriff auf lediglich diese View hat, wird nun das konkrete Gehalt eines Mitarbeiters ermitteln können. Ebenso sind die Details der Speicherung in drei Tabellen und die Schlüsselbeziehungen versteckt. Genauso gut hätte die View natürlich auch noch eine Aus-

wahl über die Zeilen vornehmen können, so dass diese View zum Beispiel nur die Daten einer Abteilung zeigt. Bei geschickter Planung kann eine View die Daten für jeweils unterschiedliche Benutzer auch unterschiedlich zusammenstellen. Oft wird dies genutzt, um Daten einer Datenbank, die von mehreren Niederlassungen gemeinsam verwendet werden, so zu filtern, dass sie für einen angemeldeten Benutzer nur die Daten dessen Niederlassung zeigt. Falls nötig oder erwünscht, können aber natürlich auch übergreifende Informationen, wie zum Beispiel Unternehmensbenchmarks, in eigenen Views angeboten werden, die dieser Einschränkung nicht unterliegen.

Natürlich werden solche Views ein eindeutig komplexeres SQL benötigen als in unseren bisherigen Beispielen. Aber dieses SQL dürfte immer noch bei Weitem einfacher sein als die Anstrengungen, die unternommen werden müssen, um innerhalb der Anwendung die Daten auszublenden, die ein Anwender nicht sehen soll.

10.2.3 Programmieren nach dem Gelbe-Seiten-Prinzip

Ein ganz ähnlicher Vorteil wie die Kapselung der Logik existiert beim Gebrauch von Views noch in anderer Hinsicht: Sollten Sie eine Abfrage erstellt haben, die zwar die richtigen Daten liefert, dies aber nicht so schnell, wie Sie sich das wünschen, können Sie jederzeit einen Kollegen oder externen Dienstleister bitten, sich die Abfrage einmal anzuschauen und eventuell zu beschleunigen. Dieser Kollege kann dann in einer kontrollierten Umgebung so lange an der Abfrage arbeiten, bis identische Daten zur Ursprungsview in der vorgegebenen Antwortzeit ermittelt werden. Sie merken bei den Auswertungen, die auf diesen Views aufbauen, nichts von diesen Veränderungen, außer einer erhöhten Performanz. Ganz ähnlich ist das Argument, wenn diese View dann aus Performanzgründen materialisiert werden soll: Bei diesem Verfahren wird die Auswertung, falls das logisch kein Problem darstellt, in verkehrsarmen Zeiten der Datenbank gerechnet und auf Festplatte gespeichert. Wird die View abgefragt, wird statt der Abfrage nun auf die bereits gerechneten Daten zurückgegriffen. Details zu materialisierten Views finden Sie in Abschnitt 10.4.

10.2.4 Lösung komplexer Probleme in Teilschritten

Views helfen bei der Lösung komplexer Probleme. Dies erreichen Sie dadurch, dass Sie in einem komplexen Bericht zum Beispiel mit der Integration der Stammdatentabellen beginnen können. Viele Berichte setzen Informationen über Bewegungsdaten mit Daten aus Stammdaten, wie den Kundendaten oder Ähnlichem, in Beziehung. Wenn dann in einer View schon einmal die Interna der Stammdaten gekapselt wurden, können Sie sich in einer zweiten View auf die Zusammenstellung der Bewegungsdaten konzentrieren und im letzten Schritt die Daten beider Teilauswertungen kombinieren.

Eng daran angelehnt ist der Vorteil, dass eine View auf die Stammdatentabellen mit relativ hoher Sicherheit auch für andere Berichte wiederverwendet werden kann. Ich erinnere mich an eine Reihe von Abfragen, die sich auf eine Gruppe externer Dienstleister bezog. Diese Dienstleister konnten mehrere Adressen für ihre Niederlassung haben, wobei eine Adresse die Hauptadresse war. Zudem waren viele Tabellen historisierend gestaltet, so dass der Verlauf der Umzüge und die Vertragsverhältnisse über die Zeit nachgezeichnet werden konnten. Für die Auswertungen war das aber im Regelfall irrelevant: Hier zählten die aktuelle Hauptadresse sowie die aktuell gültigen Vertragsverhältnisse. Eine View auf diese Daten konnte einen guten Teil der Komplexität der gesamten Abfrage aufnehmen und zudem in vielen Berichten wiederverwendet werden.

Ähnlich ist eine zweite Stoßrichtung dieses Arguments: Sie können Views dafür benutzen, komplexe Abfragen sozusagen vertikal zu vereinfachen. Damit meine ich, dass Sie Teilprobleme einer Abfrage lösen, um auf dem Ergebnis dieser Abfrage weitere Teilprobleme zu lösen usw. Wenn Sie nicht zu tief schachteln, ist dieser Weg zumindest in der Erstellung der Abfrage oftmals ein einfacher Weg, ein komplexes Problem in den Griff zu bekommen. Später dann, wenn die gesamte Abfrage fachlich korrekt arbeitet, können Sie immer noch überlegen, ob Sie einige Views wieder auflösen und die SQL-Anweisungen direkt in die Abfragen schreiben. Doch oftmals ist das Verständnis eines Problems mindestens ebenso schwierig zu erlangen wie die Formulierung in SQL. Dabei hilft dieser Ansatz durchaus. Dieses Argument hatten wir schon einmal bei der Besprechung der `with`-Klausel bemüht, und auch an dieser Dopplung sehen Sie die Nähe zwischen der Klausel und einer View. Der einzige Unterschied besteht darin, dass eine `with`-Klausel die Daten anonym für eine konkrete Abfrage vorbereitet, während eine View diesen Schritt mit einem Namen versieht und in der Datenbank speichert. Werden Vorbereitungsarbeiten also in mehreren Abfragen benötigt, spricht dies für eine View, ist eine Vorarbeit nur für einen Bericht erforderlich, spricht dies für die `with`-Klausel.

10.3 Wer sollte Views verwenden?

Wer also sollte Views verwenden? Ich meine, jeder, der auch SQL-Abfragen schreibt. Für mich gehören diese beiden Rechte ähnlich eng zusammen wie das Recht, Word-Dokumente zu erstellen und diese auch auf Festplatte speichern zu dürfen. Was macht es für einen Sinn, komplexe Anweisungen schreiben zu dürfen, diese aber in lokalen Dateisystemen zu speichern? Dazu sind Datenbanken da. Views fressen kein Brot und erweitern Ihre Rechte auf die Daten nicht unzulässig. Dabei ist natürlich, gerade in größeren Entwicklerteams, darauf zu achten, dass die Anzahl der Views nicht überhandnimmt. Wenn Views zu Testzwecken, als Entwicklungswerkzeug oder aus sonstigem Zweck erstellt und dann in der Datenbank »vergessen« werden, halte

ich dies für problematisch, weil der Überblick über das, was noch benötigt wird, und das, was »weg kann«, verloren geht. Als Ausweg bietet sich eine Namenskonvention an, zum Beispiel mit einem Präfix, abgeleitet aus den Initialen des Entwicklers. Durch das Präfix lassen sich nun alle Views, die privat für einen Entwickler sind, durch einen Filter oder die Sortierung separieren. Wird eine dieser Views dann für die Allgemeinheit freigegeben, wird der Name der View entsprechend geändert. So kann man auch bei vielen Views den Überblick behalten.

Die interessantere Frage ist vielleicht: Wann sollten Sie Views *nicht* verwenden? Eigentlich sprechen nur zwei Gründe gegen die Verwendung: wenn eine View zu breit oder zu tief wird. Vielleicht planen Sie die *One-size-fits-it-all*-View, eine View, die für alle denkbaren Fragestellungen bereits die Spalten bereithält. Das ist keine gute Idee. Generell und ein bisschen über den Daumen können Sie stets davon ausgehen, dass eine Schere aufklafft zwischen den Anforderungen *generisch verwendbar* und *schnell*. Ausnahmen bestätigen immer die Regel, aber wenn eine View Hunderte Spalten definiert und im Regelfall nur ganz wenige dieser Spalten auch wirklich angezeigt werden, müssen Sie im Hinterkopf behalten, dass alle Tabellen, die Spalten zuliefern, durch die Abfrage auch angesprochen werden – ob deren Spalteninformationen durch Ihre Abfrage nun benötigt werden oder nicht. Übertreiben Sie hier, sind Sie sicher auf dem falschen Weg. Hier sind mehrere kleine Views mit einem Ausschnitt der Daten richtiger.

Die andere Richtung wäre die exzessive Schachtelung der Views. Wenn eine View auf einer View auf einer View usw. aufbaut, ist irgendwann ein Punkt erreicht, an dem die Datenbank keinen guten Ausführungsplan für diese Abfrage mehr wird errechnen können. Meiner Erfahrung nach sollten Sie nicht über zwei bis drei Schachtelungsebenen hinausgehen. Natürlich hängt das vom Einzelfall ab, aber um ein Gefühl zu bekommen, mag das hinkommen. Besonders schlecht ist, wenn eine View eine andere View referenziert, aber nicht alle ihrer Spalten benötigt. Dann kann es sein, dass zusätzlich zum Problem der großen Schachtelungstiefe auch noch das erste Problem mit den vielen abgefragten, aber nicht genutzten Spalten hinzukommt. In Summe wird dann – bildlich gesprochen – die halbe Datenbank befragt, nur um im Endeffekt drei Informationen zu ermitteln. Das ist ebenfalls sicher der falsche Weg.

Ansonsten empfehle ich Views ganz dringend den Anwendungsentwicklern. Views sind ein natürliches Mittel zur Entkoppelung der Schichten einer Anwendung. Warum muss die Änderung einer Spaltenbezeichnung zur Folge haben, dass der gesamte Anwendungscode durchsucht und neu kompiliert und ausgeliefert werden muss? Hier ist der Zugriff über Views viel einfacher und logischer. Natürlich weiß ich, dass heutzutage sehr oft Frameworks zur Entkopplung der Datenbank eingesetzt werden, wie etwa *Hibernate* oder *TopLink*. Doch auch diese Technologien durchbrechen dieses Problem nicht, sie verschieben es nur, denn nun müssen XML-Konfigurationsdateien angepasst werden. Auch hier können Views dazu führen, dass

weniger Datenmodelländerungen über die Datenbank hinaus propagiert werden müssen, einfach, weil sich die Spaltenbezeichnungen der Views nicht ändern. Ändert sich eine Tabellenspaltenbezeichnung, wird die zugehörige View invalide. Fixen Sie dieses Problem, ohne das Spaltenalias der Spalte zu ändern, wird die aufrufende Umgebung davon nichts erfahren. Zudem haben Views den Vorteil, dass sie einfach und ohne Aufwand lokal in der Datenbank auf fachliche Korrektheit hin geprüft werden können.

10.4 Materialized View

Was zunächst wie ein schlechter Einfall aus Star Trek klingt, ist ein Datenbankkonstrukt, das die Vorteile von Views mit denen eines Index und einer Tabelle vereinigt: die *Materialized View*. Eine relativ kleine Änderung am Begriff öffnet eine ganz neue Welt für die Beschleunigung von Anwendungen, Abfragen und Berichten.

10.4.1 Was ist eine Materialized View?

Zunächst einmal ist eine Materialized View eine `select`-Abfrage, deren Ergebnis zu einem bestimmten Zeitpunkt berechnet und auf Festplatte geschrieben wurde. Sie können sich eine Materialized View also wie eine Tabelle auf Basis einer `select`-Abfrage vorstellen. Im Gegensatz zu einer solchen Tabelle hat die Materialized View aber eine Reihe von Vorteilen, denn sie kann vereinbaren, zu bestimmten Zeitpunkten aktualisiert zu werden. Das kann entweder eine wiederkehrende Zeit sein, aber auch ein Ereignis, wie etwa, dass sich die Daten der Tabellen ändern, die der materialisierten Sicht zugrunde liegen. Sie können die Aktualisierung aber auch händisch oder programmtechnisch anfordern und so an beliebige Umstände binden.

Da die Daten der Abfrage zu einem bestimmten Zeitpunkt gerechnet und anschließend gespeichert werden, ist die Materialized View also per Definition *nicht aktuell*. Das ist natürlich ein Nachteil, ist aber, je nach Situation, nicht schlimm. So ist es zum Beispiel so, dass sich ein Großteil des Berichtswesens nicht auf die aktuelle Situation, sondern auf den Datenbestand von gestern, des letzten Monats oder des abgelaufenen Quartals bezieht. In einem solchen Zusammenhang ist es natürlich kein Problem, wenn die letzten Millisekunden fehlen. Auf der Habenseite der *MV* (Materialized View – ich habe den Begriff nun, glaube ich, oft genug geschrieben) steht dabei ein äußerst gewichtiges Argument: Da sich die Berichtsabfrage nun nicht auf die sich ständig ändernde Produktionstabelle bezieht, sondern zumindest in Teilen auf die MV, entsteht kein Konflikt zwischen dem Lesezugriff und den Transaktionen der Tabellen, wie ich das bereits beschrieben habe.

Kenner der Materie werden mich nun eher als aus dem Bereich der transaktionsorientierten Datenbanken kommend verorten. Im Gegensatz dazu standen ja die

Datenwarenhäuser, die als Entscheidungssystem für ein komplexes Berichtswesen dienen. Streng genommen haben die MVs in diesem Umfeld ihre originäre Heimat, und zwar als transparente Beschleunigungsmöglichkeit für komplexe Berichte. Wie das geht, erläutere ich in Kapitel 23, »Datenwarenhaus«. Hier beschränke ich mich zunächst auf eine direkte Verwendung.

Eine MV stellt also eine Art Tabelle dar, die Daten anderer Tabellen für das Berichtswesen oder andere Zwecke bereithält. Nun könnten Sie sich vorstellen, dass wir diesen Zweck auch erreichen könnten, indem wir eine Tabelle bereitstellen und mit Hilfe einer `select`-Abfrage in regelmäßigen Abständen mit Daten füllen. Welchen Vorteil sollte dann eine MV gegenüber einer solchen Tabelle haben? Am Ende, so viel ist klar, ist eine Tabelle eine Tabelle. Ob Sie diese händisch oder mit Hilfe der Automatismen einer MV mit Daten gefüllt haben, ist letztlich für die Benutzung der Tabelle uninteressant. Aber: Zum einen bietet die MV eine Reihe weiter gehender Optionen (zum Beispiel das inkrementelle Aktualisieren, bei dem lediglich die seit der letzten Aktualisierung geänderten Daten aktualisiert werden), doch wiegt zum anderen mindestens ebenso schwer: Die MV dokumentiert, welche Daten in ihr enthalten sind, denn die `select`-Abfrage, die für die Datengewinnung benötigt wird, ist Teil der Definition der MV. Zudem sind alle nötigen Programmierarbeiten, um die Daten zu aktualisieren, bereits erledigt. Tom Kyte hat diesen Vorteil einmal so umschrieben:

> *»Also, die internen, optimierten Trigger in C, die Oracle geschrieben hat, sind also langsamer und verbrauchen mehr Ressourcen als der interpretierte, in PL/SQL geschriebene Code, den Sie entwickelt haben? Cool …«*

Es kommt hinzu, dass dieser Code bereits getestet und millionenfach eingesetzt wurde. Zudem können MVs wie ein Index dazu herangezogen werden, andere Abfragen zu beschleunigen, ohne dass dies explizit programmiert werden müsste. All dies sollte Sie davon überzeugen: MVs existieren mit guter Begründung. Sie stellen keine allein selig machende Lösung dar, sind aber ein wichtiges Werkzeug in Ihrem SQL-Werkzeugkasten.

10.4.2 Erstellung von materialisierten Sichten

Lassen Sie uns etwas konkreter werden. Wie wird die Aktualisierung von MVs gesteuert, welche Optionen bieten sich hier? Zunächst einmal müssen Sie ein neues Element kennenlernen, und zwar den sogenannten `JOB`. Ein `JOB` in der Datenbank ist ein Programm, das zu einer bestimmten Zeit ausgeführt wird, ähnlich einem `cron`-Job in UNIX oder einem `at`-Job in Windows. Zu einem Job gehören also eine Aktion, die ausgeführt werden soll, und eine Angabe über die Zeit, zu der die Aktion ausgeführt werden soll. Jobs in einer Oracle-Datenbank können wiederkehrend ausgeführt werden. Die Zeit, zu der dies passiert, wird über eine Datumsberechnung ermittelt. Was passieren soll, ist bei einer materialisierten Sicht relativ einfach zu beschreiben: Sie soll

aktualisiert (`refresh`) werden. Dazu stellt die Datenbank ein vorgefertigtes Programm zur Verfügung. Dieses Programm befindet sich in einem *Package*, das eine Sammlung von Programmen zu einem Thema darstellt. Konkret benötigen wir das Package `dbms_refresh` und daraus das Programm `refresh`. Dies als Hintergrundinformation. Wenn wir nun eine MV erstellen möchten, benötigen wir beinahe die gleichen Klauseln wie zum Erstellen einer normalen View, allerdings werden Sie einige Optionen kennenlernen, die für die Aktualisierung der View von Belang sind. Aus dem eher komplexen Umfeld, in dem diese MVs verwendet werden, resultieren auch einige sehr spezielle Optionen. Diese werde ich mir und Ihnen hier ersparen, ebenso wie eine in die Tiefe gehende Diskussion über die logischen Grenzen inkrementeller Datenaktualisierung, ich denke, es ist auch so schon anspruchsvoll genug.

Ein einfaches Beispiel

Stellen wir uns zunächst eine einfache Aufgabe. Diese soll darin bestehen, lediglich eine der Views, die Sie bereits kennengelernt haben, als MV zu definieren. Wir möchten, dass sich die Daten der MV jeweils um `01:00 Uhr` aktualisieren, damit wir am nächsten Morgen einen aktuellen Datenbestand vorfinden. Wir verwenden das einfachste Aktualisierungsmodell, das da heißt: Alles Alte weg, mach neu!

```
SQL> create materialized view emp_mv
  2  refresh complete on demand
  3  start with sysdate
  4  next trunc(sysdate) + interval '1 1' day to hour
  5  as
  6  select e.ename mitarbeiter,
  7         d.dname abteilung,
  8         m.ename manager,
  9         e.job beruf,
 10         s.grade gehaltsstufe
 11    from emp e
 12    join dept d on e.deptno = d.deptno
 13    left join emp m on e.mgr = m.empno
 14    join salgrade s
 15      on e.sal between s.losal and s.hisal;

Materialized View wurde erstellt.
```

Listing 10.13 Erstellung einer Materialized View

Von hinten her gesehen ist alles ganz bekannt, dort gibt es eine SQL-Abfrage, die die Daten bereitstellt. Zu Beginn der Anweisung fällt auf, dass keine Klausel `or replace` verwendet wurde. Das geht auch nicht, denn Objekte, die physikalisch auf die Platte

geschrieben werden, können mit `replace` nicht ersetzt werden. Dann legen wir fest, dass wir eine `materialized view` erstellen und ihr einen Namen geben möchten. Das Spannende kommt danach:

Mit der Klausel `refresh complete on demand` legen wir fest, dass diese MV bei der Aktualisierung komplett gelöscht und neu beschrieben werden soll. Zudem sagen wir, dass dies auf unser Verlangen hin geschehen soll (und nicht etwa, weil eine Transaktion die *Basistabellen* der MV, also die in der `select`-Abfrage der MV verwendeten Tabellen, verändert hat). Dann legen wir fest, dass die MV sofort gerechnet (`start with sysdate`) und dann jeweils am Folgetag um `01:00 Uhr` aktualisiert wird. Sagen Sie bloß, Sie müssten die Rechenregeln für Datumsangaben noch einmal nachschlagen? Wie Sie sehen, hat die Anweisung funktioniert und die MV erstellt. Das muss nicht so sein, insbesondere dann nicht, wenn Sie den Standardbenutzer `SCOTT` unverändert übernommen haben. Diesem Benutzer fehlt das Recht `create materialized view`, daher dürfte dies bei Ihnen nicht ohne unser Einrichtungsskript aus Kapitel 1, »Einführung«, gelingen.

Der Unterschied der MV zu einer normalen View zeigt sich, wenn sie abgefragt wird. Bei einer normalen View würden die Basistabellen angesprochen, was eventuell mehr Rechenzeit erfordert, aber den Vorteil aktueller Daten hat. Im Gegensatz dazu sollten bei einer MV die Basistabellen nicht mehr angesprochen werden, sondern nur noch die bereits berechneten Daten der MV. Dass dies so ist, können wir relativ leicht überprüfen, wenn wir die Ausführungspläne der beiden `select`-Abfragen (gegen die View `EMP_VW` und gegen die MV `EMP_MV`) miteinander vergleichen:

```
SQL> set autotrace on
SQL> select mitarbeiter, abteilung
  2    from emp_mv;

MITARBEITER ABTEILUNG
----------- -------------
KING        ACCOUNTING
FORD        RESEARCH
SCOTT       RESEARCH
...
14 Zeilen ausgewählt.

Ausführungsplan
----------------------------------------------------------
Plan hash value: 2967684236
----------------------------------------------------------
| Id  | Operation            | Name    | Rows
----------------------------------------------------------
|   0 | SELECT STATEMENT     |         |    14
```



```
|   1 |  MAT_VIEW ACCESS FULL| EMP_MV |    14
----------------------------------------------------------
Note
-----
   - dynamic sampling used for this statement (level=2)

SQL> select mitarbeiter, abteilung
  2    from emp_vw;

MITARBEITE ABTEILUNG
---------- -------------
KING       ACCOUNTINGFORD       RESEARCH
SCOTT      RESEARCH
...
14 Zeilen ausgewählt.

Ausführungsplan
----------------------------------------------------------
Plan hash value: 2614604844
----------------------------------------------------------
| Id  | Operation                      | Name     | Rows
----------------------------------------------------------
|   0 | SELECT STATEMENT               |          |    42
|   1 |  MERGE JOIN                    |          |    42
|   2 |   SORT JOIN                    |          |    14
|   3 |    MERGE JOIN                  |          |    14
|   4 |     TABLE ACCESS BY INDEX ROWID| DEPT     |     4
|   5 |      INDEX FULL SCAN           | PK_DEPT  |     4
|*  6 |     SORT JOIN                  |          |    14
|   7 |      TABLE ACCESS FULL         | EMP      |    14
|*  8 |   FILTER                       |          |
|*  9 |    SORT JOIN                   |          |     5
|  10 |     TABLE ACCESS FULL          | SALGRADE |     5

Predicate Information (identified by operation id):
---------------------------------------------------
   6 - access("E"."DEPTNO"="D"."DEPTNO")
       filter("E"."DEPTNO"="D"."DEPTNO")
   8 - filter("E"."SAL"<="S"."HISAL")
   9 - access(INTERNAL_FUNCTION("E"."SAL")>=
              INTERNAL_FUNCTION("S"."LOSAL"))
       filter(INTERNAL_FUNCTION("E"."SAL")>=
         INTERNAL_FUNCTION("S"."LOSAL"))
```

Listing 10.14 Vergleich der Ausführungspläne

Der Unterschied ist offensichtlich: Die erste Auswertung hatte wenig mehr zu tun, als einen *Full Table Scan* auf die MV auszuführen, während die zweite Auswertung über alle beteiligten Tabellen gehen musste. Hier ist die höhere Aktualität der zweiten Auswertung mit einem relativ hohen Aufwand erkauft, der bei der MV zumindest nicht in Form von Rechenzeit und eventuellen Problemen mit der Lesekonsistenz anfällt, wohl aber natürlich mit einem erhöhten Festplattenplatzverbrauch und einer gröber granulierten Aktualität der Abfrage.

Optionen für die Aktualisierung

Die Aktualisierung der MV kann zunächst einmal vollständig (`complete`) oder inkrementell (`fast`) sein. Bei der inkrementellen Aktualisierung werden lediglich die Daten, die sich seit dem letzten Mal geändert haben, neu hinzugefügt bzw. geändert oder gelöscht. Natürlich ist bei großen MVs diese Option hochgradig charmant und wird daher nach Möglichkeit auch angestrebt. Allerdings müssen dabei komplexe Probleme aus dem Weg geräumt werden, von denen ich einige im weiteren Verlauf skizzieren werde. Dann kann die Aktualisierung zeitlich auf Verlangen (`on demand`) oder aber durch eine Transaktion auf eine der Tabellen der SQL-Abfrage (`on commit`) gestartet werden. Gerade die zweite Option ist natürlich wiederum sehr interessant, birgt aber auch komplexe Probleme, deren Lösung man sich erst zutrauen sollte, wenn man das Problem komplett verstanden hat.

Zunächst einmal zu den Voraussetzungen für ein inkrementelles Refresh der MV. Das Problem ist komplex, denn es muss auf irgendeine Weise ermittelt werden, welche Zeilen einer Tabelle sich im Vergleich zum letzten Lesezeitpunkt verändert haben. Dafür stehen bei Oracle eine Reihe verschiedener Technologien zur Verfügung, doch wird im Zusammenhang mit der MV ein sogenanntes *Materialized View Log* eingesetzt. Dieses Log ist ein kleines Datenbankobjekt, in dem die Datenbank die Veränderungen einer Tabelle protokolliert. Benötigt eine MV Daten aus mehreren Basistabellen und soll eine inkrementelle Aktualisierung erreicht werden, muss für jede Basistabelle ein solches MV-Log erstellt werden. Die verschiedenen MVs, die sich auf diese Tabelle beziehen, registrieren in dem MV-Log ihr Interesse an diesen Daten. Erst, wenn die letzte MV ältere Einträge dieses MV-Logs gelesen hat, können diese entfernt werden. Nebenbei: Seien Sie froh, dass Oracle dieses Problem bereits gelöst hat. Das Erkennen einer Änderung einer Tabelle ist alles andere als trivial, denken Sie nur an die Probleme der Lesekonsistenz, die dazu führen, dass Sie gewisse Datenänderungen zu einem gegebenen Zeitpunkt nicht sehen können etc. Wir können mit unserer Beispiel-MV versuchen, zu einer inkrementell aktualisierenden MV zu kommen, aber selbst dieses einfache Beispiel muss ich mir für später aufheben, denn es ist unerwartet komplex. Einfacher lässt sich die inkrementelle Aktualisierung an einem überschaubareren Beispiel demonstrieren: der View auf die Abteilung `20`. Zunächst richten wir ein MV-Log auf die Tabelle `EMP` ein, dann kann die MV erstellt werden:


```
SQL> set autotrace off
SQL> create materialized view log on emp
  2    with primary key;
Log von Materialized View wurde erstellt.

SQL> create materialized view emp_dept_20_mv
  2  refresh fast on commit
  3  as
  4  select empno, ename, job, sal, hiredate, deptno
  5    from emp
  6   where deptno = 20;
Materialized View wurde erstellt.
```

Listing 10.15 Erstellung einer inkrementell aktualisierbaren MV

Bei der Einrichtung des MV-Logs haben wir angegeben, dass wir in jedem Fall den Primärschlüssel der Tabelle `EMP` mitspeichern möchten. Dies ist die Voraussetzung dafür, dass die Datenbank erkennen kann, welche Zeile genau geändert wurde. Alternativ wäre auch die Pseudospalte `rowid` möglich gewesen, falls zum Beispiel kein Primärschlüssel vorhanden ist oder aber die Datenbank das Vorhandensein der `rowid` im Materialized View Log fordert. Lassen Sie uns noch prüfen, ob auch alles funktioniert. Wir erstellen dazu einen neuen Mitarbeiter in Abteilung `20` und sehen direkt im Anschluss nach, ob dieser auch in der MV enthalten ist:

```
SQL> insert into emp (empno, ename, job, deptno)
  2  values (8100, 'MEYER', 'ANALYST', 20);

1 Zeile wurde erstellt.

SQL> select *
  2    from emp_dept_20_mv
  3   where empno = 8100;

Keine Zeilen ausgewählt

SQL> commit;

Transaktion mit COMMIT abgeschlossen.

SQL> select *
  2    from emp_dept_20_mv
  3   where empno = 8100;
```

```
     EMPNO ENAME      JOB              SAL HIREDATE     DEPTNO
---------- ---------- --------- ---------- -------- ----------
      8100 MEYER      ANALYST                          20

1 Zeile ausgewählt.
```

Listing 10.16 Nachweis: Die MV funktioniert!

Die Daten sind in der MV erst zu sehen, wenn die Transaktion bestätigt wurde, denn genau so ist dies bei der Anlage der View definiert worden. Doch dann sind die Änderungen auch direkt in die MV übernommen worden. Zurück zu den Optionen: Bereits mit dieser einfachen Funktion können wir nun eine MV erzeugen, die zum Beispiel nur die Fehlerdaten der letzten sieben Tage einer anderen Tabelle enthält. Kommen neue Fehler hinzu, werden diese direkt auch in die MV eingefügt. Da diese Aktion im Rahmen der gleichen Transaktionsklammer wie das Einfügen des Datensatzes in der Ursprungstabelle erfolgt, haben wir aktuelle Daten in der MV. Doch, warum machen wir so etwas? Wäre es nicht einfacher, schlicht die Basistabelle anzusprechen? Tatsächlich ist die gesamte Aktion nicht so problemlos, wie wir erhoffen:

- Wir verteuern die Transaktion auf die Basistabelle, denn nun muss nicht nur das Materialized View Log, sondern auch die angehängte MV aktualisiert werden.
- In der MV haben wir eventuell keinen Zugriff auf die aktuellsten Daten, weil diese durch die aktuelle Transaktion genauso geschützt wird wie die Basistabelle, wir können die Daten also nicht »früher« sehen als in der Basistabelle.

Ist diese Option also sinnvoll? Angenommen, die MV soll, wie in unserem Beispiel, Fehlerdaten der letzten sieben Tage enthalten, soll andererseits aber aktuelle Fehler sofort anzeigen. Dann könnte die MV nächtlich neu aufgebaut werden, um alte Fehler zu löschen, und zeigte gleichzeitig aktuelle Daten, während die Basistabelle die komplette Historie zeigt. Das ist aber kein sehr überzeugendes Szenario, hier hilft ein einfacher Index auf die Fehlerzeit, um die aktuellen Fehlerzeiten aus der Basistabelle schnell zu selektieren, ohne den zusätzlichen Aufwand der MV und des MV-Logs. Wichtiger ist die Verwendung solcher MVs im Zusammenhang mit Replikationsszenarien, in denen die Daten einer Datenbank in eine zweite Datenbank übernommen werden sollen, und dies möglichst zeitnah. Auch hierfür sind aber andere, leistungsfähigere Technologien verfügbar, so dass insgesamt die Verwendung dieser Art von MV bei einfachen Tabellen seltener ist, als vielleicht zunächst gedacht.

Erweitern wir unser Beispiel und versuchen, auch die View `EMP_VW` inkrementell zu aktualisieren. Das Problem, das wir nun vor uns haben, ist leicht zu verstehen: Dadurch, dass die View Daten aus mehreren Tabellen integriert, ist nicht mehr klar, welche Konsequenzen das Ändern einer Zeile einer Tabelle auf die Daten der MV hat, denn die Herkunft der einzelnen Zeilen ist nur dann einwandfrei sicherzustellen,

wenn entweder die Primärschlüssel oder aber die `rowid` aller beteiligten Tabellen vorhanden sind. Denn nur mit dieser Information kann die Datenbank genau ermitteln, welche Teilinformationen der MV aus welcher Zeile der beteiligten Tabellen stammen. Dieses Wissen ist jedoch Voraussetzung für eine inkrementelle Aktualisierung. Das ist kein Problem beim `refresh complete`, denn dort wird ja alles verworfen und durch eine neue Version der Daten ersetzt.

So einfach die Problemstellung grundsätzlich zu verstehen ist, so komplex kann je nach konkreter Abfrage die Lösung dieses Problems sein. Hilfreich ist hierbei ein mitgeliefertes Programm der Oracle-Datenbank, das uns – ähnlich wie die View `USER_UPDATABLE_COLUMNS` – erläutert, ob eine MV inkrementell aktualisierbar ist und, falls nicht, warum nicht. Lassen Sie uns zunächst klären, warum die Abfrage der View `EMP_VW` ohne Änderung nicht inkrementell aktualisierbar ist. Als Voraussetzung benötigen wir nun die Tabelle `MV_CAPABILITIES_TABLE`, die Sie durch das Skript `UTLXMV.SQL` anlegen lassen können, wie in Abschnitt 2.1, »Aufsetzen einer Beispieldatenbank«, beschrieben. Das Programm, das uns die Möglichkeiten einer MV erläutert, ist in dem Package `dbms_mv` enthalten. Wie immer erkennen Sie an dem Präfix `dbms_`, dass es sich bei diesem Package um ein von Oracle erstelltes Package für die Arbeit am *Database Management System (dbms)*, also an der Datenbank, handelt. Innerhalb dieses Packages heißt die Prozedur passenderweise `explain_mview`. Im Gegensatz zu den anderen Aufrufen der Packages, die Sie bislang gesehen haben, ist dies jedoch keine Funktion, die aus SQL heraus aufgerufen werden könnte. Es ist vielmehr eine sogenannte Prozedur, die irgendetwas tut, aber keine Daten an die aufrufende Umgebung zurückliefert, zumindest nicht als Rückgabewert, wie das bei einer Funktion der Fall ist. Eine Prozedur wird durch die SQL-Anweisung `call` aufgerufen und füllt für uns lediglich die vorher angelegte Tabelle `MV_CAPABILITIES_TABLE`, die wir anschließend abfragen können. Wie wir aber bereits wissen, benötigen die Tabellen `EMP`, `DEPT` und `SALGRADE` jeweils ein MV-Log, damit wir überhaupt wissen, welche Zeilen der Ausgangstabelle sich geändert haben. Bei Tabelle `DEPT` ist das kein Problem, dort können wir das MV-Log ebenso erstellen wie bei Tabelle `EMP`, wo es ja bereits existiert, denn beide Tabellen haben einen Primärschlüssel. Tabelle `SALGRADE` hat jedoch keinen Primärschlüssel, daher müssen wir hier die Option wählen, ersatzweise die `rowid` zu verwenden. Die Anweisungen sehen also aus wie folgt:

```
SQL> create materialized view log on dept
  2    with primary key;
Log von Materialized View wurde erstellt.

SQL> create materialized view log on salgrade
  2    with rowid;
Log von Materialized View wurde erstellt.
```

Listing 10.17 Anweisungen zur Erzeugung der MV-Logs

Als Nächstes können wir uns nun die geplante MV erläutern lassen. Ich habe für diese MV die herkömmliche Join-Schreibweise gewählt, weil das Package (unfassbar nach 20 Jahren Unterstützung der Syntax, aber war!) mit der ANSI-Schreibweise der Joins größere Schwierigkeiten hat. Zudem habe ich, weil ich das Problem der Zuordnung der Zeilen der View zu den Ausgangstabellen vorausahne, bereits die Primärschlüsselspalten EMPNO und DEPTNO mit in die Abfrage aufgenommen:

```
SQL> call dbms_mview.explain_mview(
  2  'select e.empno, e.ename mitarbeiter,
  3          d.deptno, d.dname abteilung,
  4          m.ename manager,
  5          e.job beruf,
  6          s.grade gehaltsstufe
  7     from emp e, emp m, dept d, salgrade s
  8    where e.deptno = d.deptno
  9      and e.mgr = m.empno (+)
 10      and e.sal between s.losal and s.hisal');
Aufruf wurde abgeschlossen.

SQL> select capability_name, possible, related_text, msgtxt
  2    from mv_capabilities_table
  3   where capability_name not like 'PCT%'
  4   order by seq;
CAPABILITY_NAME           P RELATED_TE MSGTXT
------------------------- - ---------- ------------------------------------
REFRESH_COMPLETE            Y
REFRESH_FAST                N
REWRITE                     Y
REFRESH_FAST_AFTER_INSERT N S          Die SELECT-Liste verfügt nicht über
                                       die Row-IDs von allen Detail-Tabellen
REFRESH_FAST_AFTER_INSERT N SCOTT.DEPT MV-Log muss ROWID aufweisen
REFRESH_FAST_AFTER_INSERT N SCOTT.EMP  MV-Log muss ROWID aufweisen
REFRESH_FAST_AFTER_ONETAB_ N           Siehe Grund, warum REFRESH_FAST_
                                       AFTER_INSERT deaktiviert ist
REFRESH_FAST_AFTER_ANY_DML N           Siehe Grund, warum T_AFTER_ONETAB_DML
                                       deaktiviert ist
REWRITE_FULL_TEXT_MATCH    Y
REWRITE_FULL_TEXT_MATCH    Y
REWRITE_PARTIAL_TEXT_MATCH Y
REWRITE_PARTIAL_TEXT_MATCH Y
REWRITE_GENERAL            Y
REWRITE_GENERAL            Y

11 Zeilen ausgewählt.
```

Listing 10.18 Eine erste Analyse der geplanten MV

Die Abfrage auf diese Tabelle filtert einige Daten heraus, die für uns nicht von Interesse sind. Diese Optionen beschäftigen sich mit partitionierten Tabellen, einer speziellen Speicherform, die für sehr große Tabellen verwendet wird. Wenn wir die verbleibenden Zeilen durchsehen, wird klar, dass unsere erste Annahme, die MV-Logs seien mit Primärschlüsseln ausreichend ausgestattet, offensichtlich nicht stimmt. Die Datenbank fordert, dass auch dort rowid verwendet werden soll. Bauen wir diese also zunächst um:

```
SQL> drop materialized view log on emp;
Log von Materialized View wurde gelöscht.

SQL> drop materialized view log on dept;
Log von Materialized View wurde gelöscht.

SQL> create materialized view log on emp
  2    with primary key, rowid;
Log von Materialized View wurde erstellt.

SQL> create materialized view log on dept
  2    with primary key, rowid;
Log von Materialized View wurde erstellt.
```

Listing 10.19 Neuanlage der MV-Logs für die Tabellen EMP und DEPT

Also lassen Sie uns nun sehen, welche Auswirkung dies hatte:

```
SQL> delete from mv_capabilities_table;
11 Zeilen wurden gelöscht.

SQL> call dbms_mview.explain_mview(
  2  …);
Aufruf wurde abgeschlossen.

SQL> select capability_name, possible,
  2         related_text, msgtxt
  3    from mv_capabilities_table
  4   where capability_name not like 'PCT%'
  5   order by seq;

CAPABILITY_NAME           P RELATED_TE MSGTXT
------------------------- - ---------- --------------------
REFRESH_COMPLETE          Y
REFRESH_FAST              N
REWRITE                   Y
```

```
REFRESH_FAST_AFTER_INSERT  N S          Die SELECT-Liste verfügt nicht über
                                        die Row-IDs von allen Detail-Tabellen
REFRESH_FAST_AFTER_ONETAB_ N            Siehe Grund, warum REFRESH_FAST_
                                        AFTER_INSERT deaktiviert ist
REFRESH_FAST_AFTER_ANY_DML N            Siehe Grund, warum REFRESH_FAST_AFTER
                                        _ONE_TAB_DML deaktiviert ist
REWRITE_FULL_TEXT_MATCH     Y
REWRITE_PARTIAL_TEXT_MATCH  Y
REWRITE_GENERAL             Y

9 Zeilen ausgewählt.
```

Listing 10.20 Zweiter Versuch: Nun fehlen Spalten.

Nun fordert die Datenbank, dass statt der Primärschlüssel die `rowid` der beteiligten Zeilen in die View mit aufgenommen werden sollen. Also gut, machen wir auch dies:

```
SQL> delete mv_capabilities_table;
11 Zeilen wurden gelöscht.

SQL> call dbms_mview.explain_mview(
  2  'select e.rowid e_rowid, e.ename mitarbeiter,
  3          d.rowid d_rowid, d.dname abteilung,
  4          m.rowid m_rowid, m.ename manager,
  5          e.job beruf,
  6          s.rowid s_rowid, s.grade gehaltsstufe
  7     from emp e, emp m, dept d, salgrade s
  8    where e.deptno = d.deptno
  9      and e.mgr = m.empno (+)
 10      and e.sal between s.losal and s.hisal');
Aufruf wurde abgeschlossen.

SQL> select capability_name, possible,
  2         related_text, msgtxt
  3    from mv_capabilities_table
  4   where capability_name not like 'PCT%'
  5   order by seq;

CAPABILITY_NAME               P RELATED_TE MSGTXT
----------------------------- - ---------- ---------------
REFRESH_COMPLETE              Y
REFRESH_FAST                  Y
REWRITE                       Y
```



```
REFRESH_FAST_AFTER_INSERT      Y
REFRESH_FAST_AFTER_ONETAB_DML  Y
REFRESH_FAST_AFTER_ANY_DML     Y
REWRITE_FULL_TEXT_MATCH        Y
REWRITE_PARTIAL_TEXT_MATCH     Y
REWRITE_GENERAL                Y

9 Zeilen ausgewählt.
```

Listing 10.21 Weitere Änderung: Nun klappt's auch mit dem Nachbarn ...

Nun scheint die Datenbank zufrieden zu sein. Beachten Sie bitte, dass wir für beide Alias auf die Tabelle `EMP` jeweils die `rowid` mitschleppen müssen. Das ist auch logisch, denn der Manager eines Mitarbeiters hat ja eine andere `rowid` als der Mitarbeiter selbst. Zudem müssen die `rowid`-Spalten selbstverständlich ein Spaltenalias erhalten, denn ansonsten wären die Spaltenbezeichner zum einen mehrfach vorhanden, zum anderen ist der Bezeichner `rowid` nicht erlaubt, weil es sich um ein geschütztes Wort in (Oracle-)SQL handelt. Nachdem wir nun die Anforderungen der Datenbank erfüllt haben, können wir die MV mit dieser Anweisung auch anlegen:

```
SQL> drop materialized view emp_mv;
Materialized View wurde gelöscht.

SQL> create materialized view emp_mv
  2  refresh fast on commit
  3  as
  4  select e.rowid e_rowid, e.ename mitarbeiter,
  5         d.rowid d_rowid, d.dname abteilung,
  6         m.rowid m_rowid, m.ename manager,
  7         e.job beruf,
  8         s.rowid s_rowid, s.grade gehaltsstufe
  9    from emp e, emp m, dept d, salgrade s
 10   where e.deptno = d.deptno
 11     and e.mgr = m.empno (+)
 12     and e.sal between s.losal and s.hisal;

Materialized View wurde erstellt.
```

Listing 10.22 Nach der ganzen Arbeit: Erzeugung der MV

Vergessen Sie bitte nicht, die ältere Version der `EMP_MV` (`refresh complete on demand`) zu löschen und durch die neue Version zu ersetzen. Etwas unschön ist nun, dass die MV all diese `rowid` enthält. Um dieses Problem elegant zu umschiffen, können wir nun unsere ursprüngliche View `EMP_VW` auf die MV umleiten:

```
SQL> create or replace view emp_vw as
  2  select mitarbeiter, abteilung, manager,
  3         beruf, gehaltsstufe
  4    from emp_mv;
View wurde erstellt.
```

Listing 10.23 Umgearbeitete View EMP_VW

Nun haben wir den Vorteil der MV und keinen Nachteil durch Spalten, die uns eigentlich nicht interessieren.

Wieder haben wir also die Frage zu klären, ob der Aufwand gerechtfertigt ist. Gibt es für diese MV sinnvolle Einsatzszenarien? Ich erinnere mich daran, eine solche MV einmal verwendet zu haben, um folgendes Problem zu lösen: Ein Bericht hatte Daten aus insgesamt 13 Tabellen zusammengestellt. Der Optimizer hatte echte Probleme, einen guten Ausführungsplan zu finden. Die Folge: Die Ausführungszeit ging in Richtung 5 Minuten. Bei der Analyse der Auswertung viel auf, dass sieben Tabellen dafür zuständig waren, Stammdaten für den Bericht aufzuarbeiten. Diese Stammdaten ändern sich zwar, aber selten. Daher habe ich mich entschlossen, diese sieben Tabellen in einer MV zusammenzufassen und als `refresh fast on commit` zu vereinbaren. Das hat zwar etwas Mühe gekostet, weil auch hier einige Voraussetzungen erfüllt werden mussten, um die komplexe Abfrage dynamisch aktualisierbar zu machen, doch war das Ergebnis die Mühe wert: Die Abfrage konnte nun, um die MV erweitert, in weniger als 1 Sekunde erledigt werden. Da andererseits die Stammdaten nur selten aktualisiert werden, ist die Verteuerung der Schreibaktion in diesem Fall hinnehmbar, durch die Vereinbarung als `refresh fast on commit` war zudem sichergestellt, dass eine Änderung der Stammdaten direkt auch eine Aktualisierung der MV nach sich zog. Aber Achtung: Die MV erkennt, wenn sich die Stammdaten ändern, nicht aber, wenn diese ungültig werden. Dies kann zum Beispiel dann der Fall sein, wenn die Laufzeit einer Berechtigung oder eines Vertrags durch eine Spalte limitiert ist. Sobald dieses Datum überschritten wird, ist der Vertrag abgelaufen. Eine View kann, da sie über `sysdate` filtern kann, solche automatischen Abläufe erkennen, eine MV nur, wenn sie zusätzlich einmal am Tag komplett aktualisiert wird. Sie sehen, dass die Verwendung unerwartet steinig ist. Dies dürfte auch der Grund sein, warum nicht einfach alle Views im Hintergrund materialisiert werden.

Ähnliche logische Probleme erwarten Sie, wenn Sie Gruppenfunktionen in der Abfrage verwenden. Auch das ist verständlich: Wie ändert sich der Durchschnittswert der Gehälter einer Abteilung, wenn ein Mitarbeiter dieser Abteilung 100 Taler mehr verdient? Das können wir nur abschätzen, wenn wir wissen, wie viele Mitarbeiter in dieser Abteilung arbeiten und wie hoch die Summe der Gehälter ist. Es kann also sein, dass wir, um eine solche Abfrage inkrementell aktualisieren zu können, weitere Gruppenfunktionen wie `count(*)` oder `sum(sal)` mitführen müssen, obwohl

diese für die eigentliche Abfrage nicht benötigt werden. Eine erschöpfende Diskussion der logischen Abhängigkeiten der Gruppenfunktionen führt hier aber zu weit, dazu möchte ich gern auf die Online-Dokumentation verweisen. Nähere Angaben zu MVs und den damit zusammenhängenden Problemen finden Sie in einem sehr guten (aber auch englischsprachigen) PDF von Oracle: dem *Data Warehousing Guide*. In Version 12.2 finden Sie in den Kapiteln 5, 6, 7, 9, 11 und 12 mehr als genug Informationen zum Anlegen einfacher und komplexer MVs bis hin zur Diskussion auch exotischster Anwendungen.

10.4.3 Grenzen der Aktualisierung

Als letzten Abschnitt dieses Kapitels möchte ich Ihnen gern noch einige Fallstricke aufzeigen, die mit MVs verbunden sind. Die meisten dieser Fallstricke beziehen sich auf die vielleicht interessanteste Variante, die inkrementelle Aktualisierungsvariante. MVs werden häufig eingesetzt, um Daten zwischen Datenbanken zu *replizieren* (mit diesem Ausdruck wird die mehrfache, synchronisierte Speicherung von Daten an mehreren Stellen bezeichnet). Stellen Sie sich zum Beispiel vor, dass eine Produktionsdatenbank Teile der Daten in eine Berichtsdatenbank replizieren soll. Hierfür wären MVs wie geschaffen. Insbesondere wäre natürlich die schnelle inkrementelle Replikation in diesem Zusammenhang sehr hübsch. Doch Achtung: Technisch gesehen ist die Replikation von Daten auf eine zweite Datenbank innerhalb einer Transaktion eine sogenannte *verteilte Transaktion*. Vereinfacht gesagt hat eine der beiden Datenbanken den Hut auf und leitet die Transaktion. Wenn die Daten in beide Datenbanken geschrieben wurden, fragt nun die leitende Datenbank, ob eine Bestätigung der Transaktion mit `commit` in Ordnung sei. Schön, wenn die zweite Datenbank nun zustimmt. Nicht schön, falls nicht! Wenn also zum Beispiel die zweite Datenbank aufgrund irgendeines Problems (Netzwerkproblem, die Datenbank ist nicht verfügbar, Windows ist installiert … ☺) nicht antwortet, steht die Transaktion still und damit auch die Anwendung, die die Daten schreiben möchte. Solche Dinge möchten Sie in Produktionsumgebungen nicht erleben. Daher sollten Sie an diese eigentlich verlockende Variante nur mit großer Vorsicht gehen. Unproblematischer erscheint, wenn die Transaktion innerhalb der gleichen Datenbank abläuft, aber auch hier lauern Gefahren, zum Beispiel die, dass die MV physikalisch keinen weiteren Speicher anfordern kann oder Ähnliches. Zudem müssen Sie berücksichtigen, dass Arbeiten, die durchgeführt werden, immer länger dauern als Arbeiten, die nicht durchgeführt werden. Im Klartext: Die Einbindung einer MV in einen Transaktionsablauf bedeutet immer auch zusätzliche Arbeit für die Datenbank – spürbare Arbeit, die, je nach Zusammenhang, einen solchen Ansatz von vornherein zum Scheitern verurteilen kann.

Dann müssen Sie sich selbstverständlich noch vor einer anderen Gefahr hüten, nämlich vor dem Problem, nach dem Kauf eines Hammers nun jedes Problem für einen

Nagel zu halten. MVs sind nur *eine* Strategie zur Datenreplikation. Oracle verfügt für diesen Zweck über eine ganze Reihe von Strategien, die neuerdings (ab Version 12c) unter dem Begriff *GoldenGate* zusammengefasst werden. Ich möchte diese Diskussion nicht weiterführen, da sie sehr weit in den Bereich der Datenbankadministration reicht, doch stehen für ein gegebenes Problem meistens viele Technologien bereit, und die schlechteste Grundlage für eine informierte Entscheidung ist, nicht informiert zu sein. Daher sollten komplexe Probleme erst nach eingehender Beratung mit Leuten, die sich auskennen, und nach einer sorgfältigen Abwägung der Vor- und Nachteile angegangen werden. Weiter gehende Informationen zu diesen Themen finden Sie unter dem Stichwort *GoldenGate* bei Oracle sowie in den *Data Guard Concepts and Adminstration*, falls Ihr Ziel ist, eine parallel laufende Datenbank zur Datensicherung zu betreiben.

Ein weiteres Problemfeld stellen inkrementelle MVs dar, die Aggregationen enthalten. Diese MVs sind typischerweise nichts für transaktionsorientierte Datenbanken, denn eine Aktualisierung einer Zeile der Basistabellen der MV kann durch die Aktualisierung der MV ausgebremst werden. Dazu vielleicht ein Beispiel: Eine MV aggregiert 10.000 Einzelwerte zu einer Zeile. Nun werden an den Einzelwerten Änderungen vorgenommen. Natürlich werden all diese Änderungen Konsequenzen für die *eine* aggregierte Zeile der MV haben. Nun haben wir aber das Problem, dass eventuell Hunderte Benutzer, die die Änderungen an den Basistabellen vornehmen, innerhalb ihrer Transaktion alle diese eine aggregierte Zeile der MV aktualisieren möchten. Diese wird nun zum Flaschenhals für die gesamte Anwendung, weil natürlich immer nur ein einzelner Benutzer zur gleichen Zeit diese Zeile verändern kann. Probleme dieser Art müssen sorgfältig analysiert werden, um keine negativen Auswirkungen auf die gesamte Anwendung zu verursachen.

Ein weiteres Problem, das insbesondere die regelmäßige Aktualisierung der MV über einen Job betrifft, ist, dass diese Aktualisierung eventuell nicht durchgeführt wird. Hierfür können die Gründe vielfältig sein. Vielleicht ist der Job aus irgendwelchen Gründen auf einen Fehler gelaufen und hat sich deaktiviert, vielleicht ist der Job aber auch durch eine Unaufmerksamkeit deaktiviert oder sogar ganz gelöscht worden. Die Fehlermöglichkeiten sind hier vielfältig und sollten regelmäßig überwacht werden. Um zu sehen, ob alles so weit in Ordnung ist, können Sie die View `USER_JOBS` oder, je nach Einrichtung der MV, auch `USER_SCHEDULER_JOBS` verwenden, die Ihnen anzeigt, welche Jobs für Sie eingerichtet sind, wann diese zuletzt gelaufen sind und wann sie das nächste Mal zu starten beabsichtigen. Außerdem wird in dieser View darauf hingewiesen, ob ein Fehler aufgetreten ist oder nicht, leider aber auch nicht viel mehr. MVs sollten also regelmäßig überwacht werden, damit Sie nicht fälschlicherweise von aktuellen Daten ausgehen.

Kapitel 26

Datenmodellierung von Datum und Zeit

In diesem Kapitel beschäftigen wir uns mit der Datenmodellierung im Umfeld von Datenbanken, die Informationen über Zeiträume historisierend speichern müssen. Diese Erweiterung des Begriffs des Loggings von Datenänderungen hält wichtige Strategien bereit, die wir uns in diesem Kapitel ansehen werden.

Dieses Kapitel hat zwei große Bereiche, einmal die Speicherung von Datumsbereichen und dann die Arbeit mit historisierenden und bitemporalen Datenmodellen. Beiden Problembereichen ist gemeinsam, dass eine eindeutig beste Lösung nicht existiert, die verschiedenen Varianten sind sehr unterschiedlich, was ihre Leistungsfähigkeit angeht. Daher muss im Projekt nach gründlicher Überlegung entschieden werden, wie mit diesem schwierigen Feld umzugehen ist.

26.1 Datumsbereiche

Eine der häufigsten Anwendungen von Datumsangaben in Datenbanken besteht darin, einen Datumsbereich zu beschreiben. Damit ist gemeint, dass durch ein Start- und ein Enddatum der Gültigkeitszeitraum eines Faktums beschrieben wird. Beispiele finden sich viele: Mitarbeiter `SMITH` hat von … bis … in Abteilung `30` gearbeitet, ein Medikament wurde über eine Zeitdauer eingenommen, ein Artikel war über eine Zeitdauer einer Produktgruppe zugeordnet … Ich möchte Ihnen gern einige der weiter verbreiteten Datenmodelle zeigen, die Vor- und Nachteile benennen und Ihnen ein Gefühl für die SQL-Abfragen geben, die Sie für häufige Fragestellungen benötigen.

26.1.1 Speicherung von Datumsbereichen mit zwei Zeitpunkten

Die einfachste Form, solche Informationen in Datenbanken zu speichern, besteht darin, einfach ein Startdatum und ein Enddatum in zwei Spalten zu speichern. Oft haben diese Spalten Bezeichnungen wie `DATE_FROM` und `DATE_TO` oder ähnlich. Diese Art der Speicherung ist unmittelbar einsichtig, hat aber auch einige Nachteile: Stellen

26

Sie sich vor, ein neuer Zustand der Daten soll gespeichert werden. Nun reicht eine insert-Anweisung allein nicht mehr aus, denn das Enddatum des momentan gültigen Intervalls muss nun auf 1 Sekunde vor dem Start des neuen Intervalls geändert werden. Es hat sich eingebürgert, bei Spalten des Typs date diese Sekunde zu verwenden, denn sie ist resistent gegenüber Problemen, die aufgrund des Uhrzeitanteils einer Abfrage auftauchen und die wir schon verschiedentlich besprochen haben. Allerdings ist diese zusätzliche update-Anweisung ein Aufwand, der etwas unangenehm ist. Verschlimmert wird dieses Problem noch dadurch, dass es eventuell möglich ist, dass in ein bestehendes Intervall ein anderes Intervall eingefügt werden soll. Ist also beispielsweise ein Fakt vom 01.01. bis zum 31.12. eines Jahres gültig und soll nun dokumentiert werden, dass sich dieses Faktum vom 15.05. bis zum 01.08. in einem anderen Zustand befunden hat, müssen Sie logisch aufwendige Operationen an der Tabelle vornehmen, um diese Intervalle korrekt abzugrenzen. Das ist mit einer SQL-Anweisung allein nicht mehr zu schaffen, es muss entweder händisch in mehreren SQL-Anweisungen oder durch ein Programm gesteuert werden. Die Logik, solche Abgrenzungen stets logisch korrekt durchzuführen, ist alles andere als trivial.

Ein weiteres Problem dieses Datenmodells besteht darin, dass es keine einfache Möglichkeit gibt, zu verhindern, dass sich Intervalle überlappen können. Vielleicht ist eine solche Überlappung von Intervallen in Ihrem Geschäftsfall ja erlaubt, falls jedoch nicht, haben Sie kaum eine Chance, dies in der Datenbank zu verhindern. Das gilt leider auch für die Programmierung in der Anwendung, denn diese ist normalerweise zu weit von der Datenbank weg, um sicher verhindern zu können, dass es zu solchen Überlappungen kommt. Eine mögliche Lösung bestünde in der Programmierung eines eigenen Indextyps für solche Zwecke, aber das ist selbst für Datenbankprogrammierer schon hohe Schule. Lassen Sie uns dennoch einmal einen Blick auf dieses Datenmodell werfen. Zunächst stellen wir uns eine Tabelle vor wie folgt:

```
create table time_range_test (
  id number,
  action varchar2(25 char),
  valid_from date,
  valid_to date);
```

Listing 26.1 Eine Testtabelle zur Demonstration von Datumsbereichen

Im Skript zum Buch habe ich eine insert-Anweisung beigefügt, die folgende Daten erzeugt:

```
ID  ACTION           VALID_FROM          VALID_TO
--- ---------------- ------------------- -------------------
1   Start of project  15.02.2018 19:30:00  16.02.2018 09:30:59
1   Kick off          16.02.2018 09:31:00  16.02.2018 11:59:59
1   Phase 1           16.02.2018 12:00:00  03.03.2018 07:59:59
1   Phase 2           03.03.2018 08:00:00  31.12.2999 00:00:00
2   Start of project  19.02.2018 19:30:00  20.02.2018 09:30:59
2   Kick off          20.02.2018 09:31:00  20.02.2018 11:59:59
2   Phase 1           20.02.2018 12:00:00  01.03.2018 07:59:59
2   Phase 2           01.03.2018 08:00:00  31.12.2999 00:00:00
3   Start of project  18.02.2018 19:30:00  19.02.2018 09:30:59
3   Kick off          19.02.2018 09:31:00  19.02.2018 11:59:59
3   Phase 1           19.02.2018 12:00:00  18.03.2018 07:59:59
3   Phase 2           18.03.2018 08:00:00  31.12.2999 00:00:00
12 Zeilen ausgewählt.
```

Listing 26.2 Die Beispieldaten

Wir haben also drei Projekte, die wir in aufeinanderfolgenden Intervallen durch jeweils vier Status begleiten. Beachten Sie bitte, dass ich in meiner Version zur Speicherung der Tatsache, dass ein Faktum aktuell noch gilt, ein Abschlussdatum gewählt habe, das weit in der Zukunft liegt. Das ist relativ üblich, aber nicht notwendigerweise so. Alternativ sehe ich hier häufig auch null-Werte. Wird ein null-Wert verwendet, verbietet sich die einfache Prüfung über between, es sei denn, Sie definieren einen Ersatzwert für null in der between-Anweisung. Beachten Sie aber als Nachteil dieser Variante, dass eine solche Suche sicher nicht indiziert durchgeführt werden kann: Zum einen steht ein null-Wert nicht im Index, zum anderen ist sysdate als möglicher Ersatzwert nicht deterministisch und kann daher auch nicht funktionsbasiert indiziert werden. Natürlich können Sie immer noch andere Spalten indizieren, aber diese Spalten eignen sich aufgrund des null-Wertes nicht.

Starten wir zunächst mit einfachen Abfragen. Welche Projekte sind am 03.03.2018 in Phase 2? So etwas ist ganz einfach:

```
SQL> select to_char(id) id, action, valid_from, valid_to
  2    from time_range_test
  3   where date '2018-03-03' between valid_from and valid_to
  4     and action = 'Phase 2';

ID  ACTION           VALID_FROM          VALID_TO
--- ---------------- ------------------- -------------------
2   Phase 2          01.03.2018 08:00:00 31.12.2999 00:00:00
```

Listing 26.3 Eine einfache Abfrage mit Datumsbereichen

Diese Art der Datenmodellierung finden Sie häufig in Datenmodellen, die historisierend sind, das heißt, bei denen man auf einfache Weise nachfragen kann, in welchem Zustand sich die Daten zu einem Stichtag befunden haben. Ebenfalls häufig ist dieses Datenmodell für Abfragen, die in einem Bericht nachzeichnen müssen, welcher Arti-

kel von wann bis wann einer Artikelgruppe angehört hat, wie dies in Berichtsdatenbanken fast immer der Fall ist. Das Szenario ist hier: Wenn ein Produkt bis zum 15.05. zur Produktgruppe A gehört hat, ab dann aber zur Produktgruppe B, muss bei einer Monatsauswertung des Monats Mai der Umsatz dieses Produkts abgegrenzt werden: Die Umsätze bis zum 15.05. einschließlich werden Produktgruppe A zugeschlagen, anschließend Produktgruppe B. Zumindest ist dies das häufigste Szenario.

Andere Auswertungen sind ebenso einfach: So könnten wir uns fragen, wie lange wir durchschnittlich in Phase 1 verharrt haben, bevor wir in Phase 2 gelangt sind:

```
SQL> select avg(valid_to - valid_from) dauer
  2    from time_range_test
  3   where action = 'Phase 1';

     DAUER
----------
16,8333218
```

Listing 26.4 Eine erste Analyse

Schöner wird die Ausgabe, wenn wir aus der Zahl ein Intervall erzeugen:

```
SQL> select numtodsinterval(avg(valid_to - valid_from), 'day') dauer
  2    from time_range_test
  3   where action = 'Phase 1';

DAUER
-----------------------------
+000000016 19:59:59.000000000
```

Listing 26.5 Ausgabe der durchschnittlichen Dauer als Intervall

Die hier verwendete Datumsarithmetik ist sicher leicht nachvollziehbar. Problematisch wird die Auswertung allerdings, wenn wir uns fragen, wie viele Tage im März wir eigentlich in Phase 2 zugebracht haben. Das Problem ist die Abgrenzung auf den Monat März. Vielleicht ist das noch möglich, wenn wir konkret nur nach einem Monat fragen, aber sobald wir versuchen, dies auf alle Monate eines Jahres zu übertragen, wird die Auswertung doch recht trickreich. Beginnen wir damit, zunächst nur nach dem März zu fragen. Damit ich später leichteres Spiel habe, werde ich in einer inneren Abfrage zunächst einmal den Beginn und das Ende des Monats erzeugen, damit ich die hierfür nötigen Formeln nicht häufiger wiederholen muss. Um diese Inner View ordne ich dann die eigentliche Abfrage an:

```
SQL> select sum(least(m.month_end, t.valid_to) -
  2             greatest(m.month_start, t.valid_from)) days
```



```
  3    from time_range_test t
  4    join (select date '2018-03-01' month_start,
  5                 date '2018-04-01' - interval '1' second month_end
  6            from dual) m
  7      on valid_from <= month_end
  8      or valid_to >= month_start
  9   where t.action = 'Phase 2';

      DAYS
----------
72,9999653
```

Listing 26.6 Wie viele Tage wurden im März für Phase 2 geleistet?

Hier benötigen wir eine Funktion, um die Daten voneinander abzugrenzen. Statt einer längeren case-Anweisung habe ich die Funktionen least und greatest verwendet. Damit erreiche ich, dass beim Beginn der Phase stets das spätere Ereignis verwendet wird: Monatsbeginn oder Projektbeginn. Analog verwende ich das frühere Ereignis beim Ende des Intervalls: Projektende oder Monatsende. Dann ist eine Join-Bedingung erforderlich, die dafür Sorge trägt, dass die Phase den infrage stehenden Monat überhaupt berührt. In unserem Beispiel berühren alle Projekte den März, daher würde in diesem Fall nicht einmal auffallen, wenn die Join-Bedingung fehlte, doch ist die bei weiteren Auswertungen dringend erforderlich. Achten Sie auf die Konstruktion dieser Join-Bedingung, denn es können mehrere Fälle auftreten: Ein Intervall liegt vollständig im Monat, eine der Intervallgrenzen liegt außerhalb des Monats, beide Intervallgrenzen liegen außerhalb des Monats, beinhalten diesen aber. Bei Fragen wie diesen hilft mir da und dort ein einfaches Stück Papier, auf dem ich dann die Rahmenbedingungen skizziere (Abbildung 26.1).

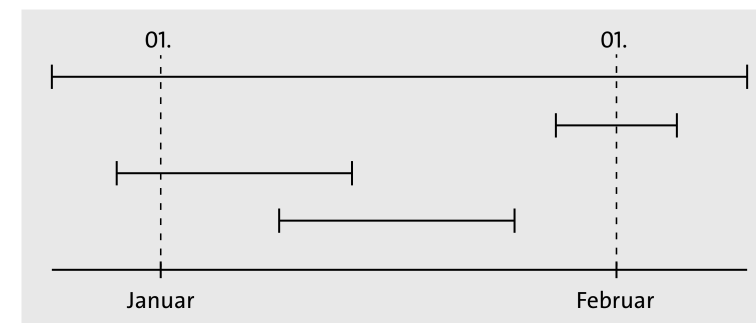

Abbildung 26.1 Improvisierte Visualisierung der Intervallgrenzen

- Was gilt für den Beginn aller dieser Intervalle?
  Sie sind kleiner als das Monatsende.

- Was gilt für das Ende all dieser Intervalle?
  Sie sind größer als der Monatsbeginn. Sind beide Bedingungen erfüllt, berührt das Intervall den Monat.
- Welchen Zeitpunkt müssen wir als Startdatum nehmen?
  Das Maximum aus Intervallstart und Monatsstart. Analog gilt für das Ende, dass wir das Minimum aus Intervallende und Monatsende verwenden müssen.

Erweitern wir nun unsere Abfrage, um zu sehen, wie viele Tage pro Phase in den jeweiligen Monaten geleistet wurden. Das erste Problem: Wir benötigen nun eine Tabelle mit den infrage kommenden Monaten. Diese Tabelle haben wir aber nicht. Daher erstelle ich diese Tabelle innerhalb von SQL dynamisch. Anschließend sollte die Abfrage weitgehend unverändert funktionieren. Zunächst die Erzeugung der Datumsangaben:

```
SQL>  select add_months(date '2018-01-01', level - 1) month_start,
  2          last_day(
  3            add_months(date '2018-01-01', level - 1))
  4          + interval '23:59:59' hour to second month_end
  5     from dual
  6   connect by level < 4;

MONTH_START         MONTH_END
------------------- -------------------
01.01.2018 00:00:00 31.01.2018 23:59:59
01.02.2018 00:00:00 28.02.2018 23:59:59
01.03.2018 00:00:00 31.03.2018 23:59:59
```

Listing 26.7 Erzeugung mehrerer Monate aus einer Abfrage

Sie können eine solche Abfrage als View hinterlegen (schön gerade auch dann, wenn Sie immer zum Beispiel das Zeitintervall von Beginn letzten Jahres bis Ende dieses Jahres verwenden, weil Sie dann das Startdatum auch noch aus `sysdate` errechnen können), oder aber Sie schaffen sich eine Tabelle, die diese Daten für die nächsten 20 Jahre gespeichert vorhält. Das ist unter dem Gesichtspunkt der Datenmenge selbst dann kein Problem, wenn Sie für jeden Tag der nächsten 20 Jahre eine Zeile anlegten, denn diese Tabelle hat eine Größenordnung von 7.300 Zeilen, was für eine Tabelle wirklich nicht viel ist. Ein Beispiel für eine solche Tabelle besprechen wir im nächsten Abschnitt. Egal wie, die Daten benötigen wir, um die Abgrenzung gegen diese Monate durchführen zu können. Die Abfrage aus Listing 26.7 hinterlege ich in der Datenbank als View mit dem Namen `MONTHS`, damit ich den Code nicht zu wiederholen brauche. Die generalisierte Abfrage ist nun um den Monat erweitert, gruppiert die Zeiten nach Monat und stellt sie dar:


```
SQL> select month_start,
  2         round(sum(least(m.month_end, t.valid_to)
  3                   - greatest(m.month_start, t.valid_from)),
  4           1) days
  5    from time_range_test t
  6    join months m on valid_from between month_end month_start
  7   where t.action = 'Phase 1'
  8   group by month_start
  9   order by month_start desc;

MONTH_START               DAYS
------------------- ----------
01.03.2018 00:00:00         20
01.02.2018 00:00:00       30,5
```

Listing 26.8 Generalisierte Abfrage

Zur Kontrolle können wir ausgeben, wie die Intervalle nun den einzelnen Monaten zugeschlagen werden:

```
SQL> select t.id, t.valid_from, t.valid_to,
  2         greatest(m.month_start, t.valid_from) interval_start,
  3         least(m.month_end, t.valid_to) interval_end
  4    from time_range_test t
  5    join months m on valid_from between month_end and month_start
  6   where t.action = 'Phase 1'
  7   order by t.id, interval_start;

 ID VALID_FROM  VALID_TO    INTERVAL_START INTERVAL_END
--- ----------- ----------- -------------- --------------------
  1 16.02 12:00 03.03 07:59 16.02 12:00   28.02 23:59 23:59:59
  1 16.02 12:00 03.03 07:59 01.03 00:00   03.03 07:59 23:59:59
  2 20.02 12:00 01.03 07:59 20.02 12:00   28.02 23:59 23:59:59
  2 20.02 12:00 01.03 07:59 01.03 00:00   01.03 07:59 23:59:59
  3 19.02 12:00 18.03 07:59 19.02 12:00   28.02 23:59 23:59:59
  3 19.02 12:00 18.03 07:59 01.03 00:00   18.03 07:59 23:59:59
6 Zeilen ausgewählt.
```

Listing 26.9 Kontrollauswertung zur Zuweisung der Intervalle

Dieses Prinzip ist sehr typisch für die Arbeit mit Datumsbereichen. Dennoch ist es immer wieder kompliziert, solche Abfragen zu erstellen, weil Sie stets von Neuem mit den Fragestellungen der korrekten Abgrenzung von Datumsbereichen konfrontiert werden. Das Beispiel oben hat im Übrigen noch eine Schwäche, falls die Auswer-

tung auch den aktuellen Monat betreffen soll. Sehen Sie diese Schwäche? Sie liegt darin, dass wir, wenn wir im aktuellen Monat, sagen wir am 18.03., die Abfrage stellen, dennoch die Tage bis zum letzten des Monats in die Auswertung übernehmen, denn falls ein Enddatum sehr weit in der Zukunft liegt, sehen wir als Minimum von Intervallende und Monatsende das Monatsende. Wir müssen dafür Sorge tragen, dass wir höchstens bis sysdate nach »vorne« gucken, falls wir die Anzahl der tatsächlich investierten Tage berücksichtigen möchten. Nur, wie erreichen wir dies auf einfachste Weise? Es ist überraschend einfach: Sie müssen lediglich sysdate als dritten Parameter in die least-Funktion aufnehmen, egal, an welcher Stelle. Nun wird der kleinste der drei Werte (sysdate, valid_to, month_end) geliefert, und das Problem ist behoben. Dennoch: So etwas ist schon hinterhältig und muss gut getestet werden.

Eine leichte Abwandlung des Datenmodells oben haben wir, wenn kein weit in der Zukunft liegendes Enddatum, sondern ein null-Wert verwendet wird. Aber letztlich können Sie sich bereits vorstellen, dass wir für den null-Wert über eine coalesce-Funktion lediglich sysdate einsetzen, was in den meisten Fällen wohl der korrekte Ersatzwert wäre. Daher denke ich, dass wir auf ein Beispiel verzichten können.

Oracle bietet im Übrigen für die Prüfung, ob sich Datumsintervalle überlappen, eine weitere Möglichkeit an. Dies ist der overlaps-Operator, der zwei Intervalle miteinander vergleicht. Der Operator liefert true zurück, wenn sich die Intervalle überlappen, und false im anderen Fall. Im folgenden Beispiel zeige ich Ihnen einmal seine Verwendung:

```
SQL> select month_start,
  2         round(
  3           sum(
  4             least(m.month_end, t.valid_to)
  5           - greatest(m.month_start, t.valid_from)),
  6         1) days
  7    from time_range_test t
  8    join months m
  9      on (valid_from, valid_to) overlaps (month_start, month_end)
 10   where t.action = 'Phase 1'
 11   group by month_start
 12   order by month_start desc;

MONTH_ST       DAYS
-------- ----------
01.03.10         20
01.02.10       30,5
```

Listing 26.10 Verwendung des (geheimen!) OVERLAPS-Operators

Der overlaps-Operator hat den großen Vorteil, Teil der ISO-Spezifikation zu sein, und den großen Nachteil, nicht Teil der Oracle-SQL-Spezifikation zu sein, denn er ist undokumentiert. Daher kann für diesen Operator auch keine Empfehlung ausgesprochen werden. Sollten Sie ihn dennoch benutzen wollen, sollten Sie eine Eigenheit dieser Funktion kennen, denn er markiert Intervalle, die sich lediglich berühren (das Ende des ersten Intervalls entspricht dem Beginn des zweiten Intervalls), als *nicht* überlappend. Das hat zur Folge, dass ein between-Operator andere logische Ergebnisse liefert als der overlaps-Operator.

26.1.2 Speicherung von Datumsintervallen mit WMSYS.WM_PERIOD

Eine interessante Alternative zur Speicherung von Intervallen findet sich im Werkzeugkasten des *Workspace-Managers*, einer Option, die in einer Oracle-Datenbank (nicht XE) normalerweise installiert ist. Falls nicht, fragen Sie Ihren Datenbankadministrator oder Apotheker. Es führt zu weit, den Workspace-Manager vorzustellen, nur so viel: In dieser Erweiterung existiert ein objektorientierter Datentyp WMSYS.WM_PERIOD. Dieser Datentyp speichert Datenintervalle. WMSYS ist ein Benutzer, dem dieser Datentyp gehört. Er definiert eine Reihe von Operatoren, die mit diesem Intervalltyp arbeiten können. All diese Operatoren und der Datentyp selbst sind über ein Synonym direkt in allen Schemata zugänglich. Diese Operatoren sind:

- WM_OVERLAPS
  Prüft, ob sich Zeitintervalle überlappen. Entspricht logisch dem overlaps-Operator, ist allerdings dokumentiert.
- WM_CONTAINS
  Prüft, ob das erste Intervall das zweite vollständig enthält.
- WM_MEETS
  Prüft, ob das Ende des ersten Intervalls dem Beginn des zweiten Intervalls entspricht.
- WM_EQUALS
  Prüft, ob zwei Intervalle identische Start- und Endzeiten haben.
- WM_LESSTHAN
  Prüft, ob Intervall 1 vor Intervall 2 endet.
- WM_GREATERTHAN
  Prüft, ob Intervall 1 nach Intervall 2 endet.
- WM_INTERSECTION
  Liefert das Intervall, das beiden übergebenen Intervallen gemeinsam ist.
- WM_LDIFF
  Liefert das Intervall, das vor dem gemeinsamen Intervall beider übergebener Intervalle liegt.

- WM_RDIFF
  Liefert das Intervall, das hinter dem gemeinsamen Intervall beider übergebener Intervalle liegt.

Als Beispiel können Sie sich vorstellen, dass eine Tabelle eine Spalte dieses Datentyps anlegt:

```
SQL> create table emp_dept(
  2    empno number,
  3    deptno number,
  4    date_range wm_period);
Tabelle wurde erstellt.

SQL> insert into emp_dept
  2  select empno, deptno,
  3         wm_period(hiredate, date '2999-12-31')
  4    from emp;
14 Zeilen wurden erstellt.

SQL> select e.empno, e.deptno,
  2         e.date_range.validfrom,
  3         e.date_range.validtill
  4    from emp_dept e;

EMPNO   DEPTNO DATE_RANGE.VALIDFROM  DATE_RANGE.VALIDTILL
------ ------- --------------------- ---------------------
  7369      20 17.12.1980            31.12.2999
  7499      30 20.02.1981            31.12.2999
  7521      30 22.02.1981            31.12.2999
  7566      20 02.04.1981            31.12.2999
  7654      30 28.09.1981            31.12.2999
  7698      30 01.05.1981            31.12.2999
  7782      10 09.06.1981            31.12.2999
  7788      20 19.04.1987            31.12.2999
  7839      10 17.11.1981            31.12.2999
  7844      30 08.09.1981            31.12.2999
  7876      20 23.05.1987            31.12.2999
  7900      30 03.12.1981            31.12.2999
  7902      20 03.12.1981            31.12.2999
  7934      10 23.01.1982            31.12.2999
14 Zeilen ausgewählt.
```

Listing 26.11 Verwendung des Datentyps WM_PERIOD

Sie erkennen, dass der Datentyp zwei Attribute VALIDFROM und VALIDTIL enthält, die durch den Konstruktor, den wir für diesen objektorientierten Typ erstellt haben, mit Werten belegt wurden. Diesen Datentyp können wir nun mit den Operatoren untersuchen, wie in der folgenden Abfrage, die analysiert, wer im Mai 1981 im Unternehmen gearbeitet hat:

```
SQL> select e.empno, e.deptno, e.date_range.validfrom
  2    from emp_dept e
  3   where wm_overlaps(
  4           date_range,
  5           wm_period(
  6             date '1981-05-01',
  7             date '1981-05-31')) = 1;

     EMPNO     DEPTNO DATE_RANGE.VALIDFROM
---------- ---------- ---------------------------------
      7369         20 17.12.80 00:00:00,000000 +02:00
      7499         30 20.02.81 00:00:00,000000 +02:00
      7521         30 22.02.81 00:00:00,000000 +02:00
      7566         20 02.04.81 00:00:00,000000 +02:00
      7698         30 01.05.81 00:00:00,000000 +02:00
```

Listing 26.12 Verwendung des Operators WM_OVERLAPS

Wir übergeben der Funktion zwei Zeitbereiche vom Typ WM_PERIOD. Der erste ist ja bereits in der Tabelle EMP_DEPT enthalten, den zweiten mit dem Testzeitraum erzeugen wir uns gleich selbst. An der Ausgabe erkennen Sie, dass wir eigentlich mit Zeitstempeln hätten arbeiten sollen, doch interessiert die Uhrzeit nicht, so dass wir uns hier aus Bequemlichkeit auf die implizite Konvertierung des leicht zu erzeugenden Datums in einen timestamp verlassen können. Auch die erste Ausgabe hat Zeitstempel ausgegeben, die ich jedoch für dieses Buch um die ohnehin leeren Zeitangaben erleichtert habe.

Eines stört allerdings am Typ WM_PERIOD: Hier wäre eine zweite Konstruktormethode gut gewesen, der man einen Zeitstempel und ein Intervall für die Länge des Zeitbereichs übergeben könnte. Es lohnt auch ein Blick auf die anderen Operatoren und Funktionen, die in diesem Schema definiert sind. Möglicherweise hilft Ihnen dieser Datentyp ja bei der Lösung eines Zeitbereichsproblems.

26.1.3 Andere Datenmodelle zur Speicherung von Datumsbereichen

Alternativ habe ich weitere Datenmodelle gesehen, um Zeitbereiche zu speichern. Ein Ansatz speichert aufeinanderfolgende Zeiträume lediglich mit einem Startdatum, ein weiterer Ansatz speichert ein Startdatum und eine Dauer.

Speicherung mittels eines Startdatums

Die Idee: Wenn für eine ID mehrere Zeilen mit jeweils definierten Startzeitzeitpunkten existieren, ist die Reihenfolge der Ereignisse über den Startzeitpunkt definiert, die Dauer über die Differenz des Startzeitpunktes des älteren zum Startzeitpunkt des jüngeren Eintrags.

Zunächst einmal scheint eine solche Speicherung eher unorthodox und wenig effizient zu sein, denn der offensichtliche Vorteil, nur ein Startdatum speichern zu müssen, scheint durch den Nachteil, stets die gesamte Tabelle nach möglichen weiteren Einträgen durchsuchen zu müssen, aufgezehrt zu werden. Allerdings ist die Suche nach dem jeweiligen Nachfolger, um die Dauer des Intervalls zu berechnen, nicht so schrecklich aufwendig, wenn Sie überlegen, dass ein Primärschlüssel dieser Spalte mindestens die Spalten ID und VALID_FROM enthalten muss. Durch diese Anforderung stehen die entsprechenden Zeilen der Intervalle im Index direkt beieinander, und das auch noch in der richtigen Reihenfolge. Ein weiterer Vorteil dieses Datenmodells besteht darin, dass es keine überlappenden Intervalle geben kann, denn ein neuer Eintrag beendet automatisch den älteren Eintrag, und der Primärschlüssel verhindert, dass ein Startdatum zweifach für eine ID gespeichert wird. Diese Gründe sind meiner Meinung nach gut genug, um uns die gleiche Projektsituation einmal in diesem Datenmodell anzusehen.

Ein Nachteil dieses Ansatzes besteht allerdings darin, dass ein Enddatum nicht ohne Weiteres eingegeben werden kann, möchten Sie ein Intervall beenden, geht dies nur durch eine weiteres Datum. Dieses Datum kann als zweite Spalte eingefügt werden oder aber als weitere Zeile, die durch ein Flag als Endereignis definiert wird. Für den Umbau unseres Datenmodells werden wir diesen Nachteil dadurch umgehen, dass wir einen weiteren Status definieren: End of project. Das Projektende wird dann durch eine eigene Zeile definiert. Problematischer ist es allerdings, dass die Zeiträume keine Lücken enthalten dürfen. Das ist zwar ebenfalls über einen weiteren Status (Project paused) zu organisieren, doch ist die Darstellung der Zeiträume dann doch recht aufwendig. Die Tabelle zur Speicherung solcher Daten sähe dann etwa so aus:

```
SQL> create table time_point_test (
  2    id number,
  3    action varchar2(25 char),
  4    valid_from date,
  5    constraint pk_time_point_test primary key (id, valid_from)
  6  );
SQL> Tabelle erstellt.
```

Listing 26.13 Erstellung der Tabelle im neuen Datenmodell

Wieder habe ich im Skript zum Buch eine SQL-Anweisung beigefügt, die unsere Beispieldaten einfügt. Wir haben nun folgende Daten, die ich im Fall des ersten und dritten Projekts um ein Enddatum ergänzt habe:


```
SQL> select *
  2    from time_point_test
  3   order by id, valid_from;

        ID ACTION                    VALID_FROM
---------- ------------------------- ----------------
         1 Start of project          15.02.2018 19:30
         1 Kick off                  16.02.2018 09:31
         1 Phase 1                   16.02.2018 12:00
         1 Phase 2                   03.03.2018 08:00
         1 End of project            19.04.2018 08:00
         2 Start of project          19.02.2018 19:30
         2 Kick off                  20.02.2018 09:31
         2 Phase 1                   20.02.2018 12:00
         2 Phase 2                   01.03.2018 08:00
         3 Start of project          18.02.2018 19:30
         3 Kick off                  19.02.2018 09:31
         3 Phase 1                   19.02.2018 12:00
         3 Phase 2                   18.03.2018 08:00
         3 End of project            07.04.2018 08:00

14 Zeilen ausgewählt.
```

Listing 26.14 Beispieldaten im neuen Datenmodell

Interessant wäre als Erstes vielleicht, eine Abfrage zu erstellen, die uns zeigt, von wann bis wann eine Projektphase galt. Um auf eine Zeile vor oder nach der aktuellen Zeile zu sehen, erinnern wir uns der analytischen Funktion lag bzw. lead, daher lautet die Abfrage:

```
SQL> select id, action, valid_from,
  2         coalesce(
  3           lead(valid_from) over
  4             (partition by id order by valid_from) -
  5                interval '1' second,
  6            date '2999-12-31') valid_to
  7    from time_point_test;

        ID ACTION                VALID_FROM      VALID_TO
---------- --------------------- --------------- ---------------
         1 Start of project      15.02. 19:30:00 16.02. 09:30:59
         1 Kick off              16.02. 09:31:00 16.02. 11:59:59
         1 Phase 1               16.02. 12:00:00 03.03. 07:59:59
         1 Phase 2               03.03. 08:00:00 19.04. 07:59:59
         1 End of project        19.04. 08:00:00 31.12. 00:00:00
```

```
        2 Start of project       19.02. 19:30:00 20.02. 09:30:59
        2 Kick off               20.02. 09:31:00 20.02. 11:59:59
        2 Phase 1                20.02. 12:00:00 01.03. 07:59:59
        2 Phase 2                01.03. 08:00:00 31.12. 00:00:00
        3 Start of project       18.02. 19:30:00 19.02. 09:30:59
        3 Kick off               19.02. 09:31:00 19.02. 11:59:59
        3 Phase 1                19.02. 12:00:00 18.03. 07:59:59
        3 Phase 2                18.03. 08:00:00 07.04. 07:59:59
        3 End of project         07.04. 08:00:00 31.12. 00:00:00
14 Zeilen ausgewählt.
```

Listing 26.15 Abfrage des Start- und Enddatums der Phasen

Sie sehen, dass mit einer – nun ja, einfachen? – SQL-Abfrage die gleichen Daten erzeugt werden konnten, die auch vorhin gespeichert wurden. Allerdings werden die Enddaten der Phasen nun dynamisch gerechnet und nicht mehr gespeichert, was schon angenehm ist, denn so müssen wir nicht mehr beim Aktualisieren einer Zeile auf diese Abgrenzung achten. Ein Nachteil dieses Ansatzes ist sicherlich, dass wir nun nicht mehr einfach unterscheiden können, ob ein Projekt beendet wurde (`status = 'End of project'`), denn auch hier wird als Enddatum ein Datum in der Zukunft angegeben. Dies könnten wir noch durch eine `case`-Anweisung entschärfen, allerdings sehen Sie, dass der Aufwand nun doch steigt.

Wenn Sie dieses Beispiel nachvollziehen, werden Sie zudem feststellen, dass der Ausführungsplan noch relativ einfach aussieht, insbesondere auch dann, wenn Sie lediglich die Daten eines einzigen Projekts auswerten und lediglich `ID` und `VALID_FROM` anzeigen lassen. In diesem Fall kann Oracle diese Abfrage direkt aus dem Index heraus beantworten. Grundsätzlich wäre es also möglich, diese Daten als View zur Verfügung zu stellen und die Auswertung gegen diese View durchzuführen.

Mich interessiert allerdings, ob es auch möglich ist, direkt mit den Daten zu arbeiten und eine Abgrenzung nach Monat durchzuführen. Wenn wir über das Problem nachdenken, wird zunächst klar, dass es schon nicht ganz einfach sein wird, zu erkennen, ob ein Intervall zu einem Monat gehört oder nicht. Natürlich, das Startdatum kann leicht daraufhin überprüft werden, ob es kleiner ist als das Monatsende oder nicht. Aber umgekehrt muss eben der Beginn der nächsten Zeile bekannt sein, um zu entscheiden, ob der aktuelle Monat überhaupt berührt wird oder nicht. Das ist allerdings nur mit der analytischen Funktion zu beantworten. Es ergibt sich also relativ schnell, dass wir den Umweg über die analytische Funktion gehen müssen. Ob dies ein Vor- oder Nachteil dieses Datenmodells ist, möchte ich im Moment noch nicht diskutieren. Wir könnten uns aber zum Prinzip machen, Datumsbereiche, die auf die hier geschilderte Weise gespeichert werden, durch eine View in das in Abschnitt 26.1.1, »Speicherung von Datumsbereichen mit zwei Zeitpunkten«, geschilderte Datenmodell zu überführen und die Auswertungen gegen diese View zu formulieren.

Andere Probleme, die wir für die erste Speicherform mit einer `VALID_FROM`- und einer `VALID_TO`-Spalte bereits diskutiert haben, existieren in dieser Form im Übrigen auch, denn wenn Sie zum Beispiel ein Intervall in ein bestehendes Intervall einschachteln möchten, kämen Sie auch hier nicht um eine neue, zusätzliche Zeile herum, die das Ende des eingeschachtelten Datums definiert. Allerdings stellen sich die logischen Probleme in diesem Zusammenhang als einfacher heraus als beim Modell mit zwei Spalten.

Ein echter Vorteil dieses Modells der Speicherung besteht aber darin, dass zur Abgrenzung eines Datensatzes der bestehende, derzeit aktuelle Datensatz nicht angefasst werden muss. Speichern Sie das Anfangs- und Enddatum in einer Zeile, müssen Sie stets eine Aktualisierung der momentan aktuellen Zeile durchführen, um den bestehenden Datensatz zu beenden, sowie eine `insert`-Anweisung für die nun gültige Zeile ausführen. Das Problem erscheint zunächst einmal nicht sehr groß, doch wenn extrem viele Zeilen vorliegen, ist die zusätzlich erforderliche `update`-Anweisung teuer. Diese Kosten haben Sie bei der Speicherung nur eines Datums nicht, allerdings wird die Ermittlung des aktuell gültigen Datensatzes hier auf den lesenden Zugriff verschoben. Dies kann jedoch, je nach konkreter Situation, für die Gesamtperformanz günstiger sein.

Speicherung mittels Startdatum in Dauer

Bei dieser Speicherung mittels Startdatum gehen wir ganz ähnlich vor wie bei der Speicherung des vorigen Abschnitts. Wir erweitern allerdings die Speicherung um eine Intervallspalte, die die Dauer des Intervalls enthält. Durch diese zusätzliche Information ist es nun möglich, Lücken zwischen den Intervallen zuzulassen, die immer dann entstehen, wenn eine Dauer kürzer ist als die Differenz zum Folgestartdatum. Dies war ja in dem Datenmodell mit nur einem Zeitstempel nicht möglich. Dieser Ansatz erscheint mir allerdings eher die Probleme beider Ansätze auf sich zu vereinigen, ohne signifikante Vorteile zu bieten, denn nun muss bei direkt aufeinanderfolgenden Intervallen doch eine Dauer berechnet und in einer anderen Spalte gepflegt werden, zudem ist die Nutzung eines Index hier kaum möglich, denn letztlich werden wir aus dem Startdatum und dem Intervall doch wieder ein Enddatum berechnen müssen. Dieses Enddatum könnten wir zwar indizieren, doch ist der Aufwand höher, als wenn wir direkt das Enddatum speicherten. Auch für die Diskussion um das Problem überlappender Bereiche helfen uns Intervalle nicht recht weiter, denn auch hier existiert keine einfache Möglichkeit, überlappende Intervalle zu verhindern. Zudem ist ein Intervall, wie Sie wissen, auch keine kleine Speicherstruktur, auch von daher ist also kein Vorteil zu erwarten.

Wenn Sie sich Abfragen gegen solche Datenstrukturen vorstellen, dann werden Sie wahrscheinlich ebenfalls zu einer View kommen, die das erste Datenmodell unserer kleinen Betrachtung simuliert. Dennoch möchte ich nicht automatisch eine Empfeh-

lung für das erste Modell aussprechen. Es ist zwar das häufigste Datenmodell, aber die Nachteile bei der Abgrenzung von Intervallen wiegt schwer, insbesondere dann, wenn Intervalle im Nachhinein noch ineinander geschachtelt werden können. In diesem Fall könnte die Speicherung durch ein Startdatum und die anschließende View auf diese Tabelle ein eleganterer Weg sein, dieses Problem zu lösen, zumindest, wenn keine Lücken zwischen den Intervallen möglich sein müssen.

26.1.4 Analyse gegen eine Zeitdimension

In Abschnitt 26.1.1, »Speicherung von Datumsbereichen mit zwei Zeitpunkten«, haben Sie gesehen, dass zeitbezogene Auswertungen sehr schnell auch zeitliche Abgrenzungen gegen ein festes Intervall, wie etwa einen Tag, eine Woche, einen Monat oder Ähnliches, erfordern können. Wir haben in diesem Beispiel die Auswertung gegen eine maßgeschneiderte View vorgenommen, mit deren Hilfe wir eine Referenztabelle anbieten konnten, um die Abgrenzung der Zeitintervalle gegen die Monatsgrenzen vornehmen zu können. Ein allgemeineres, aber nicht mehr so häufig außerhalb von Datenwarenhäusern anzutreffendes Verfahren, mit solchen Zeitabgrenzungen umzugehen, besteht darin, sich – sozusagen als Anleihe bei Datenwarenhäusern – eine Tabelle zu erstellen, die verschiedene Zeitintervalle bereits fertig enthält. In einer solchen Tabelle könnten wir zum Beispiel für jeden Tag eines Jahres und über einen Zeitraum von zehn Jahren eine Zeile vorhalten, die nicht nur das Datum des entsprechenden Tages, sondern darüber hinaus auch weitere Angaben zu diesem Tag enthält, zum Beispiel der wievielte Tag des Monats, des Quartals, des Jahres dieser Tag ist, wie viele Tage der Monat enthält usw. Eine solche Hilfstabelle dient als Quelle für die Zeitintervalle, mit deren Hilfe dann die Auswertung gegen Datumsbereiche vorgenommen werden kann. Die Grundlagen der Verwendung solcher Tabellen in Datenwarenhäusern besprechen wir in Kapitel 23, »Datenwarenhaus«.

Im Datenmodell des Beispielbenutzers `SH` findet sich ein Beispiel für eine solche Referenztabelle. Sie trägt den Namen `TIMES` und ist dort im Rahmen eines Datenmodells für Datenwarenhäuser als sogenannte Dimensionstabelle eingerichtet. Ich möchte an dieser Stelle nicht allzu tief in die Denkweise von Datenwarenhäusern eindringen, sondern lediglich grob erklären, dass der Begriff da herrührt, dass entlang vorgegebener Dimensionen ein Bericht gefiltert werden kann. So könnte eine Dimension die Zeit beschreiben, eine andere die Kunden, eine dritte die Produkte. Möchten Sie nun einen Bericht nach Produkt und Monat filtern, können Sie dies tun, indem Sie die Tabellen für die Produkte und die Zeit filtern. Da jeder Eintrag dieser Dimensionstabelle über eine Fremdschlüsselbeziehung zu einem Messwert steht, zum Beispiel einem Verkauf eines Produkts an einen Kunden an einem Tag, wird über diese Filterung auch die Verkaufstabelle gefiltert.

Auf unser Beispiel übertragen wäre eine solche Tabelle ebenfalls hilfreich, denn die Filterung eines Berichts nach der Zeit ist eine absolut übliche Anforderung, daher

profitieren wir von dieser Idee auch außerhalb des Rahmens eines Datenwarenhauses. Sehen wir uns doch einmal einige der Spalten der Tabelle `TIMES` an (Tabelle 26.1).

Spaltenname	Bemerkung
TIME_ID	Primärschlüssel, Datum des Tages, 00:00 Uhr feinste Granularität
DAY_NAME	Montag bis Sonntag
DAY_NUMBER_IN_WEEK	1–7, 1 = Montag
DAY_NUMBER_IN_MONTH	1–31
CALENDAR_WEEK_NUMBER	1–53
WEEK_ENDING_DAY	Datum des letzten Tages der Woche
CALENDAR_MONTH_NUMBER	1–12
DAYS_IN_CAL_MONTH	Anzahl der Tage des Monats
END_OF_CAL_MONTH	Datum des letzten Tages des Monats
CALENDAR_MONTH_NAME	Januar bis Dezember
DAYS_IN_CAL_QUARTER	Anzahl der Tage des Quartals
END_OF_CAL_QUARTER	Datum des letzten Tages des Quartals
CALENDAR_QUARTER_NUMBER	1–4
CALENDAR_YEAR	Zahl (zum Beispiel 2012)
DAYS_IN_CAL_YEAR	Anzahl der Tage eines Jahres
END_OF_CAL_YEAR	Datum des letzten Tages des Jahres

Tabelle 26.1 Einige Spalten der Dimensionstabelle TIMES

Die Auswahl, die ich getroffen habe, schließt die Spalten des fiskalischen Jahres aus, die bei abweichendem Geschäftsjahr durchaus sinnvoll sein könnten, sowie einige ID-Spalten, die mir für eine solche Tabelle etwas zu weit führen. Ebenfalls habe ich die Beschreibungsspalten entfernt. Es bleibt aber das Prinzip erkennbar: Wir haben eine Tabelle, die redundant für ein Datum speichert, zu welchem Monat es gehört, wie viele Tage der Monat enthält, welches Jahr existiert usw. An eine schöne Erweiterung hierzu erinnere ich mich aus einem Projekt im Einzelhandel: Dort wurde in einer Spalte für jedes Jahr gepflegt, wie viele Tage vor oder nach Ostern der entsprechende Tag ist. Der Hintergrund: Da die beweglichen Feiertage in Deutschland auf Ostern bezogen sind, können mit dem Abstand zu Ostern Brückentage etc. leicht identifi-

ziert und jahresübergreifend ausgewertet werden. Im Skript zum Buch habe ich eine Anweisung hinterlegt, die als Startpunkt für eigene Projekte eine solche Zeittabelle als materialisierte Sicht erzeugt. Ein Problem ist die korrekte Berechnung der Kalenderwoche, ich habe im Skript die ISO-Definition verwendet.

Eine solche Tabelle ist durch eine entsprechende SQL-Abfrage relativ leicht automatisiert mit Daten zu füllen (wenn wir einmal von der Differenzspalte zu Ostern absehen, die die Programmierung einer Datumsfunktion nach Gauß erfordert). Darüber hinaus ist diese Tabelle ein Kandidat für eine materialisierte Sicht, die sich zu Beginn eines neuen Jahres neu berechnet und so immer einen Zeitraum von 10, 15 oder 30 Jahren relativ zum aktuellen Jahr abdeckt.

Lassen Sie mich, einfach als Wiederholung und weil es so praktisch ist, noch einmal zeigen, wie Sie es schaffen, mit einer dynamischen Anweisung ein relatives Datum zu bestimmen. Erinnern Sie sich bitte an die Datumsarithmetik, die von `trunc(sysdate)` ausgeht und von dort aus das Startdatum berechnet:

```
SQL> select trunc(sysdate, 'Y') - interval '5' year start_datum
  2    from dual;

START_DATUM
-----------
01.01.2010
```

Listing 26.16 Berechnung eines relativen Startdatums

Von hier aus können Sie die Pseudospalte `ROWNUM`, `LEVEL` oder was auch immer aufaddieren, bis Sie beim gewünschten Enddatum angekommen sind. Entweder rechnen Sie das aus, oder Sie lassen SQL entscheiden, wann es genug ist, wie im folgenden Beispiel, das ein Zeitintervall von +/– fünf Jahren zum aktuellen Jahr ermittelt:

```
SQL>   with datum as(
  2         select trunc(sysdate, 'Y') - interval '5' year start_datum
  3           from dual)
  4  select start_datum + numtodsinterval(rownum - 1, 'day') tag_beginn,
  5         start_datum + numtodsinterval(rownum, 'day')
  6                     - interval '1' second tag_ende
  7    from all_objects
  8   cross join datum
  9   where start_datum + numtodsinterval(rownum, 'day')
 10          <= trunc(sysdate, 'Y') + interval '5' year;

TAG_BEGINN TAG_ENDE
---------- -------------------
01.01.2010 01.01.2018 23:59:59
02.01.2010 02.01.2018 23:59:59
03.01.2010 03.01.2018 23:59:59
...
31.12.2019 31.01.2019 23:59:59

3652 rows selected.
```

Listing 26.17 Erzeugung einer relativ begrenzten Zeitdimension

Selbst wenn Sie zu Beginn etwas über solche Abfragen nachdenken müssen: Es lohnt sich, denn dadurch bleiben Ihre Abfragen stabiler über die Zeit. Der Vorteil liegt auf der Hand: Alle Zeitauswertungen können nun gegen diese »Zeitdimensionstabelle« ausgeführt werden und werden dadurch einheitlicher. Wenn Sie beispielsweise eine Auswertung auf einen Monat gruppieren möchten, können Sie dies erreichen, indem Sie das Umsatzdatum auf den Tag abrunden und dann gegen die Tabelle `TIMES` joinen. Über die Spalte `CALENDAR_MONTH_NUMBER` steht Ihnen nun ein Gruppierungs- und Filterkriterium zur Verfügung. Natürlich funktioniert dieser Trick so auch gegen Quartalsauswertungen etc. Die Tabelle `TIMES` filtert und gruppiert nun den Bericht.

Wie bereits gesagt, ist dieser Tipp etwas für Datenbanken, die gelegentlich auch Berichte erzeugen müssen, ohne den ganzen Weg hin zu einem spezialisierten Datenwarenhaus gehen zu müssen. Als Anleihe ist diese Idee aber sicher eine gute Wahl. Ein Join auf diese Zeittabelle ist natürlich nicht immer über einen Primärschlüssel-Fremdschlüssel-Join möglich, zum Beispiel dann nicht, wenn die Zeit in der Verkaufstabelle mit Uhrzeit gespeichert wurde. In diesen Fällen ist eine andere Join-Bedingung (zum Beispiel `trunc(umsatzdatum) = time_id`) erforderlich, wie es ja auch in unserer Abfrage aus dem letzten Abschnitt notwendig war, in der wir die Abgrenzung zum Monat durchgeführt haben.

Machen Sie so etwas hingegen häufiger und beginnen Sie, eigene Tabellen für Ihre Berichte zu pflegen, denken Sie bitte über eine zusätzliche Spalte in diesen Tabellen nach, die das um die Uhrzeit bereinigte Datum als Join-Bedingung für die Zeitdimension mit sich führt.

26.2 Historisierung und Logging

Gehen wir von dem Problem aus, dass Sie nicht wissen, warum einem Kunden vor drei Monaten eine Rechnung an eine falsche Adresse gesendet wurde, weil in der Zwischenzeit die Adresse korrigiert wurde. Ein pragmatischer Ansatz, dieses Problem zu lösen, besteht darin, der Rechnung einfach die Adresse, an die die Rechnung ging, beizufügen. Das habe ich durchaus schon häufiger in Datenbanken gesehen. Die Datenmodellierung ist dann so, dass die Spalten, die eine konkrete Adresse beschrei-

ben, in der Tabelle RECHNUNG noch einmal auftauchen und die Daten dorthin kopiert werden. Zunächst einmal sieht das sehr nach redundanter Speicherung von Daten aus, doch könnte man argumentieren, dass mit dieser Speicherung eben nicht die Adresse des Kunden, sondern die Adresse des Kunden *zum Zeitpunkt der Rechnungsstellung* gespeichert werde, was eine andere Information darstellt. Zudem, so könnte man argumentieren, sei diese Form der Speicherung auch deshalb nicht schlecht, weil für die Ermittlung der Rechnungsadresse kein Join auf eine andere Tabelle gebraucht werde. Schließlich könnte man sogar auf die Idee kommen, diese Speicherung deshalb zu bevorzugen, weil das Programm eventuell die Änderung der Adresse für eine Rechnung zuließe, ohne die tatsächliche Adresse in der Datenbank zu aktualisieren. In diesem Fall würde dann die Adresse so, wie sie eingegeben wurde, gespeichert. Natürlich bietet sich eine solche Strategie nicht für sehr viele Tabellen und Anwendungsfälle an, Sie würden ja Tabellenspalten im großen Stil in mehreren Tabellen anlegen und pflegen müssen.

Nun ja, in jedem Fall haben diese Argumente einen faden Nachgeschmack. Richtig sauber wirkt das nicht. Nicht nur die Rechnung benötigt ja eventuell das Wissen über die Adresse des Kunden, sondern auch andere Teile Ihrer Anwendung. Auch das letzte Argument, dass eine abweichende Lieferadresse nur bei der Rechnung gespeichert wird, überzeugt nicht, weil eine Adresse auch außerhalb einer konkreten Rechnungsstellung einen Wert an sich darstellt, der nicht dadurch eingeschlossen werden sollte, dass er nur im Zusammenhang einer speziellen Rechnung gefunden werden kann. Doch will ich nicht verschweigen, dass es für diese Art der Speicherung auch Befürworter gibt. Dort wird vor allem das Argument herangezogen, dass eine Rechnung nun einmal ein Dokument ist, das als Ganzes an einen Kunden gegangen ist und deshalb auch als Ganzes in der Datenbank gespeichert werden soll. Da nun die Adresse zur Rechnung gehört, ist diese eben auch mit der Rechnung zu speichern. Konsequent zu Ende gedacht wäre dieser Gedanke im Übrigen durch das Ablegen eines finalen Dokuments, etwa als PDF, in der Datenbank, um genau nachweisen zu können, welches Dokument den Kunden erreicht hat oder eben auch nicht. Lassen wir das einmal beiseite (die Anforderungen der Realität lassen alle möglichen Modellierungen sinnvoll erscheinen), möchte ich mich mit meinen weiteren Betrachtungen an die häufige Situation richten, dass sich nämlich Adressen über die Zeit ändern können und die alten Adressdaten einfach nicht verloren gehen sollen.

Für dieses Problem finden sich Datenmodelle, die sich damit auf grundlegende Weise beschäftigen. Unterscheiden möchte ich dabei zwei (oder drei) Strategien: das *Logging* von Stammdatenänderungen und die *historisierenden Datenmodelle*. Der Unterschied liegt in der Verwendung: Während das Logging von Stammdatenänderungen eher den Fokus hat, später nachzuvollziehen, wie sich ein Datenbestand geändert hat, stellen wir an historisierende Datenmodelle die Forderung, dass diese auf einen alten Zustand zurückgestellt werden können, also ohne großen Aufwand

die Daten so zeigen können sollen, wie sie zu einem Stichtag ausgesehen haben. Als Sonderfall eines historisierenden Datenmodells stelle ich Ihnen dann noch das *bitemporale Datenmodell* vor.

26.2.1 Logging von Stammdatenänderungen

Wenn ein Datenmodell die Adressdaten eines Rechnungsempfängers mit der Rechnung speichert, hat das etwas von einem Logging, allerdings mit dem Nachteil, dass dieses Logging nicht generell alle Änderungen der Adressen dokumentiert, sondern nur die Adresse, die zum Zeitpunkt einer Rechnungsstellung gerade in der Datenbank aktuell war. Ändert sich die Adresse nicht, werden aber viele Rechnungen gestellt, ist dieser Ansatz zudem sehr redundant. Das Logging von Stammdatenänderungen soll diesem Problem begegnen.

Dieses Logging wird normalerweise dadurch erreicht, bei Änderungen an einer Tabelle den alten Zustand der geänderten Tabellenzeile in eine speziell dafür eingerichtete Logging-Tabelle zu kopieren. Es bietet sich an, für die »Historisierung« dieser Daten einen Trigger einzurichten, der im Fall einer DML-Anweisung den alten Zustand in diese Tabelle kopiert. Zwar habe ich meine Bedenken gegen den Einsatz von Zeilen-Triggern geäußert, doch ist dies hier möglicherweise kein Problem, weil das Logging von Änderungen normalerweise nur auf Stammdatentabellen ausgeführt wird, nicht auf Bewegungstabellen. Halten Sie mögliche Auswirkungen auf die Performanz jedoch im Blick. Ich benutze den Begriff »Historisierung« hier in Anführungsstrichen, weil eigentlich keine vollständige Historisierung, sondern lediglich ein Logging der Datenänderung durchgeführt wird. Ich verstehe unter einem Datenmodell, das Stammdatenveränderungen aufzeichnet, um sie anschließend für die gezielte Recherche »wieder sichtbar« zu machen, kein historisierendes Datenmodell, denn an ein solches Datenmodell möchte ich weiter gehende Anforderungen stellen. Im Grunde ist das Logging von Datenänderungen eine Vereinfachung des historisierenden Datenmodells, das wir uns im nächsten Abschnitt etwas genauer ansehen: Es schreibt, ähnlich einem Bondrucker einer Registrierkasse, einfach weg, was gewesen ist, bevor Daten durch neue ersetzt werden. Dadurch bleiben die Daten verfügbar, in der aktuellen Tabelle sind sie aber nicht mehr sichtbar.

Ein solches Datenmodell finden Sie beim Benutzer HR, genauer in der Tabelle JOB_HISTORY. Diese Tabelle speichert den alten Beruf eines Mitarbeiters, der einen neuen Beruf erhält, sowie den Zeitraum, zu dem diese Kombination von Mitarbeiter und Beruf gültig war. Beachten Sie, dass diese Tabelle das Verfahren nutzt, die Gültigkeit über zwei Zeitpunkte zu steuern, daher besteht die Problematik der Abgrenzung der Zeitintervalle. Dieses Datenmodell hat hier aber einen Vorteil, denn durch die Trennung in zwei Tabellen enthält die Tabelle mit den aktuellen Daten immer nur das Datum, ab dem ein Beruf gegolten hat, während die Logging-Tabelle zeitlich abge-

grenzt wird. Da in dieser Tabelle aber nur Intervalle der Vergangenheit stehen, ist eine parallele Änderung von bestehenden Zeilen in der Logging-Tabelle nicht nötig. Bitte vollziehen Sie diesen Gedanken an den Tabellen von `HR` nach, damit Sie sich in die Problematik von Zeitverwaltung besser eindenken können. Ähnlich verfährt diese Tabelle bei Änderungen der Abteilung. Und diese Tabelle hat bereits eine Menge der Probleme, die wir mit dem Speichern von Datenänderungen über die Zeit haben. Zum einen ist es nämlich gar nicht so einfach, herauszufinden, welchen Beruf welche Mitarbeiter am 15. Juni 2005 hatten. Vielleicht versuchen Sie sich einmal an dieser Abfrage. Das Problem entsteht, weil die Tabelle `JOB_HISTORY` eben nur die Daten enthält, die geändert wurden, nicht aber den aktuellen Stand. Ist nun ein Mitarbeiter immer noch in seinem ersten Beruf, so müssen für diesen Mitarbeiter die Daten aus einer anderen Tabelle kommen als für einen Mitarbeiter, der bereits einige Berufsänderungen hinter sich hat und daher einen Datensatz in der Tabelle `JOB_HISTORY` hat.

Man könnte, das muss ich zugeben, dieses Problem recht einfach dadurch lösen, dass in der Tabelle `JOB_HISTORY` alle Zeilen aus `EMPLOYEE` initial hineinkopiert würden. Dann wäre die gesamte Historie in `JOB_HISTORY` enthalten, allerdings zulasten der Tatsache, dass Sie nun alle Daten doppelt speichern: den aktuellen Zustand in der Tabelle `EMPLOYEE`, die gesamte Historie in der Tabelle `JOB_HISTORY`. Das Problem besteht eigentlich darin, dass die Zuordnung eines Mitarbeiters zu einem Job und zu einer Abteilung eigentlich keine `1:n`-Beziehung ist, sondern eine `m:n`-Beziehung über die Zeit, so dass Sie eine eigene Tabelle für die Zuordnung benötigen. Wie das geht, erläutere ich im nächsten Abschnitt.

Ein weiteres Problem: Diese Tabelle speichert die Beziehung des Mitarbeiters zu einem Beruf *und* zu einer Abteilung. Ändert sich nun aber der Beruf, nicht aber die Abteilung, wird die neue Abteilung (die die alte ist) noch einmal in der `JOB_HISTORY` gespeichert. Umgekehrt verhält es sich, wenn der Mitarbeiter mit gleichem Beruf die Abteilung wechselt. Die Attribute `JOB_ID` und `DEPARTMENT_ID` des Mitarbeiters ändern sich nicht notwendigerweise synchron, daher entsteht eine nicht eindeutige Speicherung in der Tabelle `JOB_HISTORY`, die sehr unschön ist. Datenmodelle, die diese Art des Loggings implementieren, leiden zudem an einer großen Zahl spezialisierter »Historientabellen«, weil im Normalfall für jede Tabelle, die auf diese Weise Datenänderungen aufzeigen soll, jeweils eine Historientabelle erforderlich ist, die zudem das Problem haben, nur sehr schwierig Änderungen an der Struktur der »Basistabellen« mitzumachen, da in ihnen ja historische Daten mit eventuell anderer Struktur gespeichert wurden.

In einem Projekt, in dem von mir diese Art des Loggings gefordert wurde, habe ich versucht, das Problem dadurch zu lösen, dass ich eine generische Logging-Tabelle eingesetzt habe. Dieser Ansatz, das muss ich gleich sagen, eignet sich nicht für viele Millionen oder gar Milliarden Datensätze, ist aber aus meiner Sicht für das Logging

von Stammdatensätzen eine überlegenswerte Alternative. Wie funktioniert so etwas?

Die Basis besteht darin, einige der Informationen, die für jeden Log-Eintrag benötigt werden, in eigenen Spalten einer Tabelle anzulegen. Zu diesen Informationen gehören die üblichen Verdächtigen: *Wer* hat *was* an *welcher Tabelle wann* gemacht? Damit meine ich den angemeldeten Benutzer, eine Einfüge-, Änderungs- oder Löschoperation, den Namen der Tabelle und, wenn einheitlich Zahlen als Primärschlüssel genutzt werden, auch den Primärschlüssel sowie einen Zeitstempel. Das Problem sind aber die Daten, die geändert wurden. Hier habe ich mich entschlossen, einen objektorientierten Typ einzusetzen, der auf einfache Art eine Anzahl Tupel entgegennehmen kann in der Form `<Spaltenname>, <Wert als Zeichenkette>`. Ich zeige Ihnen hier die SQL-Anweisung zum Erzeugen einer solchen Tabelle inklusive der objektorientierten Typen:

```
SQL> create or replace
  2  type log_arg as object(
  3    col varchar2(32),
  4    val varchar2(4000 char));
  5  /
Typ wurde erstellt.

SQL> create or replace type log_args as table of log_arg;
  2  /
Typ wurde erstellt.

SQL> create sequence log_seq;
Sequence wurde erstellt.

SQL> create table log_historie (
  2     id number,
  3     wer varchar2(50 char) not null,
  4     wann date not null,
  5     tabelle varchar2(32 char) not null,
  6     aktivitaet char(1 char) not null,
  7     row_id number not null,
  8     daten log_args,
  9   constraint log_historie_pk primary key (id)
 10  )
 11  nested table daten store as log_daten
 12  return as value;
Tabelle wurde erstellt.
```

Listing 26.18 SQL-Anweisung zur Erzeugung einer generischen Log-Tabelle

Für Logging-Zwecke ist das eigentlich ausreichend, denn wir möchten im Zweifel ja lediglich eruieren können, was sich geändert hat. Nun können wir einen einfachen Trigger schreiben, der auf jede Tabelle angewendet werden kann, die Einträge in unsere zentrale Logging-Tabelle machen soll. Dieser Trigger unterscheidet die verschiedenen Aktivitäten, belegt die Daten der explizit definierten Spalten vor und erzeugt eine Instanz des Objekttyps, der die Tupel der Spalten aufnimmt. Die folgende Beispielanweisung zeigt, auf welche Weise Daten in die generische Log-Tabelle gespeichert werden können:

```
insert into log_historie (
  id, wer, wann, tabelle, aktivitaet, row_id, daten)
values(
  log_seq.nextval, 'WILLI', systimestamp, 'TEST_TABELLE', 12345,
  log_args(
    log_arg('SPALTE_1', 'Alter Wert'),
    log_arg('SPALTE_2', 'Alter Wert 2')));
```

Listing 26.19 Beispielhafte INSERT-Anweisung für einen Log-Eintrag

Ein Trigger auf Tabelle EMP könnte daher aussehen wie folgt:

```
SQL>  create or replace
  2  trigger ariud_emp
  3  after insert or update or delete on emp
  4  for each row
  5  declare
  6    aktion char(1);
  7  begin
  8    aktion := case
  9              when inserting then 'I'
 10              when updating then 'U'
 11              else 'D' end;
 12    insert into log_historie
 13      (id, wer, wann, tabelle, aktivitaet, row_id, daten)
 14    values (
 15      log_seq.nextval, user, systimestamp, 'EMP', aktion, :new.empno,
 16      log_args(
 17        log_arg('ENAME', :new.ename),
 18        log_arg('JOB', :new.job),
 19        log_arg('MGR', :new.mgr),
 20        log_arg('HIREDATE', to_char(:new.hiredate, 'dd.mm.yyyy')),
 21        log_arg('SAL', to_char(:new.sal)),
 22        log_arg('COMM', to_char(:new.comm)),
 23        log_arg('DEPTNO', to_char(:new.deptno))
 24      )
```



```
 25    );
 26  end;
31  /
Trigger wurde erstellt.
```

Listing 26.20 Ein einfacher Log-Trigger

Dieser Trigger speichert die Aktion in einer Variablen mit dem Namen AKTION und ruft anschließend die insert-Anweisung für die Log-Tabelle auf. Mit Hilfe der Pseudovariablen new können dann die neuen Werte in die Tabelle eingefügt werden. Da diese Trigger feuern, wann immer eine DML-Anweisung ausgeführt wird, können wir unsere Logging-Tabelle mit initialen Werten füllen, wenn uns danach ist, indem wir einfach eine update-Anweisung auf die Tabelle ausführen, ohne etwas zu ändern. Dies kopiert alle Daten in die Logging-Tabelle.

Eine View kann anschließend die Einträge in der Logging-Tabelle für die Oberfläche wieder »relational« sichtbar machen:

```
SQL> select l.wer, l.wann, l.tabelle, l.aktivitaet, l.row_id, d.*
  3    from log_historie l
  4   cross join table(l.daten) d

WER    WANN       TABEL A ROW_ID COL        VAL
------ ---------- ----- - ------ ---------- ---------------
SCOTT  03.08.12   EMP   U   7369 ENAME      SMITH
SCOTT  03.08.12   EMP   U   7369 JOB        CLERK
SCOTT  03.08.12   EMP   U   7369 MGR        7902
SCOTT  03.08.12   EMP   U   7369 HIREDATE   17.12.1980
...
SCOTT  03.08.12   EMP   U   7934 ENAME      MILLER
SCOTT  03.08.12   EMP   U   7934 JOB        CLERK
SCOTT  03.08.12   EMP   U   7934 MGR        7782
SCOTT  03.08.12   EMP   U   7934 HIREDATE   23.01.1982
SCOTT  03.08.12   EMP   U   7934 SAL        1300
SCOTT  03.08.12   EMP   U   7934 COMM
SCOTT  03.08.12   EMP   U   7934 DEPTNO     10
98 Zeilen ausgewählt.
```

Listing 26.21 Darstellung der Log-Tabelle in relationaler Form

Hübscher ist natürlich die pivotierte Darstellung, doch leider gelingt diese nicht mit dem pivot-Operator, weil diesem die Spalten COL und VAL zu »kompliziert« sind:

```
SQL> select l.wer, l.wann, l.tabelle, l.aktivitaet, l.row_id, d.*
  2    from log_historie l,
  3         table(l.daten) d
```

```
  4  pivot (d.val for d.col in
  5        ('ENAME', 'JOB', 'MGR', 'HIREDATE',
  6         'SAL', 'COMM', 'DEPTNO'));
 pivot (d.val for d.col in ('ENAME', 'JOB', 'MGR', 'HIREDATE', 'SAL', 'COMM',
'DEPTNO'))
                   *
FEHLER in Zeile 4:
ORA-01748: Hier sind nur einfache Spaltennamen zulässig
```

Listing 26.22 Zu kompliziert: Die Pivot-Klausel ist am Ende.

Sie können die Daten aber mit der herkömmlichen Methode pivotieren. Die select-Abfrage hierfür finden Sie online; ich hoffe, wie das geht, ist so weit klar. Allerdings benötigen Sie nun eine View pro Tabelle, die im Logging steht, denn Sie sprechen die Spalten ja explizit an. Mit etwas Programmieraufwand, und das ist die Lösung, die ich für meinen Kunden entwickelt hatte, ist es aber auch möglich, die Sicht auf diese Tabelle mit Hilfe von XML in eine HTML-Seite umzurechnen. Damit haben Sie beide Vorteile: generische Speicherung und flexible Sicht auf die Daten. Zudem ist die Logging-Tabelle durch entsprechende Indizierung auf den Tabellennamen und den Änderungsbenutzer auch für Ad-hoc-Abfragen gut gerüstet, die Informationen kommen recht schnell. Aufgrund der doch etwas komplexeren Programmierung erspare ich mir und Ihnen allerdings das Beispiel. Damit haben Sie die Keimzelle einer generischen Lösung für das Logging.

Wenn Sie im Übrigen die Speicherung der geänderten Daten als Objekt nicht mögen, bietet es sich an, die Tabelle als Detailtabelle mit den Spalten COL und VAL direkt anlegen zu lassen. Es ändert sich im Grunde nur, dass der Trigger den objektorientierten Typ an ein Stückchen Programmcode übergibt, der daraus die Detailtabelle füllt. Um ehrlich zu sein, ist das die Alternative, die ich implementiere, denn ich hatte meinen Unmut über die Speicherung von Objekten ja schon kundgetan. Zudem ist dieser Weg für die anschließende Auswertung auch einfacher. Wenn Sie mögen, können Sie zudem Logik hinterlegen, die lediglich die Spalten speichert, die sich tatsächlich verändert haben. Das geht, indem Sie entweder den alten Wert im Trigger mitgeben oder, was in der Benutzung einfacher ist, die übergebenen Werte mit den bereits in der Detailtabelle gespeicherten Werten vergleichen. Das geht durchaus auch in einer einzigen insert-Anweisung. Diese Ideen sollten Ihnen aber als Anregung ausreichen, die Implementierung einmal selbst zu versuchen. In jedem Fall ist ein gewisser Programmieranteil erforderlich, daher möchte ich dieses Thema hier nicht weiter vertiefen.

26.2.2 Historisierende Datenmodelle

Unter einem historisierenden Datenmodell möchte ich hier ein Datenmodell verstehen, das auf einfache Weise in der Lage ist, den Datenzustand zu jedem beliebigen

Zeitpunkt darzustellen. Erinnern Sie sich an das Datenmodell des Benutzers HR, dann ist die Tabelle JOB_HISTORY wohl offensichtlich so etwas wie eine Tabelle, die für eine historisierende Speicherung verwendet werden soll, denn sie speichert ja, welcher Mitarbeiter welchen Beruf zu welcher Zeit hatte. Ich habe allerdings bereits gesagt, dass ich dieses Datenmodell für kein gutes Beispiel für ein historisierendes Datenmodell halte, denn der Aufwand, um herauszufinden, welcher Mitarbeiter welchen Beruf am 15.05.2004 hatte, ist relativ hoch, wie die folgende Auswertung zeigt (aber die haben Sie ja bereits selbst entwickelt, nicht wahr?):

```
SQL> select e.employee_id, e.first_name, e.last_name,
  2         coalesce(jh.job_id, e.job_id) job_id
  3    from employees e
  4    left join job_history jh
  5      on e.employee_id = jh.employee_id
  6   where date '2004-05-15'
  7         between coalesce (jh.start_date, e.hire_date)
  8             and coalesce (jh.end_date, sysdate);

EMPLOYEE_ID FIRST_NAME           LAST_NAME                 JOB_ID
----------- -------------------- ------------------------- ----------
        102 Lex                  De Haan                   IT_PROG
        101 Neena                Kochhar                   AC_MGR
        201 Michael              Hartstein                 MK_REP
        200 Jennifer             Whalen                    AC_ACCOUNT
        184 Nandita              Sarchand                  SH_CLERK
        205 Shelley              Higgins                   AC_MGR
        137 Renske               Ladwig                    ST_CLERK
        115 Alexander            Khoo                      PU_CLERK
        174 Ellen                Abel                      SA_REP
        157 Patrick              Sully                     SA_REP
        156 Janette              King                      SA_REP
        108 Nancy                Greenberg                 FI_MGR
        204 Hermann              Baer                      PR_REP
        203 Susan                Mavris                    HR_REP
        141 Trenna               Rajs                      ST_CLERK
        206 William              Gietz                     AC_ACCOUNT
        109 Daniel               Faviet                    FI_ACCOUNT
        192 Sarah                Bell                      SH_CLERK
        100 Steven               King                      AD_PRES
19 Zeilen ausgewählt.
```

Listing 26.23 Darstellung der Datensituation am 15.05.2004

Interessant ist zum Beispiel `Neena Kochhar`, die aktuell `AC_VP` ist, zum Stichtag aber noch `AC_MGR` war. Sie sehen, dass wir zur Beantwortung dieser relativ einfachen Frage einen Outer Join sowie einige `null`-Wert-Behandlungsfunktionen benötigen. Das Problem dieses Datenmodells: Logisch existiert eine `m:n`-Beziehung zwischen dem Mitarbeiter und der Abteilung einerseits und dem Mitarbeiter und dem Beruf andererseits. Das können wir uns so klarmachen: Ein Mitarbeiter kann über die Zeit wechselnde Berufe haben, aber in der gleichen Abteilung arbeiten. Zudem kann ein Mitarbeiter auch in wechselnden Abteilungen arbeiten, aber den gleichen Beruf behalten. Natürlich kann ein Berufs- und Abteilungswechsel auch einmal zeitgleich vorkommen, doch gibt es keine fixe Abhängigkeit zwischen diesen beiden Attributen, ein Abteilungswechsel ist also nicht zwingend mit einem Berufswechsel verbunden und umgekehrt. Eventuell ist es sogar möglich, dass ein Mitarbeiter zwei Berufe zur gleichen Zeit hat oder in zwei Abteilungen arbeitet. Ähnliches gilt übrigens auch für die Zuordnung eines Mitarbeiters zu seinem Manager, was in diesem Datenmodell überhaupt nicht nachvollzogen wird.

Da also `m:n`-Beziehungen vorliegen (und zwar gleich zwei oder drei davon, wie Sie mögen), sollten diese nicht in einer Tabelle gespeichert werden, wie dies im Datenmodell aber der Fall ist. Die Lösung für das Problem, die Historie der beruflichen Entwicklung mit der Tabelle `JOB_HISTORY` nachzuzeichnen, ist nachträglich angebaut und zwingt uns in diese Form der Abfrage. Richtig komplex wird die Abfrage noch, wenn wir außerdem berücksichtigen, dass ein Eintrag in Tabelle `JOB_HISTORY` immer dann erforderlich wird, wenn sich Beruf *oder* Abteilung geändert haben.

Als kleine Zwischenaufgabe: Zählen Sie doch einfach einmal, wie viele Abteilungswechsel ein Mitarbeiter in seiner Karriere hatte. Ignorieren Sie dabei Berufswechsel, die keinen Abteilungswechsel zur Folge hatten, zählen Sie aber Wechsel, wenn ein Mitarbeiter in eine andere Abteilung und anschließend in die alte Abteilung zurückgekommen ist. Viel Vergnügen!

Dadurch kann es durchaus vorkommen, dass in der Tabelle bezüglich der Abteilung gar keine Änderung stattgefunden hat, obwohl eine Zeile in der Historie steht. Normalerweise würden diese Informationen in einer, besser noch zwei separaten Tabellen gespeichert, etwa so wie in Abbildung 26.2.

Wenn Sie genauer hinschauen, erkennen Sie, dass die Verweise auf `DEPARTMENTS` und `JOBS` aus der Tabelle `EMPLOYEES` verschwunden sind. Das ist korrekt so, diese Relation ist nun in die Zuordnungstabelle ausgelagert worden, denn sie müssen ja als `m:n`-Beziehung gestaltet werden. Nebenbei können wir dieses Spiel immer weiter treiben: Eigentlich sollten wir die Änderung auch um die Spalte `SALARY` erweitern, denn auch das Gehalt ändert sich über die Zeit. Wahrscheinlich ist das Gehalt nur lose an den Beruf gekoppelt, es kann sich ja bei gleichem Beruf über die Zeit ändern. Daher gilt analog, dass auch das Gehalt in eine eigene Tabelle ausgelagert werden sollte, wenn es wichtig ist, zu wissen, welches Gehalt wann verdient wurde. Die Zuordnung eines

Mitarbeiters zu seinem Vorgesetzten hatte ich bereits erwähnt. Da dies aber im ursprünglichen Datenmodell überhaupt nicht gespeichert werden konnte, ignorieren wir das für unsere weiteren Betrachtungen.

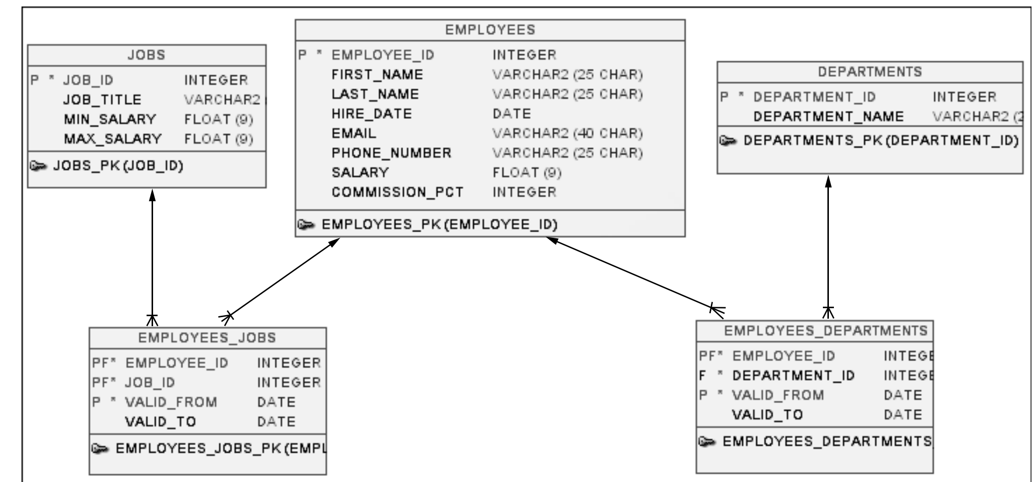

Abbildung 26.2 Umbau zu einem historisierenden Datenmodell

Im Übrigen gehört nun auch der Datumsbereich zur Join-Bedingung eindeutig dazu. Ein normaler Join zwischen den Tabellen lautet nun also (ich verwende hier einmal wegen der besseren Übersichtlichkeit die Oracle-proprietäre Join-Schreibweise):

```
SQL> select e.employee_id, e.first_name,
  2         e.last_name, e.hire_date,
  3         e.email, e.phone_number,
  4         e.salary, e.commission_pct,
  5         d.department_id, d.department_name,
  6         j.job_id, j.job_title
  7    from employees e, departments d, jobs j,
  8         employees_departments ed,
  9         employees_jobs ej
 10   where e.employee_id = ed.employee_id
 11     and ed.department_id = d.department_id
 12     and sysdate between ed.valid_from and ed.valid_to
 13     and e.employee_id = ej.employee_id
 14     and ej.job_id = j.job_id
 15     and sysdate between ej.valid_from and ej.valid_to;
```

Listing 26.24 Abfrage gegen ein historisierendes Datenmodell

Diese Abfrage ist natürlich auch nicht gerade einfach, aber immer gleich strukturiert. Wenn Sie nun wissen möchten, wie der Datenbestand zu einem gegebenen Zeit-

punkt aussah, müssen Sie lediglich sysdate durch Ihr gewünschtes Datum ersetzen. Sie erkennen aber auch: Für das gelegentliche Nachschlagen eines alten Datenbestands ist diese Vorgehensweise möglicherweise zu aufwendig, da ist die Modellierung des Benutzers HR dann der bessere Kompromiss. Denn es kommt hinzu: Auch in Tabelle DEPARTMENTS ist der Verweis auf LOCATION_ID konsequenterweise entfernt worden, da eine Abteilung durchaus umziehen und über die Zeit in unterschiedlichen Städten angesiedelt worden sein kann. Dieses Verfahren darüber hinaus aber auch noch von LOCATIONS aus auf COUNTRIES und von dort auf REGIONS zu wiederholen, machen wohl nur noch sehr grundsätzlich orientierte Datenmodellierer. Es mag unsinnig erscheinen, davon auszugehen, dass eine Stadt, die heute Land A angehört, morgen Land B angehören soll, aber da hätten Sie Ihre Rechnung wohl ohne den ehemaligen Ostblock gemacht … Ebenso verhält es sich natürlich auch bezüglich der Zuordnung von Ländern zu Verkaufsgebieten. Ich habe einmal ein Datenmodell gesehen, bei dem selbst die Zuordnung der Wochentage zu einer Woche ausgegliedert war (es könnte ja sein, dass die Woche irgendwann einmal 8 Tage haben wird …), muss aber gestehen, dass ich das nun wirklich für zu viel des Guten halten würde.

Ein solches Datenmodell hat also Auswirkungen auf die Joins. Ist dies geschafft, verhalten sich die Datenmodelle dann wieder ebenso wie die bereits jetzt im Schema HR angelegten Benutzer. Das eröffnet nette Möglichkeiten: Sehen wir uns doch an diesem Beispiel meine Empfehlung an, all diese Joins hinter Views zu verbergen. Dann könnte die hart codierte Angabe eines Datums (zum Beispiel sysdate) durch einen Join auf eine Zeitdimensionstabelle aus Abschnitt 26.1.4, »Analyse gegen eine Zeitdimension«, ersetzt werden, die Sie anschließend bei der Abfrage leicht filtern können. Damit wären die Views beliebig versionierbar:

```
SQL> create or replace view employees_history as
  2  select e.employee_id, e.first_name, e.last_name,
  3         e.hire_date, e.email, e.phone_number,
  4         e.salary, e.commission_pct,
  5         d.department_id, d.department_name,
  6         j.job_id, j.job_title,
  7         t.time_id
  8    from employees e, departments d, jobs j,
  9         employees_departments ed,
 10         employees_jobs ej,
 11         times t
 12   where e.employee_id = ed.employee_id
 13     and ed.department_id = d.department_id
 14     and t.time_id between ed.valid_from and ed.valid_to
 15     and e.employee_id = ej.employee_id
 16     and ej.job_id = j.job_id
 17     and t.time_id between ej.valid_from and ej.valid_to;
```



```
View wurde erstellt.

SQL> select first_name, last_name, job_title
  2    from employees_history
  3   where time_id = date '2004-05-15'

FIRST_NAME      LAST_NAME         JOB_TITLE
--------------- ----------------- ---------------------------
Lex             De Haan           Programmer
Neena           Kochhar           Accounting Manager
Shelley         Higgins           Accounting Manager
William         Gietz             Public Accountant
Hermann         Baer              Public Relations Representative
Susan           Mavris            Human Resources Representative
Jennifer        Whalen            Public Accountant
Daniel          Faviet            Accountant
Nancy           Greenberg         Finance Manager
Alexander       Khoo              Purchasing Clerk
Steven          King              President
Renske          Ladwig            Stock Clerk
Trenna          Rajs              Stock Clerk
Nandita         Sarchand          Shipping Clerk
Janette         King              Sales Representative
Sarah           Bell              Shipping Clerk
Michael         Hartstein         Marketing Representative
Patrick         Sully             Sales Representative
Ellen           Abel              Sales Representative
19 Zeilen ausgewählt.
```

Listing 26.25 Erweiterung zu einer versionierbaren View

Alternativ kann eine zweite View so angelegt werden, dass sie nicht auf TIMES zeigt, sondern hart über sysdate filtert. Diese View, die inhaltlich der ersten Abfrage entspräche, zeigt dann also stets den aktuellen Datenbestand. Ich habe in den Skripten zum Buch eine solche historisierende Version des Datenmodells von HR unter dem Benutzer HR_NEW erstellt. Wenn Sie mögen, können Sie dort die Beispiele nachvollziehen. Natürlich bleiben auch bei diesen Datenmodellen alle Varianten bezüglich der Speicherung von Datumsbereichen grundsätzlich bestehen, daher ist die Implementierung, die ich verwendet habe, nicht verbindlich. Generell hat ein solcher Umbau weitreichende Konsequenzen, denn wenn wir uns zu diesem Umbau entschließen, müssen wir die Altdaten in die neue Struktur migrieren. Auch die hierfür erforderlichen insert- und update-Anweisungen habe ich im Skript nachgereicht.

Ein wesentliches Problem der historisierenden Speicherung müssen wir uns allerdings noch ansehen: Wir verlieren eventuell unseren Primärschlüssel! Wie kommt das? Sehen wir uns an, wie die Änderung auf ein historisierendes Modell eine Tabelle beeinflusst. Wir starten mit einer einfachen Tabelle in folgender Form:

PROD_ID	PROD_NAME	PREIS
123	Tolles Auto	15000

Nun möchten wir diese Tabelle historisierend anlegen, und dabei haben wir natürlich den Preis im Blick, der sich über die Zeit ändern könnte. Also kommen wir zu folgender Erweiterung:

PROD_ID	PROD_NAME	PREIS_VON	PREIS_BIS	PREIS
123	Tolles Auto	15.02.2018	31.12.2011	15000
123	Tolles Auto	01.01.2012	31.12.2999	16500

Unser Artikel kann nun über die Zeit unterschiedliche Preise haben. Modellieren Sie dies so, wie oben gezeigt, ist klar, dass der Artikel nun seine Artikelnummer mehrfach in der Tabelle speichern muss. Damit ist diese Information aber nun keine Primärschlüsselinformation mehr. Die Änderung des Datenmodells hin zu einer historisierenden Speicherung macht also den Primärschlüssel unbrauchbar. Oder doch nicht? Sehen wir etwas genauer hin, stellen wir fest, dass diese Daten nun auch nicht mehr den Regeln der Normalform gehorchen, denn der Preis ist nicht nur vom Primärschlüssel, sondern auch vom Zeitbereich abhängig.

Es handelt sich hierbei also um eine funktionale Abhängigkeit des Preises von den Spalten `PREIS_VON` und `PREIS_BIS`. Den Normalisierungsregeln zufolge muss daher der Preis in eine eigene Relation ausgelagert werden. In dieser Tabelle wären die Spalten `PROD_ID` und (mindestens) `PREIS_VON` Primärschlüsselspalten, die Spalte `PROD_ID` dieser Tabelle würde dann als Fremdschlüssel die Spalte `PROD_ID` unserer ursprünglichen Tabelle referenzieren. Ein solcher Aufwand lohnt sich, wenn sich der Preis relativ häufig ändert. Ist dies nicht der Fall und ist zudem sichergestellt, dass sich der Preis nicht rückwirkend ändern kann, könnte man auch eine andere Modellierung in Betracht ziehen:

ID	PROD_ID	PROD_NAME	PREIS_VON	PREIS_BIS	PREIS
1	123	Tolles Auto	15.02.2018	31.12.2011	15000
2	123	Tolles Auto	01.01.2012	31.12.2999	16500

In dieser Variante haben wir also eine neue, technische `ID` eingeführt, unter der nun der Artikel eindeutig referenziert werden kann. Die Forderung, dass sich die Preise nicht rückwirkend ändern dürfen, ergibt sich daraus, dass in diesem Fall bereits bestehende Referenzen auf den Schlüssel 1 durch eine Referenz auf Schlüssel 2 ersetzt werden müssten, und das in Abhängigkeit von einem Datum, das Sie möglicherweise nicht einmal im Datenmodell gespeichert haben. Oracle unterstützt ein solches Verfahren, das man *kaskadierendes Update* nennt, nicht, weil es als extrem unsauber gilt. Zudem sehen Sie schon an der Wiederholung des Produktnamens, dass hier Redundanz vorliegt, die wir auf diese Weise nur in sehr speziellen Datenmodellen akzeptieren, nämlich in Dimensionstabellen von Datenwarenhäusern.

In OLTP-Umgebungen machen Sie die Erfahrung, dass historisierende Datenmodelle stets auch mehr Aufwand an Tabellen, Joins und Views nach sich ziehen. Hier wird wieder sehr schön sichtbar, was gemeint ist, wenn behauptet wird, der beste und schnellste Weg, etwas zu tun, sei, es nicht zu tun. Wenn Sie aber diese Funktionalität benötigen, kommen Sie um den damit verbundenen Aufwand nicht herum.

26.2.3 Weitere historisierende Datenmodellierungsoptionen

Gehen wir einige Schritte auf unserem Weg weiter. Wenn die Tabellen intensiv mit historisierenden Möglichkeiten ausgestattet werden sollen, stellt sich die Frage, wie das alles noch sinnvoll verwaltet werden kann. Bei der Analyse stellt sich heraus, dass durchaus nicht alle Attribute einer Tabelle historisiert werden müssen, andere dagegen schon. Was tun? Eine Lösung besteht darin, die Tabelle in zwei Teiltabellen zu trennen, deren eine die unbeweglichen und die andere die historisierenden Daten enthält. Bei einem Mitarbeiter könnten das Einstelldatum, das Geburtsdatum oder andere Daten in die erste, sein Name (Heirat, Scheidung ...), seine Adresse etc. in die zweite Tabelle gehören. Die erste Tabelle enthält die Stamm- und die zweite Tabelle die Referenzdaten, die historisierend gespeichert werden. Alle Fremdschlüsselbeziehungen referenzieren die Stammtabelle, die Historisierung erfolgt sozusagen unsichtbar hinter dieser Tabelle.

Merke

Datenmodelle, die auf diese Weise historisieren, werden auch als Hub und Satellit oder auch als Kopf- und Referenztabelle bezeichnet. Diese Art der Historisierung findet sich häufiger in Datenwarenhäusern, da sich durch dieses System auch Änderungen des Datenmodells nachzeichnen lassen: Werden einer Tabelle Spalten hinzugefügt, können diese in einen neuen Satelliten eingefügt werden.

So weit, so gut. Setzen wir dieses Datenmodell für den Benutzer `SCOTT` um, ergibt sich eine Tabellenlandschaft wie in Abbildung 26.3.

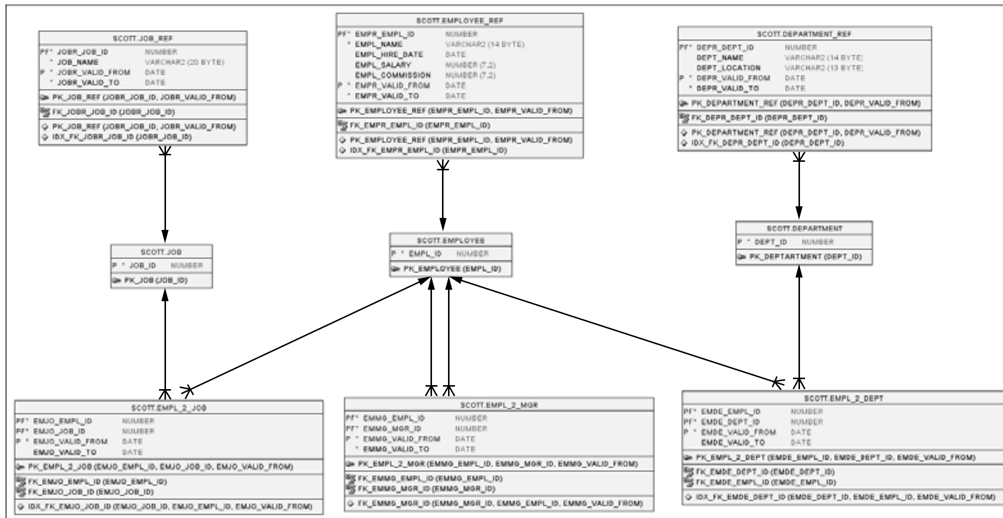

Abbildung 26.3 So komplex kann einfach sein: historisierendes Datenmodell für SCOTT

Wir haben nun ein Datenmodell, das die aufgezeigten Schwächen des Benutzers SCOTT ausgeräumt hat: Die Beziehung eines Mitarbeiters zu seinem Gehalt, Vorgesetzten, Beruf und seiner Abteilung sind in m:n-Tabellen ausgelagert, Abteilungen und Jobs können umbenannt und dennoch historisch korrekt zugeordnet werden. Alle Relationen haben nun eigene Gültigkeitsdaten, die sich überlappen können (wohl eher: die sich überlappen werden). Nur, damit Sie sich zurechtfinden: Die ursprünglichen Tabellen EMP und DEPT sind nun als Tabellen EMPLOYEE und DEPARTMENT angelegt und entvölkert worden. In unserem Beispiel sind lediglich die Primärschlüssel übrig geblieben, weil wir meinen, dass sich alles andere ändern könnte. Wäre dies nicht so, würden wir einige Spalten aus den zugehörigen Tabellen EMPLOYEE_REF bzw. DEPARTMENT_REF in die Stammtabelle verlagern. Ändert sich der Name des Mitarbeiters oder der Abteilungsname, kann dies nun in eigenen Datensätzen ausgelagert werden. Wollten Sie die Zuordnung eines Mitarbeiters zu seinem Gehalt von den anderen Dingen trennen, müssten Sie noch mehr Zuordnungstabellen einfügen, das habe ich uns hier erspart. Die Verbindung eines Mitarbeiters zu seinem Beruf und zu seiner Abteilung ist in eigene Tabellen ausgelagert, auch die Berufe können nun umbenannt werden, ohne dass dies die Beziehung ändert, denn auch die alten Berufsbezeichnungen werden historisierend gespeichert.

Wenn wir mit einem solchen Datenmodell arbeiten möchten, wäre die erste Anforderung, die aktuellen Daten aus EMP wiederherzustellen. Das ginge nun wie folgt:

```
SQL> select e.empl_id empno,
  2         er.empl_name ename,
  3         jr.job_name job,
  4         em.emmg_mgr_id mgr,
  5         er.empl_hire_date hiredate,
  6         er.empl_salary sal,
  7         er.empl_commission comm,
  8         ed.emde_dept_id deptno
  9    from employee e
 10    join employee_ref er
 11      on e.empl_id = er.empr_empl_id
 12     and sysdate between er.empr_valid_from and er.empr_valid_to
 13    join empl_2_mgr em
 14      on e.empl_id = em.emmg_empl_id
 15     and sysdate between em.emmg_valid_from and em.emmg_valid_to
 16    join empl_2_job ej
 17      on e.empl_id = ej.emjo_empl_id
 18     and sysdate between ej.emjo_valid_from and ej.emjo_valid_to
 19    join job j
 20      on ej.emjo_job_id = j.job_id
 21    join job_ref jr
 22      on j.job_id = jr.jobr_job_id
 23     and sysdate between jr.jobr_valid_from and jr.jobr_valid_to
 24    join empl_2_dept ed
 25      on e.empl_id = ed.emde_empl_id
 26     and sysdate between ed.emde_valid_from and ed.emde_valid_to

EMPNO ENAME    JOB          MGR HIREDATE      SAL   COMM  DEPTNO
----- -------- ---------- ----- ---------- ------ ------ -------
 7369 SMITH    CLERK       7902 17.12.1980    800             20
 7499 ALLEN    SALESMAN    7698 20.02.1981   1600    300      30
 7521 WARD     SALESMAN    7698 22.02.1981   1250    500      30
 7566 JONES    MANAGER     7839 02.04.1981   2975             20
 7654 MARTIN   SALESMAN    7698 28.09.1981   1250   1400      30
 7698 BLAKE    MANAGER     7839 01.05.1981   2850             30
 7782 CLARK    MANAGER     7839 09.06.1981   2450             10
 7788 SCOTT    ANALYST     7566 19.04.1987   3000             20
 7839 KING     PRESIDENT   7839 17.11.1981   5000             10
 7844 TURNER   SALESMAN    7698 08.09.1981   1500      0      30
 7876 ADAMS    CLERK       7788 23.05.1987   1100             20
 7900 JAMES    CLERK       7698 03.12.1981    950             30
 7902 FORD     ANALYST     7566 03.12.1981   3000             20
 7934 MILLER   CLERK       7782 23.01.1982   1300             10
14 rows selected.
```

Listing 26.26 Die alten Daten sind wieder da …

Na? Noch alles im Griff? Sie sehen, warum so etwas nicht standardmäßig gemacht wird, sondern nur, wenn die fachlichen Anforderungen solche Datenmodelle dringend erfordern. Doch selbst dann werden im Regelfall Schnittstellen bereitgestellt, die einfacher abzufragen sind.

Doch das ist noch nicht alles: Eventuell haben einige dieser Intervalle auch Lücken, weil zum Beispiel der Mitarbeiter für eine gewisse Zeit nicht im Unternehmen war, was dadurch ausgedrückt wurde, dass er in dieser Zeit nicht Teil einer Abteilung war, doch hat man vergessen (oder nicht implementiert), in dieser Zeit auch die Unterstellung zu einem Vorgesetzten sowie die Zuordnung zum Beruf abzugrenzen. Später taucht er dann wieder in einer Abteilung auf. Es können sich jetzt wirklich witzige Situationen ergeben. Einem Blog von Philipp Salvisberg (*http://tinyurl.com/nhcj7pf*) habe ich folgende Darstellung in Abbildung 26.4 entnommen, der zu dieser Abbildung und einem ganz ähnlichen Datenmodell wie unserem auch die zugehörige Abfrage entwickelt.

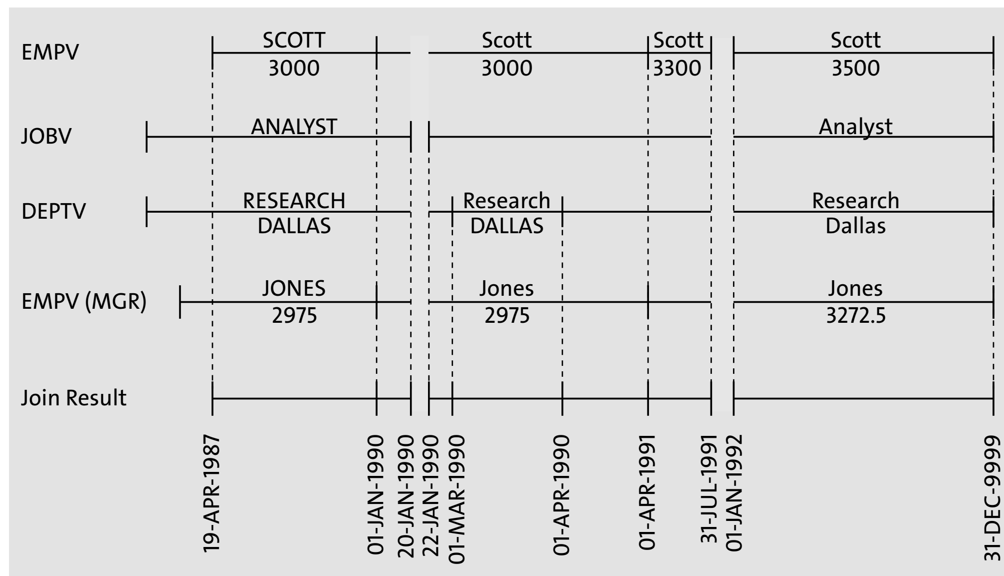

Abbildung 26.4 Überlappende und zum Teil offene Intervalle

Versuchen Sie nun einmal herauszufinden, welcher Mitarbeiter wann und wo im Unternehmen unterwegs war, welchen Beruf, welches Gehalt etc. er hatte. Fassen wir zusammen: Man kann alles übertreiben. Datenmodelle sind im Regelfall kein Tummelplatz für Überzeugungstäter, sondern Kompromisse und Vereinfachungen bestimmen das Bild, damit die Abbildung der Realität auf ein Datenmodell noch eine handhabbare Anwendung zurücklässt. Dabei ist aber natürlich zu berücksichtigen, dass Anforderungen, die in jedem Fall erfüllt werden müssen, zum Teil eben auch zu komplexen Datenmodellen führen können. Einige Gedanken habe ich vorgestellt,

einige folgen noch. Doch sollte bei jeder Entscheidung für eine weitere zeitliche Abgrenzung immer auch im Blick gehalten werden, dass sich die Komplexität für SQL stark erhöht. Das Ziel ist eine arbeitsfähige Balance.

Wenn Sie sich selbst ein Bild verschaffen möchten: Im Skript zum Buch finden Sie die Anweisungen, um das Datenmodell aus Abbildung 26.4 zu erzeugen, sowie die Anweisung, um es abzufragen.

26.2.4 Bitemporale Datenmodelle

Unter der etwas kryptischen Bezeichnung *bitemporale Datenmodelle* verstehen Datenbanktheoretiker Datenbanken, die mit zwei Zeitbereichen parallel operieren. Der entscheidende Gedanke ist, dass es durchaus nicht unwichtig ist, zu unterscheiden, wann eine Datenänderung bekannt war und wann eine Datenänderung tatsächlich existiert hat. Das klingt zunächst nach einem spitzfindigen Problem, doch ist es das nicht. Stellen Sie sich vor, Sie schreiben einem Kunden eine Rechnung. Die Adresse entnehmen Sie einer Tabelle, die in der Lage ist, Ihnen zu sagen, von wann bis wann dieser Kunde wo gewohnt hat. Aus dieser Tabelle geht eindeutig hervor, dass Ihr Kunde derzeit (es ist Mai) in Köln wohnt. Sie schicken also die Rechnung nach Köln. Nun kommt diese Rechnung aber nicht an. Erst später stellt sich heraus, dass der Kunde bereits seit März in Stuttgart wohnt. Sie korrigieren nun den Stammdatensatz. Nun haben Sie aber keine Möglichkeit mehr, später zu ermitteln, warum die Rechnung im Mai nach Köln geschickt wurde, denn es ist nicht mehr ersichtlich, von welchem Datenbestand Sie im Mai ausgegangen sind.

Ein bitemporales Datenmodell speichert im Gegensatz hierzu zwei Datumsbereiche, die als Zeitbänder bezeichnet werden: Im ersten Zeitband hinterlegen Sie, wann die eingefügte Information fachlich bestanden hat, im zweiten Zeitband, wann Sie diese Information eingefügt haben. Man spricht hier von einem *fachlichen* und einem *transaktionellen* Zeitband. Wenn Sie nun die Rechnung schreiben, beziehen Sie sich auf das fachliche Zeitband, um herauszufinden, welche Daten derzeit gültig sind. Sollten Sie aber nachweisen müssen, warum im Mai die Rechnung nach Köln geschickt wurde, beziehen Sie sich auf das transaktionelle Zeitband, das zeigt, *wovon Sie im Mai ausgegangen* sind, als Sie die Rechnung damals gesendet haben. Am Beispiel wird klar, dass bei Anlage der Korrektur (der Kunde ist nach Stuttgart gezogen) transaktionelles und fachliches Zeitband abweichen werden: Das fachliche Zeitband zeigt, dass der Kunde seit März in Stuttgart wohnt (der bislang existierende Datensatz wird fachlich zu Ende Februar beendet), während das transaktionelle Datum im Juni beginnt, als Sie vom Umzug Ihres Kunden erfahren haben. Ob Sie den vorhergehenden Datensatz nun auch transaktionell beenden, ist unterschiedlich, ich empfehle dies.

Dieses Beispiel zeigt eine weitere Dimension der Komplexität auf: Wir müssen nicht nur in der Lage sein, die Daten der Datenbank korrekt darzustellen, wir müssen (je

nach Domäne natürlich, das ist nicht immer so) auch in der Lage sein, nachzuweisen, warum wir in der Vergangenheit inkorrekte Daten hatten. Datenbanken sind also in zwei Dimensionen zu historisieren: Einerseits bezüglich der gespeicherten Fakten, andererseits bezüglich unserer Kenntnis über diese Fakten. Ein spannendes Thema, finde ich.

Aus Sicht des SQL-Anwenders ist diese Unterscheidung nicht so sehr problematisch, weil je nach Abfragesituation entweder auf das eine oder das andere Intervall Bezug genommen werden muss. Allerdings stellen sich eine Reihe von Fragen, wenn es zum Beispiel um den Schutz, die Abgrenzung und die referenzielle Integrität solcher Daten geht. Stellen Sie sich ein Datenmodell vor, das alle Daten stets multitemporal historisierend speichert. Dann ist klar, dass keine einfache referenzielle Integrität mit Primär- und Fremdschlüssel mehr durchgesetzt werden kann, denn ein Vergleich eines Faktums über eine Schlüsselspalte allein reicht ja nun nicht mehr aus, es muss immer auch die Zeitdimension berücksichtigt werden, und das noch variabel bezüglich der Kenntnis über die Fakten und der Fakten selbst. Daraus ergibt sich, dass die Datenbank grundlegend anders arbeiten muss, wenn solche Konzepte unterstützt werden sollen. Derzeit sind hierzu zwar theoretische Überlegungen erfolgt, aber daraus noch keine praktisch verwertbaren Lösungen hervorgegangen. Bitemporale Datenmodelle existieren allerdings dennoch in entsprechenden Problemdomänen. Sie werden aber meist nicht durchgängig auf das gesamte Datenmodell abgebildet, sondern lediglich für die Bewegungsdaten eingesetzt, bei denen die Unterscheidung der beiden Fälle von Belang ist. Daher war es mir wichtig, aus diesem Blickwinkel zumindest die grundlegenden Gedanken zu diesem Themenkomplex erläutert zu haben.

Seit Version 12c hat diese Diskussion insofern neuen Schub erlangt, als Oracle das Feature *Temporal Validity* der Datenbank hinzugefügt hat. Darunter wird ein Bündel von Technologien zur Unterstützung solcher Datenmodelle verstanden, die im Zusammenhang mit dem zweiten Buzzword *Information Lifecycle Management (ILM)* dazu angetan scheinen, eine konsistente Lösung anzubieten. Das ist aber leider derzeit noch nicht der Fall, es fehlt eine ausgereifte Unterstützung durch entsprechende DML-Befehle, und zudem ist noch sehr viel Handarbeit erforderlich, die diese Ansätze nicht wesentlich eleganter, schneller oder sicherer erscheinen lassen als die Do-it-yourself-Methode, die wir in diesem Abschnitt besprochen haben. Das Problem ist eben vertrackt, daher steht auch keine schnelle Lösung zu erwarten. Weitere Informationen finden Sie bei Bedarf in der Online-Dokumentation zu den oben gegebenen Fachbegriffen.

Grob gesagt geht es bei Temporal Validity eher um die Verwaltung von Dokumenten, die über ihren gesamten Lebenszyklus (Erstellung, Revision, Veröffentlichung, Archivierung) verwaltet werden. In diesem Zusammenhang lassen sich neue Funktionen nutzen, um nur veröffentlichte Dokumente anzeigen zu lassen und Ähnliches mehr.

Kapitel 27
Speicherung hierarchischer Daten

Nachdem wir uns bereits angesehen haben, wie hierarchische Abfragen auf Daten durchgeführt werden, widmen wir uns in diesem Kapitel dem Bereich der Datenmodellierung zum Speichern von Hierarchien.

Dieser Abschnitt beleuchtet andere Formen der Speicherung von Hierarchien in Datenbanken, die über die einfache Speicherung in zwei Spalten hinausgehen. Wie wir gesehen haben, ist eine Speicherung einer Hierarchie in gerade einmal zwei Spalten aufgrund ihrer Einfachheit und Eleganz weit verbreitet. Sie hat aber auch gravierende Nachteile, denn zum einen kann ein hierarchischer Eintrag stets nur ein Vaterelement referenzieren, zum anderen kann nicht gespeichert werden, dass eine hierarchische Beziehung über eine gewisse Zeit bestanden hat. Das zweite Argument will sagen, dass die einfache Speicherung in zwei Spalten nicht historisierend sein kann, denn wenn ein Eintrag in einer Spalte geändert wird, ist die vorhergehende Information verloren.

27.1 Hierarchie mittels zusätzlicher Hierarchietabelle

Beide Probleme können wir lösen, indem wir die Hierarchie in eine eigene Tabelle auslagern, denn wir erkennen damit an, dass ein Element nicht nur zu einer hierarchischen Stufe gehören kann, sondern zu mehreren und daher die Modellierung einer `m:n`-Beziehung erforderlich ist:

```
SQL> create table employee as
  2  select empno, ename, job, sal, comm, hiredate, deptno
  3    from emp;
Tabelle wurde erstellt.

SQL> alter table emplyee add constraint
  2    pk_employee primary key(empno);
Tabelle wurde geändert.

SQL> create table employee_hierarchy(
  2    empno number,
```

```
  3    mgr number,
  4    valid_from date,
  5    valid_to date /* ab 12c: on null default date '2999-12-31' */,
  6    constraint pk_employee_hierarchy
         primary key(empno, mgr, valid_from),
  7    constraint fk_emph_empno foreign key(empno)
  8      references employee_tab(empno),
  9    constraint fk_emph_mgr foreign key(mgr)
 10      references employee_tab(empno)
 11  ) organization index;
Tabelle wurde erstellt.

SQL> insert into employee_hierarchy
  2  select empno, coalesce(mgr, empno) mgr,
  3         hiredate valid_from,
  4         date '2999-12-31' valid_to
  5    from emp;

14 Zeilen eingefügt.
```

Listing 27.1 Erstellung einer separaten Tabelle für die Hierarchie

Lassen Sie sich ein wenig Zeit, um die Skripte zu verstehen. Wir haben eine Tabelle als Kopie der Tabelle `EMP` angelegt, diesmal allerdings ohne die Spalte `MGR`. Die Beziehung zwischen Mitarbeiter und Manager habe ich in die Tabelle `EMPLOYEE_HIERARCHY` ausgelagert und um zwei Spalten mit einem Start- und Enddatum erweitert. Die Tabellen haben einige Constraints erhalten. Interessant ist vielleicht der Primärschlüssel über die Tabelle `EMPLOYEE_HIERARCHY`, der nicht nur die Tupel `EMP` und `MGR` umfasst, sondern auch noch das Startdatum. So ist es nun also möglich, einen Mitarbeiter mehrfach dem gleichen Manager zu unterstellen, vorausgesetzt, dies geschieht zu unterschiedlichen Zeiten. So weit ist dieses Datenmodell also exakt das, was ich in Abschnitt 26.2, »Historisierung und Logging«, über die Erstellung eines historisierenden (aber nicht bitemporalen) Datenmodells geschrieben habe. Ein Nachteil dieser Modellierung (die aber meiner Kenntnis nach ohne Programmierung nicht zu umgehen ist) besteht darin, dass nicht vermieden werden kann, dass sich zwei Intervalle überlappen. Damit meine ich, dass unsinnige, aber formal korrekte Einträge in der Datenbank möglich sind, wie in folgender Darstellung:

```
EMPNO MGR  VALID_FROM  VALID_TO
----- ---- ----------- ----------
 8100 7936  15.05.2006 31.12.2099
 8100 7936  16.09.2010 31.12.2099
```

Listing 27.2 Formal korrekte, aber unsinnige Einträge

Offensichtlich sind die beiden Einträge insofern falsch, als vergessen wurde, den älteren Eintrag nach hinten zu begrenzen. Eine einfache Lösung für dieses Problem gibt es nicht, denn auch, wenn Sie die Spalte `VALID_TO` in den Primärschlüssel einbeziehen, wäre diese Zeile gültig. Zudem könnte ja auch ein anderes Enddatum als gerade das Maximaldatum eingetragen worden sein, zum Beispiel der `03.10.2010`. Die Daten wären auch dann formal korrekt, aber inhaltlich sinnlos. Was wir bräuchten, wäre eine Funktion, die einfach prüfen könnte, ob sich die Intervalle überlappen. Dann besteht aber das Problem, dass dies nicht durch einen einfachen `check`-Constraint durchgeführt werden könnte, denn dieser kann nicht zeilenübergreifend prüfen (was im ISO-SQL-Standard übrigens geht und vielleicht durch zukünftige Datenbankversionen implementiert werden könnte). Das alles klingt nach einem Index, doch wie indizieren Sie solche Dauern? Ich habe bislang keinen befriedigenden Weg zur Lösung dieses Problems gefunden, außer, einen Index selbst zu programmieren.

Lassen wir dieses Problem einmal beiseite und kommen zur hierarchischen Abfrage zurück. Zunächst einmal ist diese Form der Speicherung kein Problem, wir können nach wie vor mit einfachen `connect-by`-Abfragen die Hierarchie darstellen:

```
SQL>  select level,
  2          lpad('.', (level - 1) * 2, '.') || .ename name
  3     from employee e
  4     join employee_hierarchy h on e.empno = h.empno
  5    start with h.mgr = h.empno
  6  connect by nocycle prior h.empno = h.mgr;

     LEVEL NAME
---------- -------------
         1 KING
         2 ..JONES
         3 ....SCOTT
         4 ......ADAMS
         3 ....FORD
         4 ......SMITH
         2 ..BLAKE
         3 ....ALLEN
         3 ....WARD
         3 ....MARTIN
         3 ....TURNER
         3 ....JAMES
         2 ..CLARK
         3 ....MILLER

14 Zeilen ausgewählt.
```

Listing 27.3 Eine einfache hierarchische Abfrage auf dem neuen Datenmodell

Beachten Sie bitte die Konsequenz, die ich aus der Tatsache ziehen musste, dass mein Einstiegspunkt in den Baum nun nicht mehr ein `null`-Wert ist (was nicht erlaubt wäre, weil die `MGR`-Spalte Teil des Primärschlüssels ist), sondern `EMPNO` entspricht. Es ist auch nicht möglich, einfach zum Beispiel den Wert `0` einzusetzen, denn dann wäre der Fremdschlüssel auf `EMPLOYEE` verletzt. Bei einer »normalen« hierarchischen Abfrage erzeugt diese Konvention einen Zirkelschluss, den wir durch die Klausel `nocycle` ausschließen müssen.

Nun aber lassen wir unser Datenmodell zeigen, was es zusätzlich kann. Dazu füge ich zwei weitere Zeilen an und ändere eine dritte. Durch diese Änderungen ist nun ein Mitarbeiter über die Zeit einem anderen Manager unterstellt worden und ein anderer Mitarbeiter untersteht zwei Managern:

```
-- Alte Unterstellung beenden
SQL> update employee_hierarchy
  2     set valid_to = date '2010-05-15' - interval '1' second
  3   where empno = 7499;
1 Zeile wurde aktualisiert.

-- Neue Unterstellung einfügen
SQL> insert into employee_hierarchy
  2  values (7499, 7839, date '2010-05-15', date '2999-12-31');
1 Zeile wurde erstellt.

-- Doppelte Unterstellung einfügen
SQL> insert into employee_hierarchy
  2  select empno, 7839, valid_from, valid_to
  3    from employee_hierarchy
  4   where empno = 7844;
1 Zeile wurde erstellt.

SQL> -- Abfrage des neuen Datenbestands
SQL>  select level, h.empno,
  2          lpad('.', (level-1)*2, '.') || e.ename name
  3     from employee e
  4     join employee_hierarchy h on e.empno = h.empno
  5    where sysdate between valid_from and valid_to
  6    start with h.mgr = h.empno
  7  connect by nocycle prior h.empno = h.mgr;

     LEVEL      EMPNO NAME
---------- ---------- --------------------
         1       7839 KING
         2       7499 ..ALLEN
         2       7566 ..JONES
         3       7788 ....SCOTT
         4       7876 ......ADAMS
         3       7902 ....FORD
         4       7369 ......SMITH
         2       7698 ..BLAKE
         3       7521 ....WARD
         3       7654 ....MARTIN
         3       7844 ....TURNER
         3       7900 ....JAMES
         2       7782 ..CLARK
         3       7934 ....MILLER
         2       7844 ..TURNER

15 Zeilen ausgewählt.
```


Listing 27.4 Erweiterung des Datenbestandes, um die erweiterten Fähigkeiten zu demonstrieren

Wenn Sie vergleichen möchten, ist `ALLEN` nun direkt `KING` unterstellt, was er vorher nicht war. Zudem ist `TURNER` nun sowohl `KING` als auch (nach wie vor) `BLAKE` unterstellt. Ersetzen Sie `sysdate` in der `where`-Klausel durch `date '2010-05-14'`, werden Sie sehen, dass die Abfrage nun wieder `ALLEN` als Untergebenen von `BLAKE` führt.

27.2 Closure Table

Über die Datenmodellierung mit diesen zwei Spalten hinaus existieren noch andere Verfahren, die hauptsächlich dem Zweck dienen, die lesenden Zugriffe auf Hierarchien zu vereinfachen. Ein Verfahren ist mir als *Closure Table* bekannt und dem Entwurf von oben sehr ähnlich. Als Erweiterung unseres Datenmodells wird nun eine Tabelle gepflegt, die alle möglichen Verbindungen zwischen zwei Elementen speichert. Zunächst einmal hat jedes Element eine Verbindung mit sich selbst, ähnlich unserem Wurzelelement `KING`. Dann wird in der Tabelle aber auch jede mögliche Beziehung eines Knotens zu seinen Vorgängern gespeichert, also nicht nur zu seinem direkten Vorgänger, sondern zu allen in der Hierarchie. Damit uns dies ein wenig klarer wird, erzeugen wir uns eine solche Tabelle in ihrer einfachsten möglichen Form selbst:

```
SQL> create table hierarchy_closure as
  2  select h.ename mitarbeiter, e.ename vorgaenger
  3    from emp e
  4    join ( select ename,
```

```
  5                 sys_connect_by_path(ename, '|') pfad
  6             from emp
  7            start with mgr is null
  8          connect by prior empno = mgr) h
  9      on instr(h.pfad, e.ename) > 0;
Tabelle wurde erstellt.
```

Listing 27.5 Erstellung einer Closure-Tabelle

Falls Sie nun das Gefühl bekommen, das Lesen des Buches »stocke« etwas, weil Sie zwischendurch mehr Zeit benötigen, eine SQL-Anweisung zu verstehen, als vorher, dann glauben Sie mir bitte eins: Ich weiß. Dazu sind die Abfragen ja da …

Die Abfrage erzeugt eine hierarchische Darstellung, in der jeder Mitarbeiter mit sich selbst und jedem seiner Vorgänger verbunden ist. Für den Mitarbeiter ADAMS sieht das beispielsweise so aus:

```
SQL> select *
  2    from hierarchy_closure
  3   where mitarbeiter = 'ADAMS';

MITARBEITE VORGAENGER
---------- ----------
ADAMS      JONES
ADAMS      SCOTT
ADAMS      KING
ADAMS      ADAMS
```

Listing 27.6 Ausschnitt der Tabelle HIERARCHY_CLOSURE für den Mitarbeiter ADAMS

Das hat für die Abfrage gewisse Vorteile, denn nun ist für jede mögliche Verbindung ein Datensatz in der Hierarchietabelle vorhanden, der über eine einfache Filterung identifiziert werden kann. In unserem Beispiel könnten wir uns vorstellen, dass der Mitarbeiter ADAMS, der SCOTT direkt unterstellt ist, nun einmal einen Eintrag in der Tabelle enthält, in der ADAMS auf sich selbst zeigt, dann aber auch auf SCOTT, wie bisher, und zusätzlich noch auf JONES, den Vorgesetzten von SCOTT, und auf KING. Möchte ich nun alle Untergebenen von JONES sehen, reicht es, alle Datensätze zu zeigen, bei denen in der Spalte VORGAENGER der Eintrag JONES enthalten ist. Dadurch ist keine rekursive Iteration über alle Zeilen der Tabelle erforderlich, sondern es reicht ein einfacher (eventuell indexunterstützter) Scan auf die Spalte VORGAENGER. Das klingt gut und wird in speziellen Szenarien sicher auch Probleme lösen helfen. Die andere Frage ist natürlich, wie sich ein solches Datenmodell beim Schreiben verhält. Zunächst einmal muss beim Einfügen eines Datensatzes in die Tabelle EMP nun sichergestellt sein, dass dieser neue Benutzer, verknüpft mit sich selbst, auch in die Hierarchietabelle

kommt, verbunden mit einem Datensatz für jeden Vorgänger in der Hierarchie. Das können wir über eine union-all-Abfrage noch relativ einfach sicherstellen, wie in der folgenden Anweisung in Listing 27.7, in der wir den neuen Mitarbeiter WILLIAMS unter SCOTT einfügen:

```
SQL> insert into hierarchy_closure
  2  select 'WILLIAMS', vorgaenger
  3    from hierarchy_closure
  4   where mitarbeiter = 'SCOTT'
  5  union all
  6  select 'WILLIAMS', 'WILLIAMS' from dual;
4 Zeilen wurden erstellt.

SQL> select *
  2    from hierarchy_closure
  3   where mitarbeiter = 'WILLIAMS';

MITARBEITE VORGAENGER
---------- ----------
WILLIAMS   JONES
WILLIAMS   SCOTT
WILLIAMS   KING
WILLIAMS   WILLIAMS
```

Listing 27.7 Eine Einfügeanweisung eines neuen Mitarbeiters

Hier wird auch klar, warum wir einen Eintrag in der Tabelle benötigen, in der ein Mitarbeiter auf sich selbst zeigt, denn ansonsten würde der direkte Vorgesetzte durch die Abfrage in Listing 27.7 nicht gefunden.

Beim Löschen eines Eintrags gehen wir ähnlich vor und löschen alle Einträge, in denen der zu löschende Eintrag als Vorgänger enthalten ist. Lediglich das Aktualisieren eines Eintrags ist schwierig, wenn nämlich ein Mitarbeiter an einer anderen Stelle des Baums integriert werden soll. Hier dürfte es das Einfachste sein, den Mitarbeiter zunächst zu löschen und anschließend neu anzulegen. Allerdings gibt es ein größeres Problem, wenn der Mitarbeiter *und alle seine Untergebenen* an eine andere Stelle des Baums wandern sollen, denn nun sind Zeilen in der Tabelle betroffen, die sich nicht auf den Mitarbeiter beziehen, der eigentlich umzieht. Ein Nachfolger von SCOTT hat ja auch eine eigene Referenz auf den Vorgänger von SCOTT, ohne SCOTT jedoch explizit zu benennen. Das ist nun weniger schön, denn wir müssen in zwei Schritten vorgehen: Zunächst müssen alle hierarchischen Einträge entfernt werden, die sich nicht auf das direkte Verhältnis zwischen SCOTT, der umziehen soll, und seinen Untergebenen beziehen. Ich zeige die Lösch- und die anschließende Einfügeanweisung hier als select-Abfrage, um klarzumachen, welche Zeilen betroffen sein werden:

```
SQL> select *
  2   from hierarchy_closure
  3  where vorgaenger in (
  4        select vorgaenger
  5          from hierarchy_closure
  6         where mitarbeiter = 'SCOTT')
  7    and mitarbeiter in (
  8        select mitarbeiter
  9          from hierarchy_closure
 10         where vorgaenger = 'SCOTT'
 11           and mitarbeiter != vorgaenger);

MITARBEITE VORGAENGER
---------- ----------
WILLIAMS   JONES
ADAMS      JONES
WILLIAMS   SCOTT
ADAMS      SCOTT
WILLIAMS   KING
ADAMS      KING
6 Zeilen ausgewählt.
```

Listing 27.8 Löschen der überflüssigen Einträge

Anschließend müssen wir dafür sorgen, dass die Einträge (und zwar alle) am neuen Standort korrekt verdrahtet werden. Dies erfordert einen Cross-Join auf dieselbe Tabelle:

```
SQL> select n.mitarbeiter, v.vorgaenger
  2   from hierarchy_closure n
  3  cross join
  4        hierarchy_closure v
  5  where n.vorgaenger = 'SCOTT'
  6    and v.mitarbeiter = 'MARTIN';

MITARBEITE VORGAENGER
---------- ----------
SCOTT      MARTIN
SCOTT      BLAKE
SCOTT      KING
ADAMS      MARTIN
ADAMS      BLAKE
ADAMS      KING
WILLIAMS   MARTIN
```



```
WILLIAMS   BLAKE
WILLIAMS   KING
9 Zeilen ausgewählt.
```

Listing 27.9 Erneute Verbindungsaufnahme der beteiligten Mitarbeiter

Ehrlich? If it's not a simple solution, it might be the wrong solution ...

Es kommen noch deutlich komplexere Probleme hinzu. Wie heißt mein direkter Vorgänger? Keine Ahnung, denn da ich Beziehungen zu all meinen Vorgängern unterhalte, kann ich nicht sagen, »wie weit weg« diese sind. Dazu müssten wir zusätzlich noch anzeigen, wie weit ein Kontakt weg ist. Dann: Wie finde ich die Geschwister auf meiner Ebene? Auch das ginge nur mit einer Anzeige der Pfadlänge. Diese jedoch ist so einfach nun auch wieder nicht zu ermitteln, denn einfaches SQL schlägt hier fehl. Wir sehen also erhebliche Aufwände beim Schreiben und Editieren solcher Baumstrukturen, allerdings auch einen Vorteil beim Lesen. Ob dieser allerdings rechtfertigt, auf eine Lösung wie die `connect-by`-Abfrage oder die rekursive `with`-Klausel zu verzichten, vermag ich nicht zu beantworten. Vielleicht finden sich Anwendungsbereiche in extrem großen Baumstrukturen. Aber machen wir uns nichts vor: In solch extrem großen Baumstrukturen ist eine solche Tabelle nicht nur extrem, sondern geradezu exorbitant groß, denn die Anzahl der Einträge explodiert mit der Schachtelungstiefe geradezu.

27.3 Weitere Modelle

Ein weiteres Modell zur Speicherung von Hierarchien in Datenbanken besteht darin, jedem Element der Hierarchie zwei Zahlen mitzugeben, die gemeinhin als *linke* und *rechte Grenze* bezeichnet werden. Die Idee dahinter ist, dass wir einen hierarchischen Baum zunächst in die Tiefe eines Zweiges abgehen und dabei jedem Element »links« eine inkrementierende Zahl verpassen. Ich habe einmal versucht, das in Abbildung 27.1 zu visualisieren.

Wir starten also am Beginn des Baums, dem wir links eine `1` verpassen, und gehen zum ersten Kindelement, das dann links eine `2` erhält. So gehen wir diesen Teil des Baums nach unten. Sind wir unten angelangt, vergeben wir diesem Element rechts die nächste Zahl und gehen von dort zunächst zu unseren Geschwistern, denen wir links und anschließend rechts die nächsten Zahlen zuweisen. Sind keine weiteren Geschwister vorhanden, gehen wir eine Ebene nach oben, weisen dem Element, aus dem wir vorher gekommen sind, die nächste Zahl zu und gehen dann zum nächsten Geschwister dieses Elements (und eventuell seiner Kinder). Irgendwann haben wir auf diese Weise wieder unser Ausgangselement erreicht (Sie erinnern sich vielleicht noch, dass sich dieses Verfahren `depths first` nennt). Nun haben wir eine Speiche-

rungsform, die uns die Analyse des Baums ermöglicht. Im Übrigen ist es nicht erforderlich, dass die Zahlen für die linke und rechte Grenze lückenlos aufeinanderfolgen, es können beliebige Zahlen verwendet werden, solange sie stetig steigen.

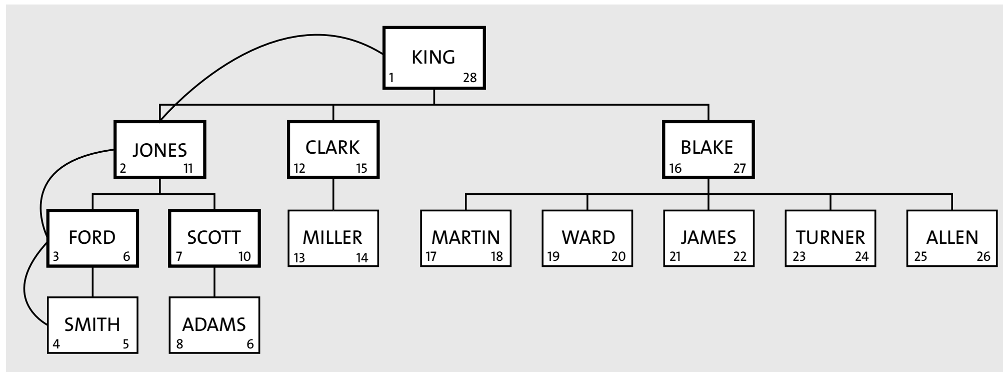

Abbildung 27.1 Abbildung der Hierarchie mit rechter und linker Grenze

Wieder erkennen wir Vorteile beim Lesen solcher Baumstrukturen, denn die Extraktion eines Teilbaums ist ganz einfach, da die linke oder rechte Grenze des Wurzelelements des Teilbaums einfach zwischen der linken und rechten Grenze seines Vorgängers liegen muss. Um also `JONES` und alle seine Untergebenen zu ermitteln, reicht es, zu fragen:

```
select *
  from emp_hierarchy
 where linke_grenze between 2 and 11;
```

Listing 27.10 Pseudoanweisung zur Ermittlung eines Teilbaums

Wir machen uns natürlich auch hier wieder zunutze, dass der `between`-Vergleich die linke und rechte Grenze enthält, daher wird in unserem Fall `JONES` mit ausgewählt. Natürlich können die hart codierten Grenzen auch über eine Unterabfrage dynamisch ermittelt werden. Das ist wohl der Fokus dieser Modellierung, hier funktioniert dies sehr gut. Allerdings sind weitere Fragen schon schwerer zu beantworten:

- Wie heißt mein direkter Vorgänger, wie mein direkter Nachfolger?
- Was sind meine Geschwisterelemente?
- Welche Elemente sind meine Vorgänger, wie lautet der Pfad?

Gar nicht nachdenken möchte ich im Übrigen darüber, wie hoch der Aufwand der Ermittlung der linken und rechten Grenzen ist, insbesondere, wenn nachträglich Elemente in den Baum eingehängt werden können oder wenn sich der Baum häufig erweitert. In all diesen Fällen müssen im Regelfall alle Elemente des Baums neue

linke und rechte Grenzen erhalten. Ein Vorteil ergibt sich allerdings doch noch: Wenn Sie einfach nur ein Element löschen, werden dessen Kindelemente automatisch Kinder des Vorgängers des gelöschten Elements. Ob das allerdings ein wirklicher Anwendungsfall ist? Ich weiß nicht recht.

Ein anderes Modell speichert die IDs aller Vorgänger, ähnlich, wie wir das zum Beispiel von einem Dateipfad her kennen, als Zeichenkette. Dadurch wird ein ähnlicher Effekt erreicht wie in der Speicherung mit Hilfe einer Closure Table, denn nun ist die Verbindung jedes Elements mit jedem anderen Element beim Element selbst gespeichert, nun allerdings denormalisiert als Zeichenkette mehrerer IDs. Die Nachteile dieser Speicherung liegen darin,

- dass die maximale Länge der Pfadtiefe nun durch die Anzahl der Zeichen der Spalte begrenzt wird, in die der Pfad gespeichert wird (und zudem von der Länge der IDs),
- dass keine referenzielle Integrität mehr durchgesetzt werden kann, es also möglich ist, auf Vorgänger zu verweisen, die es gar nicht gibt, und schließlich
- dass die Suche nach einzelnen Elementen nicht durch Indizes beschleunigt werden kann (Volltextindizierungsverfahren einmal ausgenommen), da in den »Pfaden« nur mit Mitteln der Textsuche gesucht werden kann.

27.4 Zusammenfassung

Es existieren noch weitere Vorschläge, Hierarchien in relationalen Datenbanken zu speichern. Leider ist kein mir bekannter Weg dabei, der die Probleme dieser Darstellungsform auf ein ähnlich intuitives Niveau heben könnte, wie dies zum Beispiel in XML der Fall ist: Dort werden Elemente einfach ineinander geschachtelt – und fertig. Das Umhängen, Aktualisieren und Ermitteln von Teilbäumen ist in solchen Datenstrukturen trivial. Relationale Datenbanken leiden daran, solche Strukturen auf gefällige Weise »flachklopfen« zu müssen, um sie in das relationale Konzept zu zwingen. Offensichtlich gelingt dies nicht so einfach.

Im Gegensatz zur einfachen Speicherung in zwei Tabellenspalten und der Abfrage über `connect-by`- oder rekursive `with`-Abfragen haben alle anderen Verfahren gravierende Nachteile bei der Bearbeitung der Hierarchie sowie der Beantwortung anderer als der vorgesehenen Fragestellungen. Daher empfiehlt sich das einfache Verfahren, vielleicht erweitert um die Auslagerung der Hierarchie in eine separate Tabelle, in Verbindung mit den vorgenannten Abfragen, solange Sie mit dieser Variante gut auskommen. Erst wenn die konkrete Situation und die tatsächlichen Geschäftsanforderungen Sie zwingen, für diese Lösung eine Alternative zu finden, ist eine der anderen Lösungen aus meiner Sicht sinnvoll.

Je nach Aufgabenstellung könnte aber auch ein ganz anderer Vorschlag von Interesse sein: Vielleicht ziehen Sie einen Spezialisten für Hierarchien zurate? Damit meine ich im Moment keinen externen Berater, sondern – ganz einfach – XML! Die Datenbank unterstützt die Indizierung, Bearbeitung und Auswertung von XML durch vielfältige Technologien. Warum sollte im konkreten Einsatzfall eine Speicherung hierarchischer Strukturen in einem hierarchischem Datenformat nicht die einfachste und effizienteste Speicherungsform darstellen?